TRS-80 INFORMATION SERIES - VOLUME VII

Lewis Rosenfelder

BASIC DISK I/O FASTER AND BETTER T.M.

& OTHER MYSTERIES

TRS-80® Random and Sequential File Programming: Beginner, Intermediate, Advanced

TRS-80 INFORMATION SERIES - VOLUME VII

Lewis Rosenfelder

BASIC DISK I/O FASTER AND BETTER™

& OTHER MYSTERIES

Editor — Carl Evans
Technical Editor — Gary Camp
Assistant Editor — Charles Trapp
Typesetting Software — David Moore
Graphics — D. J. Smith
Production — Cindy Hall

First Edition
First Printing, August 1983
Printed in the United States of America

ISBN 0 936200 25 1

Published by
IJG Inc
1953 West
11th Street
Upland,CA
91786 (714)
946-5805

IMPORTANT

Read This Notice

Any software information is used at your own risk. Neither the PUBLISHER nor the AUTHOR assumes any responsibility or liability for loss or damages caused or alleged to be caused, directly or indirectly, by applying any information or alteration to software described in this book, including but not limited to: any interruption of service, loss of business, anticipatory profits or consequential damages resulting from the use or operation of such software information or alterations. Also, no patent liability is assumed with respect to the use of the information contained herein. While every precaution has been taken in the preparation of this book, the PUBLISHER and the AUTHOR assume no responsibility for errors or omissions. The reader is the sole judge of his or her skill and ability to use the information contained in this book.

Contents

Chapter Ten

Chapter Eleven

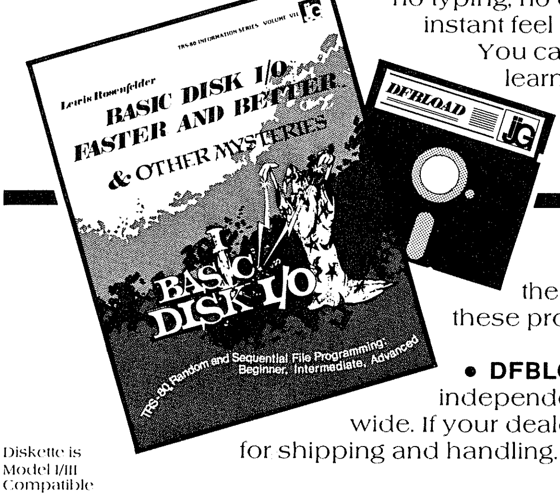
Chapter Twelve

Chapter Thirteen

Chapter Fourteen

Chapter Fifteen

Chapter Sixteen


TRS-80 INFORMATION SERIES VOLUME VII
Lewis Rosenfelder
BASIC DISK I/O FASTER AND BETTER & OTHER MYSTERIES
BASIC DISK I/O
TRS-80 Random and Sequential File Programming: Beginner, Intermediate, Advanced
DFBLOAD
JG

Introduction

How This Book Can Help You

This book is about a capability that is as important to your computer as the ability to read and write is to you and me. In fact that's what we'll be talking about — reading and writing. Like you and me, your computer has a brain with a memory. Just as we can open a book and turn to any page to refresh our memory, the computer, in its way, can do the same thing. With a pencil or pen we can write notations on any page in a book for later reference. Your computer, by following the commands you give it, can record information on any page of its memory for future reference or computations.

Magnetic disks are the "books" that give your computer the human-like ability to read and write information of all kinds. In fact, if we want a more accurate analogy, disks are like bookcases, because a disk can be, and usually is, subdivided to contain multiple files of often unrelated information. Data storage on diskettes takes your computer beyond its basic computational capabilities to open a whole world of possibilities in accounting, inventory, scientific, engineering, information handling and word-processing applications.

My goal is to show you practical ways to take advantage of the real power of disk programming, whether you intend to write new programs, or improve programs that you already have. Radio Shack's TRS-80 is a fantastic computer — but as you shall see, the programming techniques you select and adopt can make it faster and better than you ever thought possible.

This book is a by-product of my day-to-day programming efforts for local small businesses using the TRS-80 computer. The techniques we shall discuss have been used successfully in sophisticated programs and have undergone many modifications and improvements over the past few years. My goal is to be practical and real-world oriented.

We won't cover anything that doesn't have a definite purpose and application or dwell on subjects just for their "academic" value alone. On the other hand, I've taken great pains to explain everything in detail so you can quickly gain a complete understanding. For each technique, we'll go beyond how it works and what it does — to the more important level of why the technique is designed as it is, and how the technique will be useful to you.

If you are new to disk programming, this book is for you. If you are a professional who earns his living programming, this book is for you, too. Even if you are simply a Radio Shack computer user and wish to do no programming yourself, certain sections of this book

(Chapters 1, 2, 3 and 16) teach "survival technieques" that can protect your valuable data and time, while making you more self-sufficient with your computer.

How can one book satisfy such a wide audience? In the first few chapters, we'll start at ground-zero and cover all the disk commands, one-by-one. If you are a beginner, you will be learning "how." If you are at an intermediate level, you'll begin to get a better understanding of "why." Most of the book, though, is devoted to "plug-in" modules that can be integrated into any program by beginners and professionals.

Just as a professional carpenter may use "pre-fabricated" components to build a house efficiently — this book is full of pre-written "standard" routines that can help you build a program efficiently. And each is fully explained so that you can make modifications and enhancements where you wish.

There are many books available about programming, but this one is different in many ways. Some are too general — trying to cover every make and model of computer on the market. Others will lead you down a winding path, showing every solution to a problem until they get to the "best" solution. Many will discuss a programming technique, such as how to add information to a disk, and then "cop-out" at the very point where you need help, like how to delete that information. Some books are too conservative in their appoach. They show you the "safe" way, but they never remove the "training wheels" to show you the fast, efficient and flexible ways of doing things. Many books totally ignore real-world operating conditions — like what happens if a diskette is inserted upside-down or what to do if a disk file becomes magnetically damaged. I've given my best effort to avoid committing any of these sins. I want this book to be one of the most useful additions to your library — and one that you'll refer back to again and again. I use this information daily, and I trust you'll find it as important to you as it is to me.

This book is my second offering in the "other mysteries" series. Its companion, *BASIC Faster and Better and Other Mysteries*, covers a wide range of programming tricks and techniques that can dramatically improve the speed and power of your computer in areas such as memory management, keyboard and video display handling, searching and sorting, data compression and mathematical computations. In this book, we will "borrow" a few items from *BASIC Faster and Better*, where they are fully explained. I recommend that you buy a copy — but to be fair to you, this book can stand alone.

Learning Disk Programming

If you are at a beginning level, this book will get you into disk programming the fastest possible way. We won't review the simple BASIC commands, so I suggest that you at least get your feet wet by going through the instructional manuals provided with your computer. Type in the examples they show, try a few modifications, and create a few short programs of your own. They don't need to be especially practical at this point. Just get familiar with the simple capabilities of inputting data, displaying and printing it, doing simple arithmetic, using IF-THEN logic, FOR-NEXT loops, subroutines, GOTO's and arrays. You'll find that your computer is the best teacher, especially when you are willing to experiment with it. It will tell you when you've made an error. When you write even a simple program or routine that works as you intended, the satisfaction and feeling of power it gives you spurs you on to try even more.

The first few chapters of this book introduce you to the disk commands with short, simple programs. Type them in exactly as they are shown. I'll be pointing out the reasons for each step, calling your attention to the things you should notice, and filling you in on some of the "behind-the-scenes" work that your computer is doing. I want you to know more than simply how to do it. The deeper understanding you'll get will make it possible for you to take advantage of the real speed and power of your computer and use it more efficiently.

Subroutines and Handlers

Once you've gotten a basic understanding of the disk commands, you'll begin to see ways to apply them. You will also begin to appreciate some of the complexities that can arise when you wish to make your programs fast and reliable. If you write many programs, you will soon find yourself "re-inventing the wheel" for the procedures involved in disk access.

Most of this book is devoted to modules that you can merge into any program that uses disk access. A program is really a series of steps that you teach your computer to perform, along with rules for handling any decisions it must make. As the programmer, it's as if you were the top executive of a company. Once you've trained subordinate employees to perform the clerical tasks, and you have coordinated them, you are free to concentrate on the big picture. In programming, subroutines are the "subordinates" that let you be more productive by concentrating on the unique design of the particular application at hand.

A subroutine is a series of program lines that performs a particular task in a larger program. Whenever necessary, the larger program simply specifies the line number at which the subroutine begins, the task is performed, and upon completion of that task the main program continues.

Let's say that in line 150 of your program you need to retrieve some data from a disk file, but in lines 210 and 520 you also have the same requirement, perhaps for different purposes. If you put all the logic required for accessing a disk file, say at line 58200, your program, in lines 150, 210, and 520 can simply instruct the computer to GOSUB 58200. Then, each time control is passed to line 58200, the logic begins there, and follows through any subsequent program lines until a RETURN command is encountered, and the computer continues with the next command after the GOSUB.

Subroutines can be simple one-liners, or they can be large blocks of program logic. Quite often you will find subroutines that call other subroutines. It's all done with the idea of saving program memory by reducing duplicated logic, and saving program development time by avoiding rethinking, retyping and retesting.

A handler is a group of subroutines that work together to perform a major function within a program. If you've got *BASIC Faster and Better and Other Mysteries*, you are already familiar with some handlers that standardize the programming of formatted data entry screens. In this book we will introduce several disk file handlers. The "Random Disk File Handler," for example, gives you standardized programming methods for handling most random disk files. The "Detail File Handler" is a set of routines and procedures that you might wish to use for a more specialized (but common) programming requirement that we will discuss.

Handlers, as found in this book, provide several important benefits in disk programming:

- Standard line numbering and variable naming conventions. You'll always know what a particular line does and how any variable related to disk accessing is used because they are standardized from program to program and they are documented in this book.

- Standard data file layouts. This makes it easy for you to make files compatible among different programs and makes it possible to develop special programs to help you reconstruct damaged data files.

- Standardized messages to the computer operator. These simplify the writing of operating instructions. In the event of error conditions you can diagnose the exact problem more easily, and in most cases, they can tell the operator how to correct the problem and continue without restarting the program.

- Step-by-step procedures for adapting and using the handler subroutines in your programs. Program development becomes systematic. You know exactly what program lines will be required, which lines must be modified and how, and the lines that may be deleted if certain capabilities are unnecessary to your application.

- Line-by-line explanations. Each BASIC statement is fully explained in the line comments that accompany each handler. This makes the operation of the handler easy to understand so that anyone can learn from it or modify it when necessary. Your task of technical documentation is greatly simplified.

- Standard methods for calling the handler subroutines from your main program. You simply load the variables specified with the required data, and execute GOSUB's to the standard line numbers within the handler. Should you wish to adapt your program to another computer or a version of BASIC that uses different disk commands, you can simply modify the handler. The main logic of your program can often remain unchanged.

- Speed and memory efficiency. The handler subroutines are packed for maximum efficiency. That is, all unneeded spaces and comments have been removed, and they have been compressed to minimize the number of program lines required. The subroutines have been developed and improved over a period of years to provide better execution speeds.

- Easy program key-in. The subroutines and program lines are shown in this book exactly as they will appear on your video display. That is, if you have a TRS-80 Model I or III, you have 64 columns across the screen. The listings we've provided are also 64 columns wide so you can more easily catch your own typing errors when you are transcribing them into your computer. If you wish, you can purchase a diskette that has the programs and handler subroutines already keyed-in for you. See the back of this book (Appendix 4) for details!

USR Subroutines

For the most part, we will be discussing ways to take advantage of disk storage through the Disk BASIC programming language. BASIC lets us talk to the computer with commands and mathematical formulas that are quite consistent with the way we think and communicate. Disk BASIC provides additional commands that allow us to create and access disk files. Disk BASIC is easy to understand and use, convenient to modify and fast enough for most applications.

The real "brain" within the TRS-80 computer is the Z-80 microprocessor. If you compare the Z-80 to the engine in your car, BASIC provides the steering wheel, the comfortable upholstery and the the other features that make the "car" easy and safe to run. For operations where great speed is desirable, such as searching and sorting, USR subroutines give us a way to bypass BASIC temporarily, and then return when the task is completed. With a USR subroutine, we can talk directly to the Z-80 in its own "machine-language."

You don't need to know a single Z-80 command, and you don't need an editor/assembler program to implement the USR subroutines shown in this book. Just follow the instructions that are given and you'll be able to unlock the real speed and power of the TRS-80. By the way, all the USR subroutines in this book are relocatable. That is, you can load and execute them at any location in available memory without modification. Appendix 2 shows you how.

Getting Started

For every important technique we'll discuss in this book, I've provided a demonstration program. The demo programs are designed to be as short and simple as possible. The purpose of each is to let you test each routine to your satisfaction, to try out modifications, and to get a feel for the speed and capabilities the techniques provide. For the demo programs, I've avoided some of the operator-oriented features for fancy video display handling and error-trapping that would normally be present in "finished" programs just to keep the demo programs short, simple, and to the point.

A picture can be worth a thousand words. The demo programs in this book are worth a thousand words of explanation, plus perhaps, a thousand minutes avoided in hunting syntax errors and illegal function calls. Even more importantly, experience with the demo programs will help you select and apply the best tool for each programming challenge.

To get started, I suggest you begin with Chapter 1. If you are beyond a beginning level, you'll be able to skim the first chapters rather rapidly, but keep in mind that the information builds from one chapter to the next.

Once you are familiar with the information covered in this book, my hope is that you'll use it much as I do. I use it daily as my own programming handbook, and I often refer back to the sections I need when I encounter a particular programming requirement or problem. For my finished programs, it serves as documentation of the techniques I've used. These techniques have made my programs faster and better. I hope you'll profit from them too!

Lewis Rosenfelder
July, 1983

Getting Familiar with Disks and Your DOS

Why Disks?

Disk, diskette, mini disk, floppy disk, flexible disk — these are all names for a flat, circular piece of flexible mylar plastic, which is coated with a magnetically sensitive material, and can be rotated within a thin plastic jacket by a device called a disk drive. The diskette, as we shall call it, is the most popular and practical solution to the problem of storing and retrieving information for use with microcomputers.

The "problem of storing information" can best be understood by thinking of the TRS-80 microcomputer in its simplest and least expensive form. For a few hundred dollars you can get a typewriter-like keyboard and a TV-like video display. Within the case is a microprocessor (or "CPU" — Central Processing Unit) that has the brains to do certain fundamental arithmetic and logical operations.

Also within the case, we get some amount of memory (storage) capacity, ranging up to 65,536 characters of information. That memory can store numbers, letters and codes that specify logical procedures or "programs."

Some of the memory, (14336 characters of it on the TRS-80 Model III) is Read Only Memory, or ROM. The ROM area of memory has been permanently programmed — it cannot be erased. On Models I and III the ROM contains a program called the BASIC language interpreter. Its job is to translate the English-like and mathematical commands we give it into the simple operations that the microprocessor can perform and to translate the results back for us into usable formats. On the TRS-80 Model II the ROM is much smaller. The logic within its ROM is devoted to startup procedures that give it enough intelligence to load additional programs from a diskette.

The remaining memory in your TRS-80 computer is called RAM, or Random Access Memory. The microprocessor can address any of the locations within RAM, where it may store, recall or act upon the coded information that the memory holds. The important point about the RAM in your computer is that it is erasable. When you turn off the computer, everything in RAM is forgotten. When you type-in, or load, a program or add new data, the previous contents of RAM at the addresses affected are replaced by the new data.

As I write this chapter, I am typing the text into my TRS-80 computer. Some of the RAM is storing a word-processing program that tells the computer how to accept the characters I type, and how to do operations such as word and paragraph insertions and deletions. The remaining portion of RAM can temporarily store about a dozen pages of text at a time. Later, I may wish to work on a different chapter, or I may want to perform an entirely different function, such as an accounting operation.

Diskette storage provides a relatively inexpensive method of storing larger volumes of data and programs than can be accommodated in RAM at any one time. When I'm finished typing several pages of text, I can record or "save" what I've typed onto a diskette. The diskette can store many dozens of pages, so, whenever I wish, I can "load" any section of the text that I've previously saved, and I can resume work on it. Diskettes are removable and are easily stored in a small amount of space.

The data available to my computer, and the capacity to record new data is limited only by the number of diskettes I have and the capacity of each. Diskettes are easily duplicated, making it easy to have extra copies for safe-keeping, and practical for companies to sell programs they've written to thousands of TRS-80 computer users. The magnetic surface on a diskette will retain stored information for years. Also the medium is reusable when you intentionally erase it or when you elect to record new information on it.

A Useful Analogy

The most common analogy used in describing a diskette is to compare it to a phonograph record. With the goal of better understanding data storage on diskettes and some of the terminology used to describe it, let's look at the comparison in more detail.

A phonograph record is a medium for the storage of a type of information — music. The information is "reads" by a needle moving through the grooves on the record as it rotates on a turntable. The data is then sent through an amplifier and eventually out the speakers so we can hear it.

A diskette is a medium for the storage of computer information. A disk drive is the device that "reads" the diskette. On a computer diskette, magnetic patterns are read by the disk drive and sent to the computer's memory, where the computer may do such things as process it, display it, or print it on paper.

A phonograph record is pre-recorded at the factory. It may not be re-recorded. The turntable and needle act as an "input" device. The speakers are the "output" devices.

Since the surface material on a diskette is magnetically sensitive, like that of audio tapes and cassettes, a disk drive can record or "write" information onto a diskette. And similarly, diskettes can be magnetically erased, or sections can be re-recorded. Because of this capability, the disk drive can act as both an input device and an output device.

We often talk of "disk I/O," meaning disk input and output. As an input device, a disk drive reads data from the diskette so that it can be copied into computer memory. As an output device, a disk drive magnetically writes data onto the diskette as it is copied from computer memory.

The Benefits of Random Access

You can pick up the tone-arm on a phonograph and move it to any song on the record. This provides an advantage over audio cassettes, where it may be necessary to play (or fast-forward) through unwanted songs to get to the one you want.

The mechanism within the disk drive can quickly move the read-write head to any spot on the radius of the diskette. This is all done according to instructions given to it by the computer. This capability is called random access. In the TRS-80 Model II manuals they use a different, but perhaps more precise, term for the same thing — direct access.

Tape cassetes can also be used with computers for data storage. They, in contrast to diskettes, are serial or sequential devices, in that they are read, or written, from beginning to end without the speed or flexibility of random access.

Tracks, Sectors, Bits, and Bytes

From the point where it is first set down, the stylus follows the grooves on a phonograph record in an inward spiral as it rotates at 33 1/3 or 45 rotations per minute. You can read the label at the center of the record to see what songs are on it and you can count across the recorded bands to aid yourself in setting the needle at the beginning of the song you want to hear.

There are no physical grooves on a diskette. Instead, magnetic patterns are used to subdivide it. Before your computer can use a blank diskette for the storage of data, these patterns must be recorded using a process called formatting. With a special command (FORMAT) you can tell your computer to format a diskette. In some cases, the formatting process is executed automatically, as is done when you tell the computer to duplicate the contents of one diskette onto another with the BACKUP command.

In contrast to phonograph recording methods, the data on diskettes is organized in concentric circles rather than in a continuous spiral. These concentric circles are called tracks. The disk drive mechanism can "step" the read-write head in or out to any track. Radio Shack's Model I computer was most often sold with 35-track drives. The Model III is sold with 40-track drives. Some TRS-80 users elect to buy drives from other manufacturers that have 77 or 80 tracks.

On each track, the formatting process makes magnetic marks to further subdivide the diskette into sectors. The data that is useful to the computer, and ultimately to us, is recorded within these sectors. Each sector contains 2048 magnetic indicators called "bits," which (according to how they've been recorded or re-recorded) are interpreted as ones and zeros.

These ones and zeros are used by the computer's processor in groupings of 8 bits. A combination of 8 bits is called a "byte." Earlier we spoke of memory capacities in terms of characters of information. We were really talking about bytes. The byte is the most commonly used measure of data storage today. So, as a programmer, you'll most often think of the capacity of a diskette sector as 256 bytes.

There are 256 possible patterns of one-bits and zero-bits within a byte. For example, 00000000 and 11111111 are two such patterns. In some cases, your computer is instructed to interpret or record these patterns according to standard conventions such as the one adopted as the American Standard Code for Information Exchange, or ASCII. The ASCII code for the letter "A," for example, is 65, represented by the bit pattern 01000001. The code for "B" is 66, represented by the bit pattern 01000010. The character codes used are usually listed in the instruction manuals that are provided with your computer.

In other cases, the 256 different codes may be assigned to efficiently represent special data that is unique to the computer system or program you are using. For example, within a BASIC program stored on a diskette in its non-ASCII compressed format, the bit pattern representing the number 132 is interpreted as the command "CLS", which instructs the computer to clear the screen.

A sequence of two or more contiguous bytes may represent data in compressed form. A byte containing a pattern representing 142 followed by one containing a pattern representing 12 might be interpreted as the integer 3214. To store "3214" in ASCII format would take 4 bytes, one byte for each character. To store it in a compressed or "binary" format requires only 2 bytes. In the programs you write and some of those that you buy, there will be other data compression schemes. They may be unique to a specific application. A payroll program might, for instance, record a 1 to indicate that a particular employee is single, or a 2 to indicate that the employee is married.

Diskette Storage Options

In our discussion of tracks and sectors we've seen that there are many ways information can be recorded on diskettes. In contrast to a phonograph record, where you can be relatively certain it will play on any make or model phonograph, incompatibility in diskette storage formats between different computer makes and models is the rule, rather than the exception.

We've said that the tracks on a diskette are divided into sectors. The question of how many sectors per track introduces another possible incompatibility. The standard TRS-80 Model I uses a "single-density" recording method, providing 10 sectors per track. On a 35-track diskette, you have 350 of these 256-byte sectors. The standard TRS-80 Model III uses a "double-density" recording method. In essence, 80% more data is squeezed onto a track so that there are 18 sectors per track. Thus, on a 40-track diskette you have 720 of the 256-byte sectors.

The recording density and some of the other particulars about the way data is stored on diskettes is managed by an electronic component within the expansion interface of the Model I and somewhere inside the case of the Model III computer. It is called the disk controller.

The disk controller is like a middle-man between your disk drives, the CPU and the memory of your computer. It accepts simple commands from the processor and tells the disk drive to perform them. The different disk controller models account for another compatibility consideration. Radio Shack and a few other manufacturers offer modification kits for the Model I called "doublers" that allow it to operate in double-density.

Still another variation in diskette storage is whether both sides of the diskette are used or not. On a phonograph record, you generally have music recorded on both sides, but you have to manually flip the record over. The standard, off-the-shelf disk drives for the Model I and III use single-sided recording.

Some manufacturers offer drives that have two read-write heads and are capable of recording on both sides of a diskette without your needing to remove and flip it over, effectively doubling the single-side capacity. As an example, you could configure a TRS-80 to use 40-track, double-sided, double-density disk drives.

Another variation among disk drives is the size of the diskettes that they use. The two popular sizes are 5-1/4 inch and 8-inch diameter. The TRS-80 Models I and III use 5-1/4 inch diskettes, while the Model II uses the 8-inch diskettes. The 8-inch diskettes have the same types of options and variations in recording densities and schemes. With a special disk controller (and attention to several other considerations that we will discuss) it is possible to use 8-inch diskettes and disk drives with a Model I or III for greater speed and storage capacity.

Disk Directories and Files

We noted that it is easy to locate songs on a phonograph record by looking at the label. On a diskette, we may write certain identifying information on the jacket with a felt-tipped pen, but the computer locates and keeps the information organized by reserving a few sectors to magnetically store a "directory." On a phonograph record we might have one long song, or perhaps several. On a diskette, we have "files." A single file may encompass a whole diskette, there may be just a few files, or there may be dozens of files on a diskette. Each file has a file name which is stored in the directory, along with notations as to which and how many sectors are being used to store the data in the file. When the computer is commanded to access a specific file, it searches the disk directory for the specified file name.

When you write a BASIC program, you give your computer a series of "how-to" instructions for performing a specific task. As you write the program, make modifications to it or use it, the BASIC program is stored in the memory of your computer.

However, the memory will "forget" everything you've "taught" it if you turn the computer off or load another program. So, for more permanent safe-keeping, you record the program on a diskette with BASIC's SAVE command. When you SAVE a BASIC program, the computer records the file name you specify in the disk directory and determines whether or not there is sufficient available space on the diskette. The computer then records the program instructions onto one or more disk sectors and notes where the information is stored.

A BASIC program file is just one of many types of files. As a disk user, you will probably also have data files. A data file may contain information such as customer names, account balances, inventory records, tables of numbers and statistics, or anything else imaginable.

If you are using Scripsit, Electric Pencil or another word processing program, you may have text files that store letters, contracts, research reports, or perhaps even poetry that you've written. As I mentioned earlier, this book is being typed into a TRS-80 computer. Each section of up to a dozen pages or so is stored as a text file on a diskette.

System files will also reside on your diskettes. The average computer user and programmer is hardly aware that they are there, but system files contain sophisticated programs and reference tables that provide much of the operating intelligence that your computer has. These programs contain much of the machine-language logic that organizes data on your diskettes and extend the capabilities of the built-in "Level 2" BASIC language to include the disk-related features and special functions of "Level 3" or "Disk Basic". On the Model II, since there isn't a built-in BASIC interpreter, the system files provide nearly all the operating logic.

Buffers and Error Detection

A phonograph has no memory, and its "intelligence" is limited to such things as bringing the volume, bass and treble to preset levels. As soon as the music "data" is input from the turntable, it is sent out through the speakers for us to hear. Disk storage works differently in several ways that we have not yet discussed.

With a disk drive, data is usually read one sector at a time. As the rotating surface of the diskette passes under the read-write head, the stream of 2048 bits or 256 bytes that are present in the selected sector are copied to an area of 256 bytes in the RAM memory of the computer. This memory area is called a buffer.

Once all 256 data bytes have been copied into the buffer, the microprocessor, under the direction of the currently active program, can use or manipulate the data in various ways.

When the computer is given an instruction to write data onto a diskette, the opposite happens. The data to be written must first be present in the memory buffer. Upon receipt of a write command, the data in the buffer is copied onto the specified diskette sector.

As you can see, a buffer acts as an intermediate storage area, just as an airport terminal might be considered an intermediate storage area for people who have just arrived or are awaiting an outgoing flight.

The airport analogy makes it convenient to bring up another important feature of disk storage. A departing pasenger, upon arriving at the airport, registers at the check-in counter. When the passenger boards a plane, the flight attendants make a count to verify that all intended travelers are present. When a sector of data is written to disk, the program in control may immediately re-read that sector, comparing it to the contents of the memory buffer to verify that the data has been recorded correctly. This "verify" option makes disk write operations slightly slower than they could be otherwise, but it can help prevent costly losses of data due to imperfections in the magnetic surface of the diskette or malfunctions in the equipment.

Disk systems also use methods to check the accuracy of data copied into the buffer during read operations. Suppose for a minute, in our airport example, the flight attendant recorded the age of each passenger boarding the plane, and after all passengers had boarded, she had added the ages together to get a total. Then, upon arrival at the plane's destination, while disembarking, each passenger had again reported his age. The "disembarking" total could be checked against the quantity computed when boarding. If the two totals differed, we would know that something was wrong (unless, of course, someone had a birthday during the flight).

In disk systems, the numerical value of each byte written to a sector is added as the data is recorded onto the diskette. The total is recorded at a special spot next to the sector data. Later, when the data is read back from the diskette, the system can compute a total and compare it to the one that was recorded the last time the sector was written.

Thus, our computer is able to warn us of fingerprint smudges, dirt, scratches or magnetic erasures that may have destroyed, or otherwise made invalid, some or all of the data on the diskette.

The error detection schemes used in disk I/O take different forms. Sometimes the numerical values of each byte are added, and the fact of whether an odd or even result was obtained is recorded and checked. This is called parity error checking.

Another method is CRC, or Cyclical Reduncy Checking. To the average computer user, an error message such as "Parity Error" or "CRC error" simply means that the computer has detected a problem. The problem may be on the diskette itself, or due to a malfunctioning disk drive or due to a temporary misalignment because of the way the diskette was inserted. Many times, one or more retries will result in a succesful read or write operation.

On a "dumb" phonograph, the "errors" are simply reproduced for us to hear in the form of annoying scratches, pops and hisses. On our intelligent computer disk system, even one erroneous bit can mean the difference between valid accounting totals, or word-processing text, and useless jibberish, or between working programs and those that won't run at all. Disk error detection is a vital safeguard, and we'll be devoting much attention to it.

Multiple Drive Systems

Quite often, and perhaps most often, computer disk systems are sold and used with more than one disk drive attached. It is common to configure a computer with 2, 3 or 4 disk drives. Any of the 4 drives may be accessed by the operator or a computer program by specifying its number — 0, 1, 2 or 3.

The most obvious reason for multiple drive systems is that each drive adds more "on-line" data storage. More information is instantly available to the computer without the need for the operator to remove and re-insert diskettes.

A second justification for having more than one drive is that it provides for very convenient duplication of diskettes. The operator may insert the diskette to be duplicated in one drive, and a blank diskette (or one whose data is no longer required) into another drive. Then, via the BACKUP command, the entire contents of the original or "source" diskette can be recorded onto the "destination" diskette. Backups can be made with only one drive, but the procedure is much more time-consuming and complicated for the operator. Daily backup procedures for protection of vital data are very important to the success of most computer installations.

Another benefit of multiple disk drive systems is the flexibility and efficiency that they provide in designing applications. In an inventory system, for example, a business may wish to control more items than will fit on one diskette. The inventory can be divided into departments. The programs that are required for the system might be stored on the diskette in drive 0. Then, depending on which department is to be processed, the proper diskette can be inserted in drive 1. When another department is to be processed, the diskettes in drive 1 can be swapped, while the drive 0 diskette remains on-line at all times.

The variations in diskette sizes, recording densities, capacities, and recording formats makes multiple drive systems important to program developers who must support users with differing setups. Conversion programs that copy information from a disk drive in one format to another drive in a different format satisfy this need.

Another case in which multiple drives are important is with the type of disk drive called "hard disk," "fixed disk" or "Winchester." This kind of high-speed, high-capacity storage system uses many of the same concepts that we've been discussing for diskettes, and the programming considerations are nearly identical. However, unlike diskettes, the recording surface on a Winchester disk is hermetically sealed. A removable media storage device, such as a floppy disk drive, is usually used in conjunction with the hard disk, so you can make backups and load programs from other computers into your computer.

The Disk Operating System

We've been discussing many of the "behind-the-scenes" considerations of disk I/O. The technical details of diskette formatting, directory handling, track and sector allocation, and error checking are quite complicated. The many variations in disk hardware compound the problem. A disk operating system, or DOS, is a collection of sophisticated programs that perform these functions and many others for us automatically. The result is that the job of disk programming is simplified a hundredfold, and the differences between various types of disk drives are made virtually invisible to us.

The disk operating system is stored in "system" files on the diskette in drive 0. The diskette these system files are stored on is called the "system diskette". Because the system files require a large amount of storage space, the available storage capacity on a system diskette is as much as 30 percent less than for non- system diskettes in drives 1, 2 or 3. As you perform various operations on your computer, the DOS automatically loads the system files required from the drive 0 diskette to a reserved area of memory, usually just above the ROM. Control is passed to these routines, the requested operation is quickly performed, then control is returned to the BASIC program being executed or to the operator at the keyboard, usually without your even being aware of what's happening.

In a certain way, the importance of the disk operating system is immediately apparent to any disk user. You can't begin to work with the computer after turning it on until you insert a system diskette in drive 0. Let's look at what happens.

When the computer is switched ON, a built-in program in ROM takes over immediately. Its only function is to access track 0, sector 0 of the diskette in drive 0. From that sector, instructions are obtained that tell the computer how to load another routine that gives the computer enough additional intelligence to load more system routines from the diskette, until, within a second or two, the computer is in a "DOS Ready" mode, from which any system function can be selected by simple keyboard commands. This process of automatic startup from a simple program that loads in progressively more complex programs is called "bootstrapping" or "booting." It is called into action when you turn the computer on, or when you press the reset button.

We've called a diskette containing the system files a "system" diskette. It may also be called a "DOS" diskette, or if we assume that the system files will be present, we may simply call it a "drive 0 diskette." There are many different disk operating systems. The computer manufacturer generally makes at least one available. Radio Shack's disk operating system is called "TRSDOS." Independent companies sell other DOSes that may have special capabilities or that are able to support the special models of disk drives and other peripherals that other vendors market.

NEWDOS, LDOS, DOS PLUS, ULTRADOS, VTOS, DBLDOS and CP/M, to mention a few, are some of the disk operating systems that can be purchased for use with Radio Shack's TRS-80 computers. You'll sometimes see a system diskette called by its name. For example, the initial message on the video display might read, "Please insert TRSDOS disk in drive 0."

The disk operating system routines are given to you on a diskette when you purchase your computer, or when you buy a DOS from another vendor. You may make backup copies of the original DOS diskette. To set up your computer for an application, you generally start by erasing unwanted files on a backup copy of the system diskette. Then, you record on that diskette the programs and data that you want to access on drive 0. Programs that you purchase from vendors other than the supplier of your DOS usually will be distributed on non-system diskettes to avoid copyright and royalty problems. If you wish to use them on drive 0, you will need to copy them onto a system diskette.

The main reason for having the DOS on a diskette is that if all of the system files were stored in memory at the same time, you would have precious little room left over for any programs. Being on diskette, small system files can be loaded to RAM as they are needed, usually overlaying the same memory area occupied by the routine that was used previously.

An equally important benefit is the flexibility that diskette storage of the DOS gives us. As improvements and corrections are made to the system routines, new versions of a DOS can be released. For the TRS-80 Model III, for example, Radio Shack released a TRSDOS version 1.1. A few months later, 1.2 was released, and soon after, TRSDOS 1.3 was distributed. We get the "state of the art" as the DOS continues to improve.

Disk operating systems for the TRS-80 also usually contain extensions and enhancements to the BASIC interpreter. For the Models I and III, these system routines add disk-related commands, and other new functions and features to the built-in Level 2 BASIC in ROM. On the TRS-80 Model II, the entire BASIC interpreter resides on the TRSDOS system diskette. If you're willing to pay for them, the system diskettes provided by other vendors add even more "bells and whistles" to the BASIC interpreter. The BASIC interpreter executes the BASIC language. The interpreter itself is a program that sends commands to the disk operating system.

When in DOS Ready mode, your contact with the disk operating system is through the command interpreter. Its function is to accept simple commands from the keyboard, known as "library" and "utility" commands. Each library and utility command loads a routine that performs a specific function. For instance, you can type "DIR" to display a directory of the files that reside on drive 0. You can type "BACKUP" to call a program that duplicates one disk to another.

In the following pages, I'll be guiding you through some hands-on examples that will really help you get acquainted with your DOS. Our "tour" of the disk operating system commands will cover everything you need to know as a TRS-80 user or programmer. I suggest that you have your disk operating system owner's manual at your side. It will expand on the details of each command we discuss and show you other, but non-essential, features that are available.

We will be splitting our attention between the TRS-80 Models I and III. Please read the discussions on both because, in many cases, a feature or explanation about one system will help you to understand something about the other.

We will center our discussions around TRSDOS, the disk operating system provided when you purchase a Radio Shack disk system. If you've purchased a different DOS, you'll probably find that most, or all of the commands work the same. In fact, many of the vendors of disk operating systems assume you already have TRSDOS and its owner's manual. Their manuals simply explain the differences.

Powering Up Your System

We'll start our "hands-on" discussion by assuming that you are sitting in front of your computer. First you must be sure that all modules are plugged in and properly interconnected, and all switches and buttons are in their ON position. You might refer to the owners manual for further details.

There are several things you must do to get a TRS-80 Model I ready for use. The checklist below will help if you are running the computer for the first time or training someone else to run it. Some day, if you encounter trouble getting your Model I started, a careful review of the list may help you solve the problem.

- The video display should be plugged in. Turn it ON by pressing the button on the front of the unit, just to the right of the screen. A cable should connect it to the video socket at the right rear of the keyboard unit.

- The expansion interface unit gets its power from a small black power supply. A cable connects this power supply to a socket in the compartment under the upper right cover of the unit. The expansion interface's power supply, along with an identical power supply for the keyboard, may be installed in the compartment. When the power supplies are installed in the expansion interface, you'll see two power plugs exiting from the rear and a cable that goes to the POWER socket on the right rear of the keyboard.

On the other hand, since the power supplies can get rather warm, many users have found that it is a good idea to leave them outside of the expansion interface box. The cable from one supply goes to the socket on the inside of the interface. The cable from the other goes to the socket on the rear of the keyboard.

Once power is properly connected to the expansion interface, turn it ON with the button on the front of the unit. Make sure that this button properly "clicks" to its "ON" position.

A short flat ribbon cable, (or on some units, a "buffered EI cable") connects the left rear of the keyboard to the left front of the expansion interface.

- Each disk drive unit has its own power plug and ON/OFF toggle switch. The drives are connected to the socket at the rear of the expansion interface with a ribbon cable. The ribbon cable has connectors for up to 4 disk drive units. If you have fewer than 4 drives, you must use the connectors on the cable that are closest to the interface. Any unused connectors will hang free at the end of the cable.

The Radio Shack "mini disk" drives are sold with two different catalog numbers. You should have one "26-1160" drive. It contains a termination resistor that tells the computer that it is the last (or only) drive on the cable. If you have more than one drive, the others should be the "26-1161" type. They should be connected to the sockets on the cable between your "26-1160" drive and the expansion interface.

All disk drives must be connected properly, receiving power, and turned ON for the system to work properly. After you've turned them ON, the indicator lights on each should be off and the motors in each should be stopped. If one or more of the motors continues to run, or one or more of the indicator lights remain on, your drives are connected improperly and you should correct the condition before you proceed.

- Many users have found that, especially with systems having 3 or 4 drives, it is a good idea to mount a small fan that blows air toward the rear of the drives. This can improve performance by preventing the drives from getting too hot.

- The keyboard should be getting power from its POWER socket, via its own black power supply. When you press the ON button at the right rear of the keyboard, the red light just to the right of the typewriter pad should glow.

If you've got a TRS-80 Model III, turning the unit ON is much simpler. You just plug it in and press the rocker switch under the right side of the keyboard section. If you've got one or two external drives, these should be connected to a socket on the bottom of the case with a ribbon cable that has a connector for each drive. For proper operation be sure, if you've got external drives, that they are properly connected, receiving power, and turned ON, whether or not you intend to use them during the current session.

Now, since you have not yet inserted a diskette, your computer can go no further until you do so. Your Model I most likely is displaying a jumbled pattern of characters and graphics blocks. Your Model III, which is a little friendlier, is patiently displaying the message "Diskette?"

DOS READY — Booting Your System

"DOS READY," "TRSDOS READY" or a similar message on your video display indicates that your disk-drive equipped computer is ready to accept and interpret your commands. This message tells you that your disk operating system is now in control. The DOS performs the functions required to manage the use of the random access memory, the disk drives, and the other input/output devices — most commonly, the keyboard, video display and line printer.

You've probably taken these functions for granted, but actually they are quite complex. Just as each of the controls on your automobile's instrument panel may be connected to dozens of gears and wires, each of the capabilities of your computer is executed by complex routines that are "invisible" to you. "DOS READY" is the starting point for all of the operations that we will be performing.

To "raise yourself by your own bootstraps" means to achieve something significant after having started with almost nothing. The computer world has borrowed the idea of "bootstrapping" in terminology, and by analogy, to describe the way a computer starts itself with very simple and limited logic — and progresses to very complex and flexible programs. When you "boot" your system, you cause it to begin execution of a permanently built-in machine-language program (in ROM — Read Only Memory). This program loads whatever short machine-language program is recorded on sector 0, track 0, of the diskette in drive 0. As soon as that program is copied into an area of RAM (Random Access Memory) it is executed. The short "bootstrap" program does little more than to tell the computer how to load the other central modules of the disk operating system. As these are loaded and executed, your computer takes on a personality and intelligence that is determined by the sophistication of the DOS you are using.

To begin operations on your computer with disks you must insert a "system" diskette in drive 0. On the Model I, drive 0 is the disk drive that is connected to the socket on the disk drive cable that is nearest to the expansion interface.

On the Model III, the bottom drive in the cabinet is drive 0. Again, a system disk is one on which your disk operating system has been recorded. It will most likely have been labeled with the name of the disk operating system you are using. We'll assume for now that you are using Radio Shack's system, TRSDOS, but most other disk operating systems operate similarly.

On the Model I, once you've inserted your system diskette and closed the drive door, press the reset button at the left rear of the keyboard. The drive 0 light will glow for a few seconds, the motor will whir, and your display will read:

```
TRSDOS - DISK OPERATING SYSTEM - VER 2.3

DOS READY
```

When the Model III is displaying the "Diskette?" prompt, the booting process is automatic as soon as you insert your system diskette in drive 0 and close the door. (If the "Diskette?" prompt is not showing on the screen, simply press the recessed orange reset button on the right side of the keyboard.) When the DOS is booted, the video display shows a neat little graphics picture of the Model III, an identification of the disk operating system and version number you are using, and Tandy's copyright message. Then it requires that you type the two-digit month, day and year, separated by slashes and terminated by a press of the <ENTER> key. Next it asks you for the time, expressed in two digit hours, minutes and seconds, separated by colons. You can type the time and press <ENTER>, or you can simply press <ENTER>, if like me, you think that typing it is a real headache and not of any real value to the programs you will be using. Finally, the Model III displays the message:

```
TRSDOS READY
..................................................
```

Before we go on, let's look at a few opportunities for problems that you may have encountered.

If your Model I just sits there staring at you with that jumbled pattern on the display, and you've done everything else right so far, it's likely that you've inserted a blank diskette, you've inserted the diskette upside down or backwards, or the diskette has something recorded on it that is completely foreign to the equipment you have. The same applies to your Model III if it just keeps displaying the "Diskette?" message. In either case, check the system disk you've inserted, or try a different one if necessary, and press the reset button again.

If your computer displays a message such as "No System" or "Not a System Disk", this means that the diskette you're inserted has been previously formatted magnetically for use on your computer in a drive other than zero, but that it does not have the disk operating system recorded onto it. You should remove it, find a disk that does have a DOS on it, insert that one, and press the reset button to try again.

If your computer displays "Disk Error", it means that the disk you inserted was previously formatted and that it may indeed have a disk operating system recorded on it, but the drive is having trouble reading its track 0, sector 0.

Remove the diskette, carefully re-insert it, and try resetting again. If a few attempts don't solve the problem, try a different diskette. Repeated unsuccesful attempts may convince you that your drive 0 is malfunctioning, but before you send it in for repair, be sure that all cable connections are good and tight. It may help to clean the contacts with TV tuner spray and a brush or piece of cloth. It may also help to unplug and reconnect all cables (with power OFF).

If your computer starts up fine, momentarily displays the DOS READY or TRSDOS READY message, and then keeps on going by displaying other messages or starting into a program, the AUTO function on your diskette is active. We'll talk more about AUTO later. For now, you want the DOS READY message, so press reset again, this time holding down the ENTER key until the message is displayed and the drive 0 motor and light go off. (The ENTER key disables auto startups, so to get to DOS READY, I've gotten into the habit of always holding it down when I press reset because I often use the AUTO function for application programs.)

Using and Understanding the DOS Commands

The Library Commands

Now that you're in DOS READY mode, you can take a look at the commands that are available. To execute a command, simply type it, followed by the optional or required parameters, and press <ENTER>. Begin by typing the "LIB" command. LIB displays the library of DOS commands that are available. On the Model I with TRSDOS 2.3, your display shows:

```
LIB
APPEND      ATTRIB      AUTO        CLOCK
COPY        DATE        DEVICE      DIR
DUMP        FREE        KILL        LIB
LIST        LOAD        PRINT       PROT
RENAME      TIME        VERIFY
```

On the Model III with TRSDOS 1.3, your display shows:

```
LIB
APPEND  ATTRIB  AUTO    BACKUP  BUILD   CLEAR   CLOCK   CLS
COPY    CREATE  DATE    DEBUG   DIR     DO      DUAL    DUMP
ERROR   FORMS   FORMAT  FREE    HELP    KILL    LIB     LIST
LOAD    MASTER  PATCH   PAUSE   PROT    PURGE   RELO    RENAME
ROUTE   SETCOM  TAPE    TIME    WP
```

Your *Disk System Owner's Manual* gives you exact details on how to use each of the library commands. In practice, you'll probably find that there are several that you tend to use often. We'll concentrate on these for now.

Displaying a Directory

To a diskette, the directory is like a card catalog is to a public library. Among other things, it contains the names of all files that are currently stored on the diskette, the length of each file, and protection information about the conditions under which you are authorized to access the file.

The DIR library command prints the directory of any diskette on your video display.

For our first example, we will use a TRS-80 Model I to display the directory of a TRSDOS 2.3 diskette currently inserted in drive 0. The last message displayed on your screen should be "DOS READY". Now, simply enter the command "DIR" and then press the ENTER key — which must be pressed after all commands to DOS. Here's how the directory looks, assuming the disk hasn't been altered since it was purchased from Radio Shack — no files added or erased:

```
FILE DIRECTORY ---- DRIVE Ø    TRSDOS  --- 12/1Ø/8Ø

GETDISK/BAS          TEST2/BAS          DISKDUMP/BAS
GETTAPE/BAS          TAPEDISK/CMD       TEST1A/CMD

DOS READY
```

What you see is a list of 6 file names. These happen to be names of programs. In other cases, a directory might show file names that represent groupings of data items, such as names, addresses and accounting balances. Each file name was specified by a programmer or user, within the limitations defined in the *Disk Operating System Owner's Manual.* For convenience in identifying the purpose of each file, the names used most often are short abreviations.

Though not required for proper execution, the "/BAS" extension on a filename says that the file contains a BASIC language program. The "/CMD" extension tells you, and the disk operating system, that a file contains machine-language code. A "command" file is executed by typing in the file name while in DOS READY mode.

You can get more information about the files on a diskette by using a variation of the DIR command. "DIR :0(A)" tells the system to display information about every file on drive 0, in a columnar format:

```
FILE DIRECTORY ---- DRIVE Ø    TRSDOS  --- 1Ø/1Ø/8Ø

GETDISK/BAS          LRL= 256 / EOF=  7 / SIZE=   2 GRANS
TEST2/BAS            LRL= 256 / EOF= 57 / SIZE=  12 GRANS
DISKDUMP/BAS         LRL= 256 / EOF=  3 / SIZE=   1 GRANS
GETTAPE/BAS          LRL= 256 / EOF=  5 / SIZE=   2 GRANS
TAPEDISK/CMD         LRL= 256 / EOF=  2 / SIZE=   1 GRANS
TEST1A/CMD           LRL= 256 / EOF=  5 / SIZE=   2 GRANS
```

The LRL column tells us the logical record length being used in each file. As we said earlier, the computer reads and writes files one sector at a time. Since these are program files, the computer will be copying them, beginning to end, from disk into memory when a load command is received. To load them most quickly, the logical record length is automatically set to to the maximum, 256 bytes. Some versions of TRSDOS create data files having logical record lengths shorter than 256 bytes where the information groupings make it more efficient to do so.

EOF stands for "end of file." Thus, the EOF column tells how many logical records the file contains. The program, GETDISK/BAS contains 7 records of 256 bytes each.

TRSDOS 2.3 on the Model I allocates disk file space on a diskette in multiples of five 256-byte sectors at a time. Since there may be more than one file on a diskette, it is usually not practical for the disk operating system to organize each file as a contiguous block of data. It has to allow for the possibility that multiple files will be created, extended, and/or killed (deleted) in a random fashion that cannot be pre-determined. In practice, files often "leap-frog" across each other on the disk. By allocating space five sectors at a time, the operating system strikes a compromise between the access speed that could be obtained with a contiguous organization and the efficiency that would be obtained if files were allocated on a sector-by-sector basis.

Since a track on a single-density Model I diskette contains 10 sectors, using two 5-sector groupings per track helps to keep the file management scheme from getting too complex. These 5-sector units are called granules or "grans". The grans column on the directory display shows us how many grans have been allocated to each file.

In the example, the TAPEDISK/CMD file is only 2 sectors long, but the minimum allocation of 1 gran, or 5 sectors, has been assigned to it. This means that 3 sectors are unused. It also means the if you rewrote the program, you could make it at least 768 bytes longer without reducing the free disk space available to other files.

We said earlier that a system diskette contains the files that comprise the disk operating system. Since most programmers will rarely, if ever, work with the system files, they are not shown in the directory display unless you request them. To display them you add the "S" parameter to the DIR command. The syntax you use to get a detailed directory of drive 0 that includes the system files is "DIR :0(A,S)". In addition to the "user" files shown in our sample directory, the following files will be shown:

```
BOOT/SYS SIP     LRL= 256 / EOF=  5 / SIZE=   1 GRANS
DIR/SYS SIP      LRL= 256 / EOF= 1Ø / SIZE=   2 GRANS
SYSØ/SYS SIP     LRL= 256 / EOF= 15 / SIZE=   3 GRANS
SYS1/SYS SIP     LRL= 256 / EOF=  5 / SIZE=   1 GRANS
SYS2/SYS SIP     LRL= 256 / EOF=  5 / SIZE=   1 GRANS
SYS3/SYS SIP     LRL= 256 / EOF=  5 / SIZE=   1 GRANS
SYS4/SYS SIP     LRL= 256 / EOF=  5 / SIZE=   1 GRANS
SYS5/SYS SIP     LRL= 256 / EOF=  5 / SIZE=   1 GRANS
SYS6/SYS SIP     LRL= 256 / EOF= 15 / SIZE=   3 GRANS
```

BOOT/SYS and DIR/SYS are two system files that are recorded on every formatted diskette. BOOT/SYS always starts on sector 0 of track 0. It contains the startup bootstrap logic we talked of earlier. DIR/SYS is a file that contains the directory of all files on the diskette. It contains all the file names, attributes, lengths, and their track and sector locations. SYSØ/SYS through SYS6/SYS are files containing the routines that make up the disk operating system. Rather than being organized as one big file, the disk operating system is designed to be modular. Depending on the function that you or one of your programs requests, the proper module is loaded and executed. When another operation is requested, a different system program may be loaded into the same area of memory so that it overlays the previous one used. This modular approach minimizes the amount of memory that is used, so that you have as much memory space as possible for your own use.

The "SIP" you see following each file name is a list of the special attributes for each of these files. The "S" code means that it is a system file. The "I" indicates that it is an "invisible" file. The "I" attribute on a file prevents it from cluttering up your display when you request a directory. The "P" means that the file has a password assigned to it. Without the password, you are prevented from copying, deleting or modifying the file, unless you know how to over ride the password access system

There are still other files on the TRSDOS 2.3 diskette. So far, we've listed the user files having no special attributes, and those files having the "system" attribute. If you use the command "DIR :0(A,S,I)", all files on the diskette will be listed.

The new files listed are those that have the "invisible" attribute, but are not considered "system" files:

```
FORMAT/CMD  IP     LRL= 256 / EOF=  15 / SIZE=    3 GRANS
BACKUP/CMD  IP     LRL= 256 / EOF=  15 / SIZE=    3 GRANS
BASICR/CMD  IP     LRL= 256 / EOF=  23 / SIZE=    5 GRANS
BASIC/CMD  IP      LRL= 256 / EOF=  2Ø / SIZE=    4 GRANS
```

The new files we see are 4 invisible program files that are distributed with the TRSDOS 2.3 system diskette. The "/CMD" extension on each lets you execute any of them by simply typing the program name from DOS READY mode. As we shall see later, FORMAT prepares diskettes for use by formatting them, and BACKUP duplicates the contents of one diskette onto another. BASICR and BASIC are two versions of the interpreter program that let us use the BASIC programming language.

Before we go on to the directory commands for TRSDOS on the Model III, let's look at a non-system diskette on the Model I. If you have more than one drive, you can display a directory of the diskettes in drives 1, 2 or 3 with "DIR :1", "DIR :2", or "DIR :3" respectively. For a detailed directory that includes all invisible and system files on these drives, you can use "DIR :1(A,S,I)", "DIR :2(A,S,I)" or "DIR :3(A,S,I)". As an example, lets look at a diskette in drive 1 that is being used to store accounting data with Radio Shack's general ledger system. The command, "DIR :1" gives us:

```
FILE DIRECTORY ---- DRIVE 1    GLDATA1  ---   Ø5/15/82

GLMASTER            DETAIL

DOS READY
```

As you can see, there are two file names listed. GLMASTER is a data file that contains descriptions and balances for each of the asset, liability, income and expense accounts. DETAIL contains data for each accounting transaction processed during the current month.

You will also notice that the diskette's ID, "GLDATA1" is displayed at the top of the directory listing. You may assign an ID, of up to 8 characters, to each diskette at the time it is formatted. When used with the TRSDOS 2.3 BACKUP program, the diskette ID (or "pack ID," as it is sometimes called) provides a measure of protection when making duplicates. It is used to insure that a diskette may only be copied onto another diskette if that diskette is completeley unformatted and blank, or if it has been previously formatted with the same ID as the original.

If you go back to the previous illustration, you'll see that the diskette ID for the system disk in drive 0 is "TRSDOS". This name was assigned by Radio Shack at the time the system diskette was created. In general, the diskette ID for any system diskette will be the name of the disk operating system it contains, such as TRSDOS, LDOS, NEWDOS80, DOSPLUS or DBLDOS.

Now let's look at a detailed directory of the general ledger data diskette in drive 1. Our command is "DIR :1(A,S,I)". The display shows:

```
FILE DIRECTORY ---- DRIVE 1    GLDATA1  --- Ø5/15/82

BOOT/SYS SIP        LRL= 256 / EOF=   5 / SIZE=    1 GRANS
DIR/SYS SIP         LRL= 256 / EOF=  1Ø / SIZE=    2 GRANS
GLMASTER            LRL= 256 / EOF=  25 / SIZE=    6 GRANS
DETAIL              LRL= 256 / EOF=  1Ø / SIZE=    2 GRANS
```

Now we can see the length of GLMASTER and the DETAIL file. We can also see the two invisible system files on the diskette, BOOT/SYS and DIR/SYS.

BOOT/SYS is on all diskettes created by TRSDOS and most other TRS-80 disk operating systems. Though you can't tell it by looking at a directory display, the program and data that it contains always resides on track 0 of the diskette. On a non-system diskette it has two main functions.

First, it has program logic to display the message "Not a System Diskette" or "Disk Error" if you make an attempt to boot your computer with it in drive 0. The other important function is to tell the operating system which track is being used to store the directory on this particular diskette. The directory occupies one track, or 2 grans, and it is stored on all diskettes created by TRSDOS and similar operating systems in the file named "DIR/SYS".

Model III TRSDOS Directories

The directory format for the Model III with TRSDOS 1.3 is slightly different from that for the Model I with TRSDOS 2.3. The differences are due to the fact that Model III TRSDOS is a more advanced disk operating system. Very few of the changes were dictated because of differences in the hardware.

To illustrate the directory, we'll look at a copy of the TRSDOS 1.3 system diskette as distributed by Radio Shack. When the "TRSDOS Ready" prompt shows on your video display, use the "DIR :0" command. The display is:

```
Disk Name: TRSDOS     Drive: Ø         Ø4/Ø2/82
Filename       Attrb  LRL  #Rec  #Grn  #Ext  EOF  Date
LPC/CMD        N*XØ   256    1     1     1    Ø   Ø5/81
MEMTEST/CMD    N*XØ   256    8     3     1    Ø   Ø5/81
HERZ5Ø/BLD     N*XØ   256    2     1     1   4Ø   Ø5/81
***  166 Free Granules ***
TRSDOS Ready
```

You see 3 files listed in the directory. LPC/CMD and MEMTEST/CMD are machine-language programs that may be executed by simply typing LPC or MEMTEST while in TRSDOS Ready mode.

The "/CMD" extension means that these are "command files" which can be executed in this special way. (The purpose of each is explained in your "Disk System Owner's Manual").

The "/BLD" extension on the HERZ5Ø/BLD file tells us that it is a "DO-file". A DO-file is a file containing a list of commands that can be automatically executed by the computer just as if you were entering them into the keyboard, one-by-one. This one contains commands that modify the TRSDOS disk operating system files on your system diskette so it will work properly if you live in an area where the AC power is 50 rather than 60 Hz.

The "Attrb" column shows you the attributes assigned to each file. "N" is the "non-invisible" attribute. We'll be looking at the invisible files in a minute — they have an "I" attribute instead of the "N". The purpose of the "I" and "N" designations is to avoid cluttering up our directory display with files that are rarely of interest to the average user.

The "*" in the "Attrib" column means that these are user files. Generally, a user file is any file on the diskette that is not a system file. System files (those which contain the program logic and data of the disk operating system) are designated with an "S".

The "X0" attribute means that the files have no passwords assigned to them and that total access is available if you wish to delete, rename, read, write or otherwise modify them. In some cases, other protection attributes may replace the "X0". They can be assigned or changed with the ATTRIB library command. The "DIR" and "ATTRIB" sections of your owner's manual will tell you more.

The LRL column is exactly like the one we displayed for TRSDOS 2.3 on the Model I. It tells the logical record length (1 to 256 bytes) assigned for each file at the time it was created.

The next column (labeled "#Rec") is a count of the number of records in the file. Each has a length equal to that shown in the LRL column.

The "#Grn" column tells how many granules of disk storage have been assigned to each file. You can see a difference here from what we had in the Model I example. The Model III uses double-density storage. This gives us 18, rather than 10, 256-byte sectors per track. The designers chose to use 3 sectors per granule instead of 5, because 3 divides evenly into 18 and 5 doesn't. This size is quite efficient for most applications. To check it out, notice that LPC/CMD and HRZ5ØBLD both require 1 granule of storage space, since neither is greater than three 256-byte records in length. MEMTEST/CMD is 8 records long. We take 8 sectors, divide it by 3 sectors per granule, round down and add 1 to get our storage requirement of 3 granules.

The "#Ext" column indicates how many "extents" the file uses. This is another consideration in disk storage that shows you how the disk file is laid out on the diskette. An extent is a contiguous block of up to 32 granules. If a file is less than 32 granules long, but covers more than one extent, it has been recorded and continued at various non-contiguous locations on the disk. In general, the goal is files that are as compact as possible, to reduce the time required for the read-write head to move from one record to another.

The "EOF" column on the TRSDOS 1.3 directory has a different meaning from that for the EOF column with Model I TRSDOS 2.3. For certain types of files, the operating system maintains a notation of the last byte used in the last record of the file. In the HERZ5ØBLD file, for example, all 256 bytes of the first record, and only the first 40 bytes of the second record were used. If this file were extended, additional data could be added starting at the 41st byte of the second record.

The final column shows the date the file was created. The system uses date when starting up.

The last line of the directory shows how much space is available on the diskette. In this case, there are 166 free granules. If you multiply by 3, that's 498 sectors.

If you multiply 498 by 256, you will have the number of bytes (127,488) available for disk storage. In doing calculations like this, though, you must remember that usable disk space will be given to you in increments of 1 granule (that is, 3 sectors or 768 bytes).

The operating system owner's manual tells you that 3 optional parameters can be used with the DIR command. They are "INV" (to include invisible files in our display), "SYS" (to include system files) and "PRT" (to send the directory listing to our line printer for a hard-copy printout). Let's look at the system and invisible files on our TRSDOS 1.3 diskette. The command is "DIR :0(INV,SYS)". Three new files are added to your directory listing:

```
Filename      Attrb  LRL  #Rec  #Grn  #Ext  EOF  Date
BASIC/CMD      ISU6  256   2Ø     7     1    Ø   Ø5/81
CONVERT/CMD    ISU6  256   1Ø     4     1    Ø   Ø5/81
XFERSYS/CMD    ISU6  256    4     2     1    Ø   Ø5/81
```

Notice the attributes of the files displayed. Each has the "I" attribute, which makes it invisible on directory listings unless you specifically request otherwise, and the "S" attribute, which classifies it as a system file. BASIC/CMD is the BASIC language interpreter program. It extends the BASIC language, permanently stored in ROM, to include disk statements. CONVERT/CMD is a utility program that copies files from Model I format diskettes to Model III diskettes. XFERSYS/CMD is a utility program that transfers newly-released system files (those that comprise TRSDOS 1.3) onto older-version (TRSDOS 1.1 or 1.2) system diskettes.

Your owner's manual tells you that the "U6" attribute on each of these files lets you update them (to make correction "patches" if they should be required), but it prohibits you from "reading" them. They are designated "execute only."

It is interesting to observe that BASIC, CONVERT, and XFERSYS are considered system files on the Model III TRSDOS 1.3 diskette. You'll remember that BASIC is not considered a system file on the Model I. In the strictest sense of the definition, BASIC/CMD is not a system file. It is a program that operates with and uses the disk operating system, but it is not a part of the DOS itself and is not required for the disk operating system to function.

That brings up a question. Where are the "real" system files that correspond to "BOOT/SYS", "DIR/SYS", and "SYS0" through "SYS6" that you saw on the Model I TRSDOS 2.3 directory? You can be sure that they are there. You know that the disk drives on a Model III are capable of accessing 40 tracks. Each track contains 18 sectors or 6 grans, so by multiplication, a diskette has a capacity of 240 granules. By adding the numbers in the "#Grn" column, you see that 18 grans are being used for files. Add to this the 166 free granules and you get 184. Now, by subtracting 184 from the 240 granule capacity, you can surmise that the real system files are using 56 grans. It seems that, as far as the DIR command is concerned, Radio Shack has decided to make them truly invisible!

We'll discuss how to format a diskette later. For now, assume you have already formatted a diskette for use in drive 1. Here's how the directory will look:

```
Disk Name:  TESTDISK   Drive: 1        Ø7/Ø8/82
Filename     Attrb  LRL  #Rec  #Grn  #Ext  EOF  Date
***  233 Free Granules ***
TRSDOS Ready
```

When you look at the empty directory, you see that 233 free granules are available. From what you learned about Model I TRSDOS, you can assume that the directory file (DIR/SYS) will occupy 1 complete track (6 grans in this case), and the BOOT/SYS file will occupy 1 gran. Your guess is probably right. Adding 233, 6, and 1 gives 240 granules. You can conclude that the BOOT and DIR files on the system diskette in drive 0 have the same storage space requirements.

The FREE Command — Model III TRSDOS

We just made some speculations about system files on Model III TRSDOS diskettes. The FREE library command gives a graphic display of disk utilization in the form of a "free space map." The map it displays shows tracks 00 through 39, the granules that are currently allocated on each, and the location of the directory track. First look at the freshly-formatted diskette named "TESTDISK". In response to the "TRSDOS Ready" prompt, issuing the "FREE :1" command gives this map:

```
               Free Space Map
Trk #    TESTDISK-----------------   Drive: 1
ØØ-Ø4:    X..... : ...... : ...... : ...... : ......
Ø5-Ø9:    ...... : ...... : ...... : ...... : ......
1Ø-14:    ...... : ...... : ...... : ...... : ......
15-19:    ...... : ...... : Direct : ...... : ......
2Ø-24:    ...... : ...... : ...... : ...... : ......
25-29:    ...... : ...... : ...... : ...... : ......
3Ø-34:    ...... : ...... : ...... : ...... : ......
35-39:    ...... : ...... : ...... : ...... : ......
TRSDOS Ready
.........................................................
```

The dots show all the granules that are currently unused. Notice that there are six dots per track for the 6 granules. You know that every formatted diskette has a BOOT file as the first granule of track 0. Sure enough, an "X" marks its location.

The word "Direct" means that directory space has been reserved on track 17. Why track 17? My best guess is that it's because 17 is the middle track for a 35-track drive. Model I TRS-80's were originally sold with 35-track drives, and perhaps this is a carryover. Since the directory must be accessed often, track 17 gives the shortest average read-write head movement over 35 tracks, and thus, the greatest speed.

FREE :0 shows the free space map for the drive 0 TRSDOS system diskette, having 166 free granules:

```
               Free Space Map
Trk #    TRSDOS -----------------   Drive: Ø
ØØ-Ø4:    XXXXXX : XXXXXX : XXXXXX : XXXXXX : XXXXXX
Ø5-Ø9:    XXXX.. : ...... : ...... : ...... : ......
1Ø-14:    ...... : ...... : ...... : ...... : ......
15-19:    XXXXXX : XXXXXX : Direct : XXXXXX : XXXXXX
2Ø-24:    XXXXXX : XXXX.. : ...... : ...... : ......
25-29:    ...... : ...... : ...... : ...... : ......
3Ø-34:    ...... : ...... : ...... : ...... : ......
35-39:    ...... : ...... : ...... : ...... : ......
TRSDOS Ready
.........................................................
```

You can see that the directory is in the same location (track 17). Most of the system files occupy the first 6 tracks. Others, along with the user files currently active, occupy tracks 15 through 21.

One of the primary functions of a disk operating system is "file management." The system routines automatically decide how all new files will be laid out on the diskette. As users of TRSDOS or other similar disk operating systems, we are generally unaware of which tracks and sectors are being used by any particular file, and rarely need to be concerned about it. In typical disk storage usage, you will be continually adding new files, extending them, and deleting old ones — creating some rather complex problems for the disk operating system to handle. It must act like a super hotel reservations clerk, besieged with constant check-ins, check-outs and requests for "adjoining rooms." The convenience afforded by DOS's freeing your mind from file management concerns is really quite a blessing!

The FREE command with Model III TRSDOS serves basically as a curiosity. When you want to know how many granules are free on any diskette, the easiest way is to simply display a directory. A free space map does come in handy at times though. I use it in rare situations (with certain non-TRSDOS operating systems) when I am using data diskettes formatted for 40 tracks on 35-track drives and I want to see if tracks 35 through 39 are in use.

The FREE Command — Model I TRSDOS

The FREE library command operates differently with TRSDOS 2.3 on the TRS-80 Model I. It correctly assumes that your main interest is not how the files are laid out on the diskette, but rather, how many free granules of storage are available.

This command is entered by simply typing "FREE" while in DOS READY mode. No drive specification is used. Starting with drive 0, each drive light will come on briefly as the system reads the diskette present in each. The display shows the diskette ID assigned to each, the date each diskette was created, the number of unused spaces remaining for file names in the directory, and how many free granules are available. A 4-drive accounts-receivable package that stores data on drives 0, 1, and 2, and programs on drive 3 might give the following display when all diskettes are inserted and the free command is entered:

```
DOS READY
FREE
DRIVE Ø - TRSDOS    Ø5/21/82   36 FILES,    4 GRANS
DRIVE 1 - AR1       Ø5/21/82   42 FILES,    7 GRANS
DRIVE 2 - AR2       Ø5/21/82   44 FILES,    5 GRANS
DRIVE 3 - ARPROG    Ø4/18/82   4Ø FILES,    3 GRANS

DOS READY
```

The FREE display should remind you that there are two concerns in disk capacity. The main one is how many grans remain. If the grans column shows 0 for any diskette, it is full, or at least, almost full. I say "almost full" because any file on the diskette may not be using all 5 sectors in the last granule assigned to it. You can check this by displaying a detailed directory for the diskette in question. When the count in the EOF column is an even multiple of 5, it will be necessary for the system to allocate a new granule should you try to extend the file. Otherwise, there are up to 4 sectors past the current end-of-file to play with.

By the way, a non-system diskette formatted on 35-track drives with TRSDOS on the Model I has 67 grans free when it is empty. You can judge the percentage of a diskette that is unused by dividing 67 into the number of grans free.

Another concern is the limitation on directory entries. The directory track must store the names for each file in use, along with information on the length for each, and the specific tracks and sectors allocated. Diskettes used with TRSDOS on the Model I have a capacity of 48 directory entries. You'll probably find that you rarely use over about 10 directory entries on a non-system (data) diskette. On many applications you might use only 1 or 2.

Directory capacity can be a real limitation if you are using high-capacity drives, such as an 8-inch disk drive or a hard disk. Fortunately, the disk operating systems used with these drives usually allow for larger directories. Some let you have 256 directory entries. Others use a design that automatically expands as required, or pre-allocates it according to your specifications.

Except on drive 0, the diskette ID's in the example are arbitrary. In the accounts receivable system shown, "AR1" and "AR2" were used as ID's to make it easy, when using FREE, to see if the correct disks were in the correct drives.

FREE is quite handy for quickly checking the ID of any diskette. Under Model I TRSDOS it is also a good way to verify that each drive is plugged in and working. If everything is OK, the lights on each drive will flash, one-by-one. If a drive is OFF or not connected, its light will not flash. If a diskette is inserted improperly or it has a magnetically damaged boot sector or directory track, or if it is unformatted (or formatted on a non-compatible system), the FREE command will halt the display when it gets to the bad disk, and an error message will be displayed. Generally, the error indicated (if any) will be a "GAT Read ERROR", meaning that the system has made an unsuccesful attempt to read the "granule allocation table" stored in the diskette's directory.

Command Error Messages — Model I TRSDOS

Before we continue this introduction to the disk operating system, let's look at some of the things that can go wrong. In this section we will talk about error conditions which are usually caused by improper commands entered through the keyboard. Unless you are somewhat familiar with what's going on, the error messages won't help much, other than to let you know that you made a mistake and had better try the command again. We will cover the Model I with TRSDOS 2.3 first, and then the Model III with TRSDOS 1.3. If you are using a different operating system, check your manual. The error conditions and messages are usually very similar.

On the Model I, get your display to show DOS READY. Now notice that if you simply press the ENTER key, the system responds with "What?". It is simply telling you that a valid command is required. It won't suffice to enter nothing!

Note that sometimes when you press the BREAK key in DOS Ready mode with TRSDOS 2.3, the screen will clear and the prompt "Memory Size?" will appear at the top of the display. This "Memory Size?" prompt may appear under other conditions when you don't want it — such as when the expansion interface isn't turned ON or there isn't a system diskette in drive 0. It tells us that, somehow, the computer has gone into its "Level 2" operating mode, and is performing as if no disk drives are attached.

If you can easily identify the source of the problem, simply correct it and press the reset button (while holding the ENTER key, if necessary) to get back to DOS Ready. If the source of the problem is not immediately obvious, press the ENTER key in answer to "Memory Size?". The computer will display:

```
RADIO SHACK LEVEL 2 BASIC
READY
>
```

Now type "?MEM" and press <ENTER>. The computer will show a number indicating how many bytes of memory are available. If it says 15572 or less, and your computer has more memory than that, it usually means that the memory in the expansion interface is not activated. Check to see that it is getting power, and that all connections are good. Sometimes it helps to click the ON/OFF button a few times, or to remove and reconnect the cable that connects to the keyboard. Then, try to boot your DOS again by pressing the reset button. Better yet, turn the power to the keyboard OFF and ON too. This completely clears out all memory.

Simple typing mistakes can also cause error conditions that might be confusing. Suppose, for example, that you typed "DRI" instead of "DIR" to request a directory. It would be great if the system simply told you, "You mispelled DIR — please type it again!" Instead, you get the "PROGRAM NOT FOUND" message. It assumed that you wanted to load and execute a program file named "DRI/CMD" from one of the diskettes currently present in the system. So it checked each drive's directory and failed to find it.

If your misspelling or erroneous key stroke starts with a number or blank, or contains punctuation that the system is not expecting, you will get an "ILLEGAL FILE NAME" error. Again, the computer is thinking that you are trying to type the name of a program to be executed.

A misspelled or invalid command can be even more confusing if you've got a bad or unformatted disk in one of the drives. The system looks for the erroneous program name, and then it "hangs up" when it gets to the bad disk (or drive), giving you a message such as "DIRECTORY READ ERROR" or "HIT READ ERROR". "DIRECTORY READ ERROR" means that it is unable to read the directory on a drive. "HIT READ ERROR" is more specific. It means that it found a directory, but its method of searching the directory for the file name (something called a "Hash Index Table") has gone afoul.

Another complication can occur if the system diskette is not in drive 0 at the time you enter your command or at the time the computer terminates execution of a command. When an error occurs, the disk operating system must access drive 0 to retrieve the proper error message for you.

If the system diskette is not present, the computer may just freeze or display a jumbled pattern, or "Memory Size?", "No System" or "Disk Error". Your response to any of these conditions should be to re-insert the system disk in drive 0, press reset and try again.

Error conditions will also occur if you enter invalid parameters. If you request a directory for drive 3, and only 2 drives are present, the system responds with "DEVICE NOT AVAILABLE". This message may also occur if the drive requested is disconnected, its door is open, or a diskette is not in it. In some cases, the error condition that the computer encounters will not have been anticipated in the design of the program or library command you're trying to execute. The only evidence of an error is that the system did not respond as you intended.

The key to understanding these and other error messages is to watch your display and the disk drive lights carefully when you enter a command. Think out the problem from the computer's point of view. Sometimes looking at your "Disk System Owner's Manual" will help. In the final analysis though, you must correct the problem the best way you know how, and try, try again!

Command Error Messages — Model III TRSDOS

Similar error conditions can occur on the Model III. There are some differences, though, in the way that they are displayed for you.

For example, if you press only the <ENTER> key in response to the TRSDOS Ready prompt, it won't say "What?". Rather, the TRSDOS Ready message will simply be repeated. If you press the BREAK key, it will just be ignored.

For the more serious errors, TRSDOS 1.3 on the Model III displays the error number. If you type "DRI" instead of "DIR", the Model III displays:

```
* * ERROR 24 * *
Operation Aborted
TRSDOS Ready
.....................................................
```

What's an ERROR 24? To find out, you could check your owner's manual; but the quick and easy way is to use the ERROR library command. After the DOS Ready prompt, type

"ERROR" and the number of the error you want explained. In this case type "ERROR 24" and press <ENTER>. The display shows:

```
ERROR 24
File Not Found
TRSDOS Ready
.......................................................
```

Other than this minor inconvenience, error-handling for the Model III is the same as for TRSDOS on the Model I, but the messages may be worded differently. "PROGRAM NOT FOUND" on the Model I is "File Not Found" on the Model III. "ILLEGAL FILE NAME" is "Invalid File Name". "DEVICE NOT AVAILABLE" is "Disk Drive Not in System".

Error 11 is "Illegal Command Parameter". It may occur if the command's punctuation is incorrect or a required part of the command is misspelled. For example, if you type "DIR :0(PTR)" to request a line printer listing for the directory of drive 0, you will get an error 11 which tells you that a parameter was used incorrectly. (In this case "PTR" should have been "PRT".)

When you are unsure of the proper command parameters, or you get an error 11 and can't figure out why, Model III TRSDOS provides a help function that will save you from leafing through your owner's manual. Simply type "HELP" and the name of the library command you want explained. To check this out for yourself, type "HELP DIR" and press <ENTER>. Your display will show:

```
TRSDOS Ready
HELP DIR

DIR [([SYS], [INV], [PRT])]

SYS=System files, INV=Invisible files
PRT=Output to Line Printer
```

The brackets show the parts of the command that are optional. Other codes and punctuations that explain the syntax can be found in your owner's manual. Generally, the HELP command display is complete enough for you to get past a problem, but sometimes not every option is shown or explained, so in some cases you'll need your manual. The "HELP DIR" display, for instance, fails to tell you that you can specify a drive number if you want a directory for a drive other than 0.

HELP FREE gives you another example:

```
TRSDOS Ready
HELP FREE

FREE [:D] [(PRT)]     (Display free space map)
PRT = Output to Line Printer
TRSDOS Ready
.......................................................
```

In addition to showing you that you can specify a drive number and optionally request output to a printer, it displays a short description of the purpose for the command.

Preparing Your Diskettes

Whether you plan to design and program your own system or you've purchased ready-made programs, the process of using diskettes on your computer begins here. In the following pages we will talk about the procedures needed to set up disk media for any computer operation. We will discuss formatting, making backups, killing files, copying files and using conversion programs.

The process of making a diskette backup is important for all users. It is a vital protection measure that insures your time investment after a computer run which may involve hours of entering new data onto diskettes. The "insurance" of making backups protects you from losing too much time in the event of an equipment malfunction, magnetic or physical damage to a diskette, or major human error that you or another operator of the system may make.

To backup a diskette is to make an exact duplicate of it on another diskette. The BACKUP program that is distributed on your system diskette performs this process, so, that after executing BACKUP, you will have two diskettes whose contents are identical.

For now, we will focus on the importance of the BACKUP program when setting up new diskettes for program development, or when implementing programs you've purchased.

The formatting process is important to users having more than one disk drive. Diskettes that are to be used in drives other than drive 0 must be subjected to this process before you can use them.

We've mentioned formatting a few times so far. In a nutshell, formatting is the process of magnetically preparing or initializing a diskette. Any previous data on the diskette is overwritten with magnetic patterns that divide it into sectors and tracks, an empty directory is placed on one of the tracks, and a boot program is recorded onto the first granule of storage on the diskette. As this process is executed, the system verifies that the information it is writing is recorded correctly by immediately reading it back. In this way, formatting verifies that a diskette and the drive being used are both working properly. Should areas of the diskette be unusable, due to a bad diskette or a malfunctioning drive, the bad tracks can be automatically locked out during the formatting process. A special area in the directory maintains a list of bad tracks (up to 5 of them) so the system can avoid them in future operations. In that case, the diskette would have less than its maximum capacity, but it would, nevertheless, be useable.

The formatting process is executed with the FORMAT command which is provided as part of the disk operating system. It also may be initiated automatically by the BACKUP command, since before a BACKUP may be made, the diskette to be written onto must be divided into sectors and tracks.

BACKUP duplicates the entire contents of a diskette onto another. COPY is a disk operating system command that duplicates the contents of an individual file. The original file being copied is unaltered. In most cases, this command is used to copy a file from one diskette to another.

When entering the specifications for the copy command, you may assign a different file name. So, while the new file has the same contents as the original, it is referenced by a different name in the directory. COPY may also be used on a single diskette if enough free capacity is present and a new file name is assigned.

In the process of setting up your system, COPY may be important to let you have the files you want on the diskettes you will be using. Because of copyright restrictions that prevent most program vendors from selling programs on a system diskette without paying royalties, many programs are distributed on non-system diskettes. If you want those programs to reside on drive 0, you need to use COPY to transfer them onto a backup copy of your system diskette. That way you have a single diskette that contains your disk operating system, as well as your new programs.

To KILL a file means, in effect, to delete or erase it from a diskette. On most disk operating systems, the KILL command, rather than erasing the data that the file contained, just removes the file's name from the directory and frees the storage granules it occupied so they can be used for other purposes.

The capability of killing files can be important on a daily or periodic basis. In the following pages, we'll examine the KILL command as it applies to the original setup of a system and the preparation of diskettes for use. Typically programs will be distributed on a system diskette that we won't be needing for the application that will be developing. We will sometimes use KILL to remove these unneeded program files from a backup copy of our system diskette to make more storage space available on drive 0. In other cases, you will KILL program and data files residing on non-system diskettes. If you have just a few files that are no longer necessary on a diskette, it is sometimes faster and easier to simply kill them, rather than re-formatting it.

If you write many programs, eventually you will build up a library of program files. Most new programs you write will be modifications of older programs. Here again, the COPY capability can be important. You'll be copying programs from one diskette to another, leaving the originals unaltered, and making modifications to the copies.

In some cases, conversion programs are also important to the process of setting up your diskettes for operation. They allow the copying of one or more files between diskettes having incompatible formats. The CONVERT utility on the Model III TRSDOS diskette is a good example. It reads programs and data files from a single-density 35-track Model I diskette and transfers them onto a Model III double-density 40-track diskette.

If you are using a disk operating system from a vendor other than Radio Shack, a conversion program may be provided for copying programs and data recorded in standard TRSDOS format onto diskettes formatted for use with the other operating system. That way, you can take advantage of capabilities provided by various hard disks, 8-inch diskettes, 80-track drives and double-sided drives. The specific procedures for conversions, as well as those for defining the mix of disk drive capacities and capabilities that you will be using are usually included with the documentation for those non-TRSDOS operating systems.

The flowchart titled "Preparing Diskettes for Program Development" (Figure 2.1) shows how to use BACKUP, FORMAT, KILL, COPY, and CONVERT when sitting down to begin programming a new system. The diamond symbols indicate the decisions you will be making. The rectangles show the processes you will be performing.

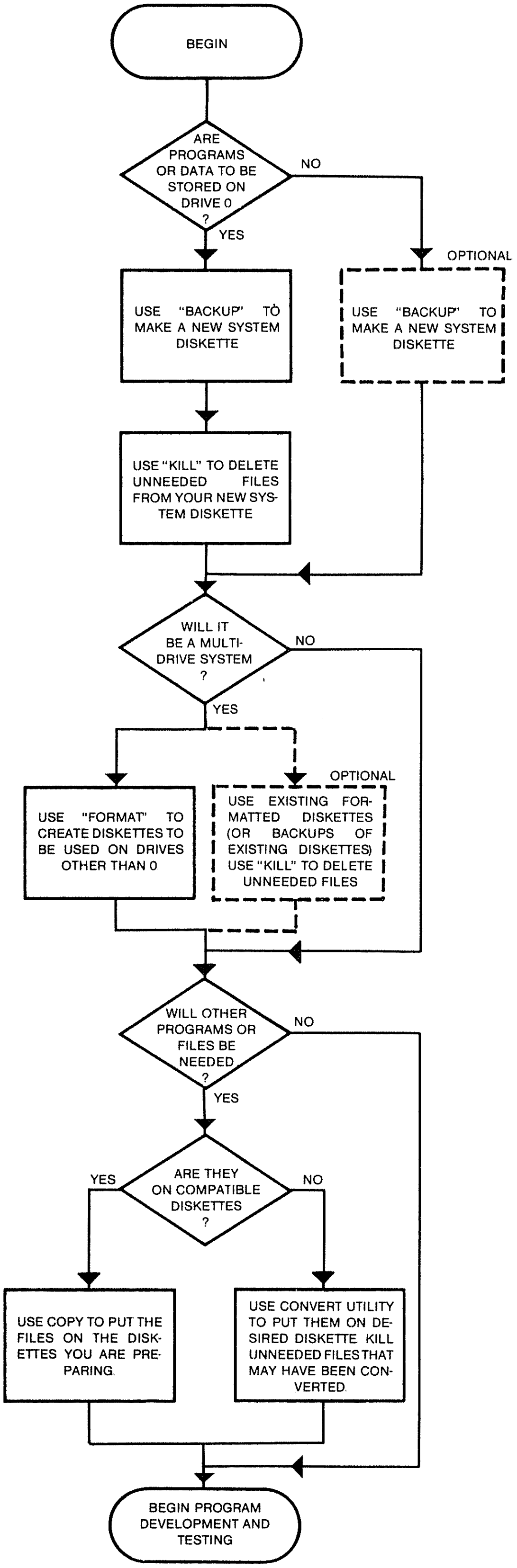
To test our flowchart, let's walk through the steps you'll need when we get to Chapter Four, where you'll need some disk space for learning the rudiments of sequential files.

All practice will be on drive 0 to accomodate those of you who have only one drive, so the answer to the first question on the chart is "yes." Programs or data are to be stored on drive 0.

Our chart says that you should use BACKUP to make a new system diskette. This gives you something to work with without being concerned about damaging your master system diskette and gives you the freedom to delete unneeded programs and files for more work space.

The next box reminds you that you can use KILL to delete unneeded programs and files. Generally, you'll want to display a directory with the invisible option at this point to see which files you won't need.

Figure 2.1 — *Preparing Diskettes for Program Development*


BEGIN
ARE PROGRAMS OR DATA TO BE STORED ON DRIVE 0 ?
NO
YES
OPTIONAL
USE "BACKUP" TO MAKE A NEW SYSTEM DISKETTE
USE "BACKUP" TO MAKE A NEW SYSTEM DISKETTE
USE "KILL" TO DELETE UNNEEDED FILES FROM YOUR NEW SYSTEM DISKETTE
WILL IT BE A MULTI-DRIVE SYSTEM ?
NO
YES
OPTIONAL
USE "FORMAT" TO CREATE DISKETTES TO BE USED ON DRIVES OTHER THAN 0
USE EXISTING FORMATTED DISKETTES (OR BACKUPS OF EXISTING DISKETTES) USE "KILL" TO DELETE UNNEEDED FILES
WILL OTHER PROGRAMS OR FILES BE NEEDED ?
NO
YES
ARE THEY ON COMPATIBLE DISKETTES ?
YES
NO
USE COPY TO PUT THE FILES ON THE DISKETTES YOU ARE PREPARING.
USE CONVERT UTILITY TO PUT THEM ON DESIRED DISKETTE. KILL UNNEEDED FILES THAT MAY HAVE BEEN CONVERTED.
BEGIN PROGRAM DEVELOPMENT AND TESTING

If you have a Model I and you are using a backup of the TRSDOS 2.3 system diskette as distributed, here are some candidates for killing:

```
GETDISK/BAS   TEST2/BAS     DISKDUMP/BAS
GETTAPE/BAS   TAPEDISK/CMD  TEST1A/CMD
```

These are test programs and cassette-to-disk utilities you don't need right now; but you might need the 20 grans they occupy. (Remember, you've still got them on your master.)

In addition, you can delete one of the two invisible files, BASIC/CMD or BASICR/CMD. Those files contain identical copies of the BASIC language interpreter program, excepting that BASICR/CMD has an added line-renumbering feature that takes one extra gran. Since you won't be needing that for now, kill BASICR/CMD to free another 5 grans. The "P" in the directory, however, tells you that the file is protected, so you must know the proper password. Lost somewhere within your owner's manual, Radio Shack kindly provided it for you! The password is "BASIC". The proper command is:

```
KILL BASICR/CMD.BASIC
```

If you need more space, you can also kill the invisible files FORMAT/CMD and BACKUP/CMD. But then, when you need to use either of those functions, you'll have to use a system diskette that still contains them. For now, let's leave them on, but should you ever want to kill either or both of them, here are the commands and passwords you'll need:

```
KILL FORMAT/CMD.FORMAT
KILL BACKUP/CMD.BACKUP
```

If you have a Model III, and you've made your backup from the TRSDOS 1.3 diskette as distributed, here are some candidates for killing from your "work" diskette:

```
LPC/CMD      MEMTEST/CMD   HERZ5Ø/BLD
```

Depending on the type of printer you have, you may need LPC/CMD, so check your manual before killing it. Also read the section about HERZ5Ø/BLD. If it applies to you, follow the instructions. Then you can kill it. In any case, you can kill MEMTEST/CMD. If you've killed all three, you now have 5 more granules of storage to work with on drive 0.

As you continue down the flowchart, you come to a question. Will it be a multiple-drive system? Since you are just going to be practicing, and the space on drive 0 is enough, your answer is "no", and you can bypass the steps that would be required to prepare diskettes for the other drives.

The arrows lead you to another question. Will other programs or files be needed? Since you will be keying in all your practice examples, the answer is "no", and the flowchart tells you that you are ready to begin program development and testing!

In the following pages we'll examine BACKUP, FORMAT, KILL, COPY and CONVERT, as well as a few other useful DOS commands in detail.

Using BACKUP

Your disk operating system owner's manual gives you the basic instructions for using the BACKUP command to duplicate the contents of one diskette onto another. Our purpose here is to look at the backup process in more detail, focusing on the issues that often confuse and sometimes trouble users.

The first thing to keep in mind is that BACKUP is a program that resides on your drive 0 TRSDOS system diskette as an invisible file named BACKUP/CMD. As a program, it is loaded into memory from disk as soon as you type "BACKUP" and press <ENTER> in response to the DOS Ready prompt. Once it is in memory, you are free to remove your system diskette from drive 0 if you wish to use this drive in the backup process. When the BACKUP program terminates, either upon successful completion or as the result of a problem, the system diskette must be re-inserted into drive 0. If your TRSDOS system diskette is not re-inserted at the proper time, your computer will hang up, or freeze, when it tries to go back to DOS Ready mode. Should this occur, the only remedy is to insert your system diskette and press reset.

The second matter to clear up is the terminology used during the process. The "source" diskette is your original. It is the one that you want to duplicate. The "source" drive is the disk drive in which you insert your source diskette. The source drive may be any drive connected to your system, and is referenced by drive number. Thus, if you have a 4-drive system, any drive: 0, 1, 2 or 3 may be specified as the source.

The "destination" diskette is the diskette that you've selected to duplicate the source onto. It may be a new diskette, right out of the package. Perhaps the diskette was used to hold unrelated data and programs that are no longer needed. In most cases, it will be a diskette that contains mostly the same information as the original, but now obsolete because of additions, deletions or changes to data or programs on the original. The "destination" drive is the disk drive in which you'll insert your destination diskette. Again, you may select any drive in the system by its number. If you've got two or more drives in your system, the source and destination drives should be different to make the process as quick and easy as possible. If you have just one drive, or just one drive that is working properly, the source and destination drives must be the same.

If the backup program mentions "system diskette" it is, of course, referring to the diskette that contains your disk operating system, TRSDOS or a counterpart. In some cases, the source diskette and the system diskette will be one and the same. This situation would apply if, for example, you want to make a duplicate of your system diskette, residing in drive 0, onto another diskette.

BACKUP with Model I TRSDOS

The BACKUP procedure on a Model I with TRSDOS can be quite simple if you have 3 drives. You can leave your system diskette in drive 0, put the source diskette in drive 1, and the destination disk in drive 2. On the other hand, it can get complex and confusing to a new user when you have just 1 or 2 drives, or if any of the drives are not working properly.

I've trained dozens of people on the backup procedure, and many times I've had to walk experienced users through a difficult backup. To make my job easier, and hopefully yours, I've prepared detailed instructions that anticipate nearly every opportunity for a problem. The instructions are written in a format that I've modified from what's known as a "Playscript Procedure." (See *The New Playscript Procedure*, by Leslie H. Matthies, Office Publications, Inc., Stamford, Connecticut.) Each step in the process is performed by an "actor." In this case, there are two actors: you and the computer. Each step is numbered so that the instructions can have you skip to other steps out of normal sequence. In the left column is one of seven statements. The right column gives the details. Here are the seven statements that may appear in the left column:

"Computer displays" is used when the computer is displaying a specific message.

"Computer shows" is used when the computer is showing some data that may be variable or dependent on the results of your entry.

"Computer" is used when the computer is performing some other operation.

"You type" is used when you are to type a specific character, word, or phrase, terminated with the <ENTER> key.

"You enter" is used when you are to enter variable information on the keyboard, terminated by pressing the <ENTER> key.

"You press" is used when you are to press a specific key or button.

"You" is used when you are to perform some other operation.

The word "may" is inserted into any of these phrases if an event is optional. Actions contingent upon optional events are given "–" instead of a step number.

Now, here are the detailed instructions for the BACKUP procedure, in "Computer-Script" format:

Figure 2.2 — *Model 1 TRSDOS Backup Procedure*

STEPS

1. Computer displays:	DOS READY
2. You type:	BACKUP
3. Computer:	Loads backup program to memory from the system diskette.
4. Computer displays:	SOURCE DRIVE NUMBER?
5. You may:	Remove system diskette from drive Ø if you wish to use drive Ø as either a source or destination drive.
6. You:	Insert source diskette in the drive you wish to use as a source drive.
7. You may:	Insert destination diskette in the drive you wish to use as the destination drive, if you are using more than one drive.
8. You enter:	The number of the drive that you've chosen as the source drive.
9. Computer displays:	DESTINATION DRIVE NUMBER?
1Ø. You may:	Re-insert system diskette and press reset if you made an error when entering the source drive number. (Computer will return to step 1.)
11. You enter:	The number of the drive that you've chosen as the destination drive.
12. Computer displays:	BACKUP DATE (MM/DD/YY) ?
13. You may:	Re-insert system diskette and press reset if you made an error when entering the destination drive number. (Computer will return to step 1.)
14. You enter:	Today's date, using a 2 digit month, day, and year, separated by slashes.

15. Computer: Checks the source drive you've specified (to see if it's ready and to get the PACK ID.)

16. Computer may display: SOURCE DRIVE NOT READY!
HIT 'ENTER' TO CONTINUE

- You: Should check the cause of the problem. Is the disk inserted OK? Is the door closed? Is the drive getting power?
- You: Re-insert the system diskette in drive Ø, if not present, and press <ENTER>. The computer returns you to step 1 where you may try again.

17. Computer may display: SOURCE DISK READ ERROR!
HIT "ENTER' TO CONTINUE

- You: Re-insert the system diskette in drive Ø, if not present, and press <ENTER>. The computer returns you to step 1 where you may try again. This time, insert the source disk very carefully, or specify another drive as your source drive.

18. Computer may: Freeze up or "die".

- You: Re-insert the system diskette in drive Ø and press reset. The computer returns you to step 1.

19. Computer: Checks the destination drive you've specified to see if it's ready, to see if the disk inside is already formatted, and if so, to check the PACK ID that is recorded.

2Ø. Computer may display: DESTINATION DRIVE NOT READY!
HIT 'ENTER' TO CONTINUE

- You: Go to step 16.

21. Computer may display: DESTINATION DISK READ ERROR!
HIT 'ENTER' TO CONTINUE

- You: Should try again, but first, use a magnetic tape bulk eraser to erase the diskette, or use a brand new diskette, or FORMAT the diskette, using the same ID that your source diskette has. (The last method sometimes won't work if you're backing up a system diskette).
- You: Re-insert the system diskette in drive Ø, if not present, and press ENTER. The computer returns you to step 1 where you may try again.

22. Computer may display: DESTINATION DISK WRITE PROTECTED!
HIT 'ENTER' TO CONTINUE

- You: Verify that you are using the diskette and drive you intended.
If so, remove the tape covering the notch on the destination diskette.
- You: Re-insert the system diskette in drive Ø, press ENTER, and start over from step 1.

23. Computer may display: PACK ID'S ARE DIFFERENT!
HIT 'ENTER' TO CONTINUE

- You: Verify that you are using the diskette and drive you intended.
If so, use a magnetic tape bulk eraser to erase the destination diskette, use a brand new diskette, or FORMAT the diskette using the same disk ID that your source disk has. (The last method won't work if you are backing up a system disk.)
- You: Re-insert the system diskette in drive Ø, if not present, and press ENTER. Go to step 1.

24. Computer may: Freeze-up or "die".

- You: Go to step 18.

25. Computer: Determines whether diskette has already been formatted. If so, it goes to step 31.

26. Computer: Formats each track.

27. Computer: Verifies that the formatting pattern it recorded in step 26 reads back correctly by checking each track and sector.

28. Computer may display: CRC ERROR! TRACK LOCKED OUT!
(The data read back was different from what it thought it wrote. So, the track number will be noted and it will be removed from use, thus reducing the disk's capacity).

- You: You'll probably want to insist on a perfect backup, so open the drive doors. When the motors stop, re-insert the system diskette in drive Ø and press reset. Start over from step 1, perhaps using another destination diskette or drive.
If you can live with tracks locked

out, you may let the process go on though this reduces the capacity.

29. Computer may display: NOT FOUND - TRACK LOCKED OUT!
(A magnetic pattern recorded when formatting was not found during the verify phase.)

- You: Go to step 28. (The consequences and remedies are the same.)

3Ø. Computer may display: BACKUP REJECTED DUE TO FLAWS ON DESTINATION DISK!
HIT "ENTER' TO CONTINUE
(8 or more tracks were locked out, or a critical track was bad).

- You: Re-insert the system diskette in drive Ø and press <ENTER>. Start over from step 1, perhaps using a different destination diskette or drive.

31. Computer displays: LOADING, COPYING, VERIFYING.
It does this for each track on the source diskette that contains data. When loading, it is reading a track from the source disk into memory. When copying, it is writing from memory to the destination disk. When verifying, it is comparing the data that it still has in memory to what it reads back from disk.

32. Computer may display: SOURCE DISK READ ERROR!
HIT "ENTER' TO CONTINUE
It has found a sector that has unreadable or damaged data.

- You: Re-insert the system diskette in drive Ø, if not present, and press <ENTER>. Start over from step 1, perhaps using a different source drive. If repeated attempts fail you will need to reconstuct the data on your source by reprocessing from another backup copy, or by using special copying techniques.

33. Computer may display: DESTINATION DISK WRITE ERROR!
HIT "ENTER' TO CONTINUE
The computer is having trouble with the destination diskette or drive.

- You: Re-insert the system diskette in drive Ø, if not present, and press <ENTER>. Start over from step 1,

```
                          perhaps using a different diskette
                          or drive as your destination.
34. Computer may display: DESTINATION DISK READ ERROR!
                          HIT "ENTER' TO CONTINUE
                          The consequences and remedies are
                          the same as for a write error.
 -  You:                  Go to step 33.
35. Computer displays:    BACKUP COMPLETE
                          HIT "ENTER' TO CONTINUE
                          The backup process has ended. Any
                          other termination results in a bad
                          and unusable backup.
36. You:                  Remove the destination diskette and
                          label it appropriately.
                          Remove the source diskette, and, if
                          it's not present, re-insert the
                          system diskette in drive Ø.
37. You press:            <ENTER>
38. Computer displays:    DOS READY
                          You are ready to perform another
                          backup from step 1, or any other
                          computer function.
```

As you can see, the relatively simple process of making a backup can be broken down into many steps. Fear not though! In most cases the whole procedure takes only about a minute. In other sections of this book we will be examining techniques that will help you when a stubborn diskette just won't back up. We'll also look at ways to keep your backups organized.

BACKUP with Model III TRSDOS

BACKUP using TRSDOS 1.3 on the Model III is very similar to the TRSDOS 2.3 BACKUP on the Model I, except a few nice enhancements have been added.

You'll come across the first improvement as soon as you specify your source and destination drive numbers. There is no date question. The system uses the date that you entered when you originally booted TRSDOS upon powering up your computer. Or, if you used the TRSDOS library command DATE, it uses whatever month, day, and year you specified then. (See your manual for details on the DATE library command.)

You can replace steps 12 through 14 (in the Model I BACKUP procedure) with a new question. The Model III BACKUP program asks you for the "SOURCE Disk Master Password". This feature helps you protect your disk against unauthorized copying. As long as a disk's master password is secret, and each of the files on the disk is protected by access passwords, relatively few people will have the technical knowledge required to make a copy. A diskette's password is assigned by you or its creator when it is first formatted. At a later date you may wish to change it. If you know the current password, you can change it with the PROT library command.

By the way, Radio Shack distributes the TRSDOS 1.3 diskette with a password already installed. The password is "PASSWORD". You may wish to remove the password so that you don't have to type it in each time you back up a system diskette. Here's how your screen will look when you do it:

```
TRSDOS Ready
PROT :Ø (PW)
Master Pasword? PASSWORD
New Master Password?
TRSDOS Ready
```

It's as simple as that. You simply enter the PROT command, give it the old password, and press <ENTER> to tell it that there is now, in effect, no password.

Returning to the BACKUP procedure, you simply enter the password as requested. If there is no password, just press <ENTER>. If you enter an incorrect password, the BACKUP program aborts immediately, returning to TRSDOS Ready.

Now that we've covered the automatic date and password features of TRSDOS 1.3's BACKUP, let's look at another change. In step 23, on the Model I, the program checked the pack ID of the destination diskette. If it was the same as the ID on the source the backup would proceed, and no re-formatting would be done. If it was different, the program would be aborted. If it was unrecognizable, formatting would begin automatically.

On the Model III, Radio Shack used a different approach. Before beginning the backup, the destination diskette is checked. If it has been previously formatted by Model III TRSDOS the following message is shown:

```
Diskette contains DATA. Use Disk or not?
```

You may answer by entering "Y" or "N". The "no" answer aborts the backup operation, so remember that if a system diskette is not present in drive 0 you'll need to replace it before pressing <ENTER>. The "yes" answer tells the program that, yes, you realize that there is data, and you really do want to write over it.

If you re-use a disk that already has data on it as the destination in a backup, a second question is asked:

```
Do you wish to RE-FORMAT the diskette?
```

A "no" answer provides the fastest backup for you since the formatting process is skipped. Generally you'll want to answer "N" unless you know that there are errors on the destination diskette. You'll be aware of them if you've been using the destination diskette and the system has given you messages indicating disk input or output errors. More likely, you'll choose the RE-FORMAT option in a second attempt after having made one try at the backup, and having it aborted with a destination read or write error.

While we're still covering the keyboard responses available when you set up your backup on the Model III, note that the <BREAK> key is active. If you make an error when answering any of the questions and wish to start over, simply replace the system diskette in drive 0 (if it's not already there) and press <BREAK>. The system displays the message "Operation Aborted" and returns to TRSDOS Ready.

The sequence of events during the automatic process with Model III TRSDOS is slightly different from Model I TRSDOS 2.3. Instead of formatting all tracks and then going back to verify all of them, the verify of each track and sector immediately follows the formatting of the track. After all tracks have been formatted, a "flawed track" count is displayed for you. The system makes about 5 attempts to format a bad track before marking it as flawed. Again, you may wish to insist on a perfect backup with no flawed tracks, so you might want try again with a different destination diskette or drive if any tracks are flawed.

Another difference is that instead of reading, writing and verifying each track individually in the backup, Model III TRSDOS uses all available memory. First it reads as many tracks and sectors as memory will hold. Then it writes the data to the destination disk and verifies it. It repeats this process several times until the diskette is copied. This scheme results in a backup that is slightly faster.

Single Drive BACKUPS

Both systems, Model I TRSDOS 2.3 and Model III TRSDOS 1.3, support a single-drive backup. This makes it possible to make a backup if you have only one disk drive, or if you have more than one drive, but only one is functioning properly.

The single-drive backup is much slower than the two-drive method. To do a single-drive backup, simply specify the same drive number for source and destination. The process involves repeated swapping of the source and destination diskettes. First the source disk is inserted, and the system reads as much data as memory will hold. Then the destination disk is inserted and the data is written and verified. This process is repeated until all data has been copied. Several swaps are required in most cases. These are prompted with the messages:

```
INSERT SOURCE DISKETTE (ENTER)
```

and,

```
INSERT DESTINATION DISKETTE (ENTER)
```

If you make an error, such as putting in the wrong diskette, you can get a message such as:

```
WAKE UP !!!  THAT'S NOT THE SAME SOURCE DISK!
```

Don't count on the system to catch all possible errors you may make though. Concentrate on what you are doing. Better yet, cover the write-protect notch on your source diskette with one of the adhesive strips that was provided in its package.

Using FORMAT

Simple instructions for using FORMAT are in your disk system owner's manual. We'll look at the formatting process in detail here, covering the whens, whys, and hows, as well as the various error messages you may get.

You will probably have no need to ever use FORMAT if you have a single disk drive system. Since a system diskette will always be required in drive 0, you'll always be using a backup of the original system diskette. The usual procedure is to make a backup copy of the system diskette and kill unneeded programs and data files to make work space.

If you have a multiple drive system, you may choose to use system diskettes in every drive. That is, each diskette may be a backup of your system diskette, and you may kill unneeded programs and files to make more storage space. This approach, however, wastes space on the diskettes that are not being used in drive 0, since only the system programs on drive 0 will be used. If you don't need full capacity, it will simplify backup procedures on two-drive systems and give you more flexibility, since any diskette can be used in any drive.

More likely, you'll want maximum capacity on each drive. FORMAT is needed to prepare the diskettes that will not be used in drive 0. As we discussed earlier, to store data, a diskette must be magnetically divided into tracks and sectors; it must have a directory, and its track 0, sector 0 must have been initialized as a boot sector so that the system will know where the directory is. Formatting prepares a diskette in this way. It must be done before you can use magnetically erased or newly purchased diskettes. It may be done on a diskette that contains unneeded data as a way of erasing everything and preparing it to be used for, perhaps, a different purpose. Formatting is also a way to re-use diskettes that may be magnetically or physically damaged. If the damage is only magnetic, the formatting process will make the diskette fully usable again, though it will destroy any data that may have been on it. If the damage is due to physical scratches or imperfections on the diskette, formatting will lock out the bad areas to prevent them from use, and it will make the remaining good areas usable.

Formatting is also an important test procedure since it checks the proper functioning of a disk drive over every track and sector. It writes a magnetic pattern onto every track. Into each sector a pattern of 256 hexadecimal **E5**'s (decimal 229's) is written. Then it reads back each track and sector to verify that the pattern was written correctly. If you suspect that a drive is malfunctioning, try formatting a diskette on it. If you are consistently unsuccessful, especially after trying to format several different diskettes, you now have a specific problem that you can report to the repair people. You're more likely to get the condition corrected the first time around if you can say, "the drive is unable to format tracks 32 through 34" than if you simply complain that the drive seems unreliable.

It is during formatting that you give a diskette its name and master password. The diskette name (ID, or or "pack-ID") of up of 8 characters, will be displayed on your screen whenever you display a directory of the disk. Under Model I TRSDOS 2.3, you can get a quick visual check of the names of all diskettes currently in the system with FREE, a library command. The diskette master password, though rarely useful under Model I TRSDOS 2.3, can be used with Model III TRSDOS 1.3 to prevent or discourage unauthorized backups.

Any drive, including drive 0, may be used to format a diskette. The drive used during formatting does not need to be the drive that will be used during subsequent operations with the diskette you've created. If you do use drive 0, you only need to remember that you must have a system diskette inserted to load the format program, and you must replace it after the formatting process is completed.

FORMAT with Model I TRSDOS

Under Model I TRSDOS 2.3, formatting is done with a program that resides on your system diskette. It has the name "FORMAT/CMD", and it has the "invisible" attribute, so it will not usually appear on a directory display. You begin the process by simply typing FORMAT from DOS READY mode. The detailed procedure, along with nearly every possible error condition, is shown on the following pages.

Figure 2.3 — *Model 1 TRSDOS Format Procedure*

```
1. Computer displays:    DOS READY
2. You type:             FORMAT
3. Computer:             Loads the disk formatter program
                         to memory from the system diskette.
4. Computer displays:    TRSDOS DISK FORMATTER PROGRAM.
                         WHICH DRIVE IS TO BE USED?
5. You may:              Remove system diskette from
                         drive Ø to do the formatting. (This
                         might be done if you have been
                         unsuccessful in formatting with
                         the other drive(s) on the system.)
```

6. You: Insert the diskette in the drive that you've chosen to use.

7. You enter: The number of the drive that you've chosen: Ø, 1, 2, or 3.

8. Computer displays: DISKETTE NAME?

9. You enter: The name, up to 8 characters, that you've chosen to identify this diskette. (In future operations, the name that you have entered here will be shown when you use DIR to display a directory of the diskette, or FREE to identify which diskettes are in the system and how much free space they have.)

1Ø. Computer displays: CREATION DATE (MM/DD/YY) ?

11. You enter: The date, using a 2-digit month, day, and year, separated by slashes. (Normally you will want to enter today's date. For easy typing you can enter ØØ/ØØ/ØØ if you wish.)

12. Computer displays: MASTER PASSWORD ?

13. You enter: A password of up to 8 characters. (Under Model I TRSDOS you will probably never use the password you type here, so just type something simple, like your name or initials.)

14. Computer displays: DO YOU WANT TO LOCK OUT ANY TRACKS?

15. You type: N
(This tells the computer to format the whole diskette, giving it maximum capacity.)

16. You may type: Y
Computer displays: WHICH TRACKS (1-34) ?
You enter: A list of the track numbers, separated by spaces, to be locked out. (Note: You'll probably never have any reason to lock out tracks.)

17. You may: Put a system diskette in drive Ø and press reset if you made an error when answering any of the questions thus far. (Computer will return to step 1.)

18. Computer: Turns on the disk drive you selected and reads the diskette.

19. Computer may display: DISKETTE CONTAINS DATA, FORMAT OR NOT ? (The computer has determined that the diskette is already, at

least, partially formatted, and it may or may not have files on it that you will want to keep.)

You may type: Y

(To confirm that you really want to format the disk you've inserted in the drive you specified.)

You may type: N

(In the event you realize that you may have selected the wrong diskette or drive.) The computer displays: HIT "ENTER' TO CONTINUE. Verify the system diskette is in drive Ø, press <ENTER>, and the computer will return you to step 1.

2Ø. Computer may display: DOOR NOT CLOSED ON DRIVE!

or DRIVE NOT READY!

or DRIVE NOT IN SYSTEM!

or NO DISKETTE IN DRIVE!

You: Check the cause of the problem, verify that a system diskette is in drive Ø, press <ENTER>, and the computer will return you to step 1 to start over.

21. Computer may display: DISKETTE UNUSABLE - HARD SECTORED!

You: Pull out the diskette and rotate the flexible surface within the jacket by inserting your first two fingers in the big center hole. If more than one small hole appears through the round opening near the center you've purchased a "hard sectored" rather than a "soft sectored" diskette. Get your money back! Then verify that a system diskette is in drive Ø, press <ENTER>, and go back to step 1.

22. Computer may display: WRITE PROTECTED DISKETTE!

(It is telling you that, either the diskette is in backwards or upside down, or the notch on it is covered by a small piece of tape, preventing the drive from recording on it.)

You: Verify that the diskette and drive you specified are correct. Then, assuming the diskette was inserted correctly, remove the tape covering the notch, verify that a system diskette is in drive Ø, press <ENTER>, and go back to step 1.

23. Computer displays: DIRECTORY WILL BE PLACED ON TRACK 17 (Unless you locked out track 17 in step 16).

24. Computer displays: FORMATTING TRACK ØØ.

25. Computer shows: The track number it's formatting, ØØ through 34, as it formats it.

26. Computer may display: CAN'T FORMAT - DISK TRANSFER RATE TOO FAST!

or CAN'T FORMAT - MOTOR SPEED TOO FAST!

or CAN'T FORMAT - MOTOR SPEED TOO SLOW!

You: Re-insert the system diskette in drive Ø, press <ENTER>, and go back to step 1 to try again. (Sometimes these messages will indicate a temporary problem, so it pays to try again a few times after checking connections, trying a different diskette, or a different drive. If the problem persists, have your computer repaired.)

27. Computer displays: VERIFYING TRACK ØØ

28. Computer shows: The track numbers, ØØ through 34, and the sector numbers Ø through 9 on each track as it verifies them.

29. Computer may display: NOT FOUND!!! TRACK LOCKED OUT!

or CRC ERROR!!! TRACK LOCKED OUT!

(It may give you either of these messages for one track or several. If there are too many flaws, or if the directory track is flawed, the formatting process will abort.)

3Ø. Computer displays:

```
INITIALIZING SYSTEM INFORMATION
FORMATTING COMPLETE
HIT 'ENTER' TO CONTINUE
```

31. You: You should verify that the system diskette is in drive Ø, and press <ENTER>. If tracks were locked out the formatted diskette will have less than maximum capacity. You may accept it as such, but more likely, you will want to try again from step 1. If troubles persist, use a different diskette. If troubles continue, have the drive repaired or use a different drive.

```
32. Computer may display: CAN'T INITIALIZE SYSTEM INFORMATION
                          HIT 'ENTER' TO CONTINUE
    or                    It may fail to give you the
                          FORMATTING COMPLETE message. (In
                          either case, the diskette is
                          probably unusable. Repeated
                          attempts may format a marginal disk.)
    You:                  Verify that a system diskette is in
                          drive Ø, press <ENTER>, and try again
                          from step 1, perhaps with a
                          different diskette or another drive.
```

If everything goes normally the entire procedure for creating a 35-track formatted diskette with TRSDOS 2.3 on a Model I takes less than a minute. Shown below is the video display as it appeared after formatting a diskette on drive 1. In this case, the name "INVEN" was given to the diskette, because it was to be used as storage for inventory files. I arbitrarily used a password of "LR" and a date of September 5, 1982. As you can see, the diskette had previously contained data, but I chose to erase it since it was no longer needed:

```
TRSDOS DISK FORMATTER PROGRAM 2.3

WHICH DRIVE IS TO BE USED ? 1
DISKETTE NAME ? INVEN
CREATION DATE (MM/DD/YY) ? Ø9/Ø5/82
MASTER PASSWORD ? LR
DO YOU WANT TO LOCK OUT ANY TRACKS ? N
DISKETTE CONTAINS DATA, FORMAT OR NOT ? Y
DIRECTORY WILL BE PLACED ON TRACK 17
FORMATTING TRACK 34
VERIFYING TRACK 34, SECTOR Ø9
INITIALIZING SYSTEM INFORMATION

FORMATTING COMPLETE

HIT 'ENTER' TO CONTINUE
```

Upon completing the formatting procedure (and the backup procedure as well), it is good practice to remove the disk you've made, re-insert it, and display a directory. This provides another check to see that the data has been recorded correctly. In rare cases, it is possible that the diskette surface may not have been centered properly within the drive during the operation. In those situations, the diskette will work fine until you remove and re-insert it.

Here's the directory display of the disk formatted in the illustration. It was obtained by typing "DIR :1" from DOS Ready. The diskette, of course, was inserted in drive 1.

```
FILE DIRECTORY - DRIVE 1    INVEN   - Ø9/Ø5/82


DOS READY
```

As you can see, the directory is empty, but the diskette has the name and date that was assigned. If you display a directory, requesting invisible files and system files, you will see that the formatting program created a directory and BOOT file on the diskette. The command is "DIR :1(A,S,I)".

```
FILE DIRECTORY  - DRIVE 1    INVEN    -  Ø9/Ø5/82

BOOT/SYS SIP        LRL= 256 / EOF=   5 / SIZE=     1 GRANS
DIR/SYS SIP         LRL= 256 / EOF=  1Ø / SIZE=     1 GRANS
```

If you type FREE from DOS Ready, assuming you have the system diskette in drive 0 and the INVEN diskette you've created in drive 1, here's what you get:

```
DOS READY
FREE
DRIVE Ø  - TRSDOS       12/3Ø/81       38 FILES,    21 GRANS
DRIVE 1  - INVEN        Ø9/Ø5/82       48 FILES,    67 GRANS
```

The display shows the diskette name and date you assigned, the fact that you've got space for 48 file names in the directory, and storage space of 67 granules. If any tracks had been locked out during formatting, the free granule count would have been reduced by 2 grans per track locked out.

FORMAT with Model III TRSDOS

On a Model III with TRSDOS 1.3, the procedure for formatting a diskette is very similar to the one we examined for Model I TRSDOS. The messages on the display are worded a little differently, but the steps and potential error conditions are about the same. You enter the program by simply typing FORMAT from TRSDOS Ready. Here is the display you get on a Model III as you format a diskette in drive 1, using the name "AR1" and the password "JP":

```
TRSDOS Model III Disk Formatter  Ver 1.3

Format Which Drive? 1
Diskette Name? AR1
Master Password? JP
Formatting track 39          Verifying sector 18
ØØ Flawed Tracks

Writing System Information
TRSDOS READY
```

Note several differences:

- The creation date is automatically assigned to the disk, based upon what you entered when you started up your Model III, or what you changed it to with the DATE library command.

- The master password is requested, but it assumes a more important role. The password you enter here will be required every time that you use BACKUP to duplicate the disk. Unless you have a special reason for password protection it is best to just press <ENTER> in response to the "Master Password?" question so that the diskette will have no password.

- The TRSDOS 1.3 FORMAT program does no pre-check of the disk you are formatting. That is, it doesn't warn you if the diskette already contains data, so be careful!

- The program gives you no opportunity to manually lock out tracks, but, like the Model I FORMAT, it will automatically lock out flawed tracks if necessary.

- Since the Model III uses 40-track, double-density, tracks 0 through 39 are formatted. On each track, sectors 1 through 18 are created. Each track is verified immediately after it is formatted, rather than first formatting all tracks and going back for a second pass to verify them.
- The program gives you a count of the flawed, and consequently, locked out tracks. Again, for maximum capacity and reliability, you'll usually want to insist on a perfect format with 0 flawed tracks, so it may be necessary to make more than one attempt in some cases.

COPY, RENAME, and KILL

COPY, RENAME, and KILL are the three library commands that are most useful for manipulating the individual files that reside on a diskette.

COPY makes an exact duplicate of an individual file. You will normally use copy to duplicate a file from one diskette onto another in a different drive. Let's say, for example, that you've formatted a diskette and it is inserted in drive 1 of your TRS-80 Model I. You want to copy the DISKDUMP/BAS program from your TRSDOS 2.3 system diskette in drive 0 onto the diskette in drive 1. Your command from DOS Ready is:

```
COPY DISKDUMP/BAS:Ø TO DISKDUMP/BAS:1
```

Upon completion, you can display a directory of drive 1 and you'll see that it now has the file, "DISKDUMP/BAS".

TRSDOS 1.3 on the Model III uses the same syntax, as well as some simplifications, in the COPY command. Suppose you've got a formatted diskette in drive 1 and your system diskette in drive 0. To copy the file MEMTEST/CMD from drive 0 tȯ drive 1, your command is:

```
COPY MEMTEST/CMD:Ø TO MEMTEST/CMD:1
```

You can simplify it, on either Model I or III, by typing:

```
COPY MEMTEST/CMD:Ø TO :1
```

or,

```
COPY MEMTEST/CMD:Ø :1
```

The Model I and Model III TRSDOS COPY commands also let you change the file name during a copy. The command,

```
COPY TESTPROG:1 TO TP:Ø
```

... duplicates a file named TESTPROG on drive 1, if one exists, onto drive 0. The resulting file on drive 0 is identical to TESTPROG, but is identified by the name TP in the directory. Changing a file's name during a copy can be important in some cases. If, in example, you already had a file named TESTPROG on drive 0 and used the command,

```
COPY TESTPROG:1 TO TESTPROG:Ø
```

. . . the contents of TESTPROG on drive 0 would be overwritten by the contents of TESTPROG from drive 1. If you knew that the contents of both files were different before the copy and wanted to preserve the original contents of TESTPROG on drive 0, then using a different name would be necessary, as you did with TP.

You can also make 2 or more copies of the same file on one diskette. The only requirement is that you use unique names to identify the different copies. Let's say, for example, that you want to do some modifications to a program named SUPERPRO on drive 0. Until you get the modifications fully tested, you want to keep a copy of the original version. To do it, type:

```
COPY SUPERPRO:Ø TO SUPERPRO/OLD:Ø
```

Now you can make modifications to the program, SUPERPRO. When you're confident that you no longer need the original, you can kill it with the command:

```
KILL SUPERPRO/OLD:Ø
```

One of the main things to be concerned about, when using COPY, is disk capacity. It sometimes happens that the COPY command will abort with a DISK FULL error. You can avoid this problem by displaying a directory to see the number of granules occupied by the original file, and displaying a directory of the source disk to see if enough space is available. If you do get a DISK FULL error, or if the COPY command is aborted by any other error condition, be aware that an incomplete version of the file will be recorded on the destination diskette. It is up to you to kill the incomplete file to remove it from the directory.

RENAME is a command that changes the name of any file in the directory. In the last example, for instance, you could have used the following commands:

```
RENAME SUPERPRO:Ø SUPERPRO/OLD
COPY SUPERPRO/OLD:Ø TO SUPERPRO:Ø
```

The RENAME command in this case changed the name in the directory from SUPERPRO to SUPERPRO/OLD. The contents of the file were not changed. The COPY command then created a new file named SUPERPRO, having the same contents as the old one.

RENAME can be useful in other cases. Suppose you have two files on the disk in drive 0 and you want to swap their names. Suppose that one has the name CURDATA and the other has the name OLDDATA. You can use the following commands to swap the names:

```
RENAME OLDDATA:Ø XX
RENAME CURDATA:Ø OLDDATA
RENAME XX:Ø CURDATA
```

The name, XX, was chosen arbitrarily and used temporarily because you cannot have two files on a diskette with the same name. You'll probably find other situations where it is useful to temporarily change a file's name to a "dummy" name, just to make the file manipulations you need.

You've seen several examples of the KILL command thus far. The important thing to remember is to make sure that you really want to erase a file before you use the KILL command.

If you think that there is a chance you might later have a need for a data or program file and are considering KILLing it, sometimes it's better to just rename the file. Then, you'll have it if you need it, and you can always KILL it later if disk capacity becomes a concern. The main benefit of KILL is that the space occupied by an unneeded file is recovered.

Though it is not required, it is good practice to always specify the drive number for the file to be killed. The commands,

```
KILL TEST
```

and,

```
KILL TEST:Ø
```

. . . will both erase a file named TEST on drive 0 if it exists. If by accident, no file named TEST was on drive 0, the KILL command would search drives 1, 2, and 3 for a file by the same name, and if found, it would be erased from a diskette that you did not intend.

KILL does not affect the contents of a file. It merely erases the file name from the directory and updates the granule allocation table. If you unintentionally kill a file, the data is still on the disk. Special recovery techniques can get it back if you discover the problem immediately, but they can be tricky and time consuming.

Important Techniques in Disk Basic

This chapter is an introduction to some of the elementary features of Disk BASIC. If you are new to Disk BASIC on the TRS-80, my goal is to give you the background you'll need to put it to work. If you are already a casual or more experienced user, I hope that this section will give you a deeper level of understanding of what exactly is going on when you use the various features of the system.

Disk BASIC adds a number of enhancements to BASIC which are not directly related to disk accessing. Among them are &H, DEF FN, DEF USR, INSTR, LINE INPUT, MID$=, and USRn. From time to time you'll see some of these features used in a program or example in this book. If you are not familiar with them, your *Disk System Owner's Manual* is the place to look for a detailed explanation.

In this chapter we will take a detailed look at program writing, and storage, as it applies to memory and disk. Then, we will look at how a BASIC program gets the data it requires in the first place, how it can store and use the data, and why disk file storage may be necessary.

BASIC and the DOS

Thus far, we've been getting familiar with the most useful commands available to us from DOS Ready or TRSDOS Ready mode. We've been communicating directly with the disk operating system. If we compare our total disk storage capacity to a warehouse, we've, in effect, been acting as a warehouseman with a little lift truck, moving boxes (files) around, changing the labels (file names) on them, disposing of them (using KILL) and inventorying them (with DIR).

We've been doing all this with little or no concern about what was in those "boxes." Now we'll be assuming a different role. By analogy, we're now going into the front office where we will make the management decisions about what files we will require, what will go into them, and how they will be organized.

BASIC is the name of the programming language that we will be using. Once you know the language, you can specify commands according to its rules. The BASIC language interpreter, itself a program, will translate the commands into codes that the microprocessor in your TRS-80 can understand and implement. When the instructions request operations related to disk access, the BASIC interpreter will automatically call upon special sub-programs within the disk operating system to execute them.

The BASIC language interpreter is a program, but in day-to-day operations you'll rarely think of it as such. It is supplied on your disk operating system diskette in an invisible system file called "BASIC/CMD". Be aware though, that it is not really a part of the disk operating system. If you've purchased the calculator spread-sheet program VISICALC, you may note that it was distributed to you on a system diskette containing the entire TRSDOS disk operating system, but the file BASIC/CMD is not on it. You can do every possible disk operation by programming in machine language, and calling the proper disk operating system subroutines when required, as the authors of VISICALC did, but BASIC makes your job infinitely simpler.

BASIC, as a programming language, lets you specify instructions for your computer in statements and mathematical formulas that are quite consistent with the way we think and communicate. Disk BASIC extends simpler versions of LEVEL II BASIC to provide commands that let you write programs to control the contents of disk files. You can define the information that they will contain and how they are organized.

You can write programs that allow the operator to enter data into the files, to change or modify existing disk file data, and to display or print out the information in disk files. In addition, you can create programs that automatically manipulate the contents of disk files, as when you want to re-arrange the sequence of the data records they contain, or to clear or re-calculate the numerical totals they might contain.

One of the main benefits of BASIC is that it is relatively standard over a wide variety of computer systems. The disk BASIC commands used on the TRS-80 computer are the same, with only a few differences, as those used on hundreds of other computer makes and models. Many of those computers use microprocessors that are quite different from the Z-80 in the TRS-80, and their disk operating systems are quite different from TRSDOS, but their version of BASIC is similar to ours. BASIC is known as a "higher level language." As such, you are insulated, to a large extent, from the peculiarities of the computer and disk operating system.

How to Enter BASIC

Look at BASIC as a specific operating mode for your computer. Thus far, you've been working under the other major mode. I've called it "DOS READY" mode or "TRSDOS Ready" mode, because it is characterized by one of those messages on your display, and while in it, your computer is under control of the disk operating system. When you go into BASIC mode, it is as if you are going to a higher level. Personally, I have a mental picture of going upstairs when entering BASIC. When I return to DOS READY, it's like going back to the ground floor again.

To start, let's go into BASIC the easiest way. Then we'll examine the various options in the procedure.

Figure 3.1 — *Entering BASIC*

```
1. Computer displays:   DOS READY
   or                   TRSDOS Ready
2. You type:            BASIC
3. Computer:            Loads the program, BASIC/CMD, from
                        your system diskette.
4. Computer displays:   How Many Files?
5. You press:           ENTER
6. Computer displays:   Memory Size?
7. You press:           ENTER
8. Computer displays:   READY
                        >
```

Look at the first and last steps in the procedure. You've gone from "DOS READY" to "READY". You are now in BASIC's "READY mode". It is characterized by the word "READY" and the ">" symbol, or simply by the ">" symbol at the leftmost column of your display. From READY mode you can do two types of operations:

- You can enter direct commands. Direct commands are specific requests to the BASIC interpreter to do something now.
- You can enter or edit program lines. Program lines always begin with a line number from 0 to 65529. Each program line may contain one or more statements to be executed later when you "run" the program.

A specific direct command that we should cover now is CMD"S". It exits BASIC to go back to DOS READY mode. To make it clear, here are the steps:

Figure 3.2 — *Exit to DOS*

```
1. Computer displays: READY
                          >
   or just                >
2. You type:              CMD"S"
3. Computer displays:     DOS READY
   or                     TRSDOS READY
```

The same thing can be accomplished by pressing the reset button and holding down the <ENTER> key until the DOS Ready prompt appears, but CMD"S"can be more convenient on the Model I, and it is the code recognized by BASIC in direct mode as well as within a program line.

When entering BASIC you were asked two questions, "How Many Files?" and "Memory Size?" These questions were required to assist the system in allocating the memory to be used during your BASIC session. You pressed <ENTER> in response to both, which told the system to use default values:

```
How Many Files:     Default = 3
Memory Size:        Default = 65535 for 48K TRS-8Ø's
                    Default = 49151 for 32K TRS-8Ø's
```

The "How Many Files" question could have been answered with a number from 0 to 15. Each file requires a certain amount of work space in memory, and your response lets BASIC compute how much space to set aside. For example, if you plan to be working with at most 2 disk files simultaneously during your BASIC session, your answer to the question is 2. This permits BASIC to reserve a minimum of memory space for file handling and it maximizes the memory you have available for other purposes.

Your response to "Memory Size" puts an upper limit on the memory to be used by BASIC for program storage, data storage and work space. If you were to respond to the question with 60000 on a 48K TRS-80, all memory addresses from 60001 to 65535 would be untouched. With this capability, you can reserve a special area for machine-language subroutines or data storage where it will not be unintentionally disturbed during your BASIC session.

To make these ideas clear, try thinking of computer memory as the top of a large office desk. When you boot the system, certain modules of the disk operating system are loaded into memory. It is as if you are pulling a book out of a book case, opening it up to a certain page, and placing it at the far left side of the desk. On that page are instructions for performing certain operations, but in addition, there is a table of contents that tells which other pages explain the procedures for performing other operations. The open book on the desk has reduced the work space on the desk top, but there is still plenty of room.

Now, to continue our analogy, when you go into BASIC, you are, in effect, pulling out another how-to book from the book case, opening it up, and placing it on the desk next to the other one. This book provides all the instructions required to interpret and execute BASIC language commands. To avoid being redundant, it, where necessary, refers to page numbers in the first book (the disk operating system) to specify the detailed steps required for disk access. Again, you've reduced the free desktop work space because you now have two books open.

You enter BASIC from DOS READY or TRSDOS Ready mode by simply typing the word BASIC and pressing <ENTER>. The screen clears, and in the upper right corner you see the message:

```
How Many Files?
```

The system is asking how many files you intend to use during this BASIC session. In effect, it is asking how much space to reserve on the memory "desktop" for other "books" that you might want to pull down from the book case. There will be many situations where it will be useful to have more than one file open at a time. In fact, you can use space for up to 15 files at a time.

Each file will require a certain amount of work space. On Model III TRSDOS 1.3 it is 360 bytes per file. On Model I TRSDOS 2.3 it is 290 bytes per file. Some of that work space will be needed to keep certain statistics. By analogy those statistics will note such things as how many "pages" the file contains, and what "page" you are on. In addition, work space will be required for the current page itself. This space is called the file's buffer.

Assuming you anticipate needing work space for 2 files, answer "How Many Files?" by entering 2. Then the system asks second question:

```
Memory Size?
```

Let's expand the analogy by assuming that the desktop is 65,536 millimeters wide, each millimeter corresponding to a byte of memory. The "Memory Size" question is asking if you want to reserve space on the far right side of the desk. If you answer by entering 65000, then 535 millimeters (or bytes) will be reserved. If you answer by simply pressing <ENTER>, no space will be reserved at the far right side of the desk (upper memory). Now the space between the area reserved for the file buffers and the upper limit specified with the memory size response is the "scratch pad" work area. This is where you will, in effect, write down your BASIC language instructions. In other areas of the available scratch pad space, you will be jotting down memos from time to time, to temporarily record numbers and words that you want to remember.

For accuracy, we should probably make one refinement to the analogy used so far. Low memory on the Model I and III is pre-programmed. It is "ROM", or "read only memory." You can not overlay that area with other programs. It is as if the far left side of the desk has a series of instructions permanently painted onto it. When you boot the disk operating system, those instructions tell the system where to find the "DOS book" in the book case, it retrieves the information and places it just beyond the ROM area. In addition, except on the Model II, the permanent ROM area contains most of the how-to information for the non-disk capabilities of the BASIC interpreter.

Program Files, Sequential Files and Random Files

Perhaps we've stretched the "desktop" analogy too far, but it is the best one I've found to illustrate the process of organizing memory when using BASIC on a TRS-80 computer. It is also a help in introducing the disk file types that you can use in BASIC. When you think of files on a diskette as being like books in a bookcase, several useful comparisons come to mind. First, there can be many kinds of files, just as there are many kinds of books.

Program files correspond to "how-to" books. "How-to" books can be very technical, containing complicated number codes, tables, and detailed schematics — as machine-language CMD files do, or they can contain easier to understand instructions — like BASIC language program files do.

Data files may contain text or other types of information that is written from beginning to end and is meant to be read from beginning to end, like a novel or a history book. Files organized this way are called "serial files," or among TRS-80 users, they are most often called "sequential files."

Other data files may contain information that is meant to be accessed as required without any pre-determined sequence. Access and data organization for these files sometimes compares to the way you might use a telephone book. You turn directly to the page you need to get the information you want. It might compare to an accounting ledger-book. You turn to the page having the account you want to access, and you can retrieve information from that page or write new notations on it. These types of files are called "random access" files. With Model II TRSDOS they are called "direct access" files, a better name for the same thing.

Another aspect of the book analogy helps us understand how files are accessed. When you read a book, you look at only one page at a time. When you access a file, you look at only one "record" at a time. The contents of that record are copied from the diskette into the proper disk buffer area in memory. With random files, the records are neatly organized so that all the records are of a uniform length. Each information "field" within the record is always at the same position, just as, in a phone directory, the name is always in one column; the number is in the next, and a certain amount of space is allocated for each.

When you read a novel, you also only read one page at a time, but your attention is not on the page, it is on the word or sentence you are currently reading. It is like that with sequential files. As you read the words and sentences, each of varying length, the operating system, under the control of our BASIC program is, in effect, turning the pages for you. You hardly notice it if a word or sentence is continued on the next page because the system automatically gets the next record for you. The same applies if you are writing a sequential file. You feed the system data, and it records the data from beginning to end, turning the "pages" when necessary.

In the following chapters we will be looking at sequential and random file programming techniques in detail. Don't worry if the concepts are still a bit foggy. Things will start to clear up as we look at the types of data we can record in disk files, how it is done and applications for the techniques.

Getting Started

We'll be going through several hands-on examples in this chapter. To accommodate readers who only have one disk drive, we will use drive 0 for all our practice programs and data files, so to get ready you should take the following steps:

1. Prepare a system diskette for drive 0 by making a backup of the TRSDOS diskette that was provided with your computer. Chapter 2 of this book and your *Disk System Owner's Manual* give you the information you need.

2. If you've got a Model I, insert the diskette you've made in drive 0, go to DOS Ready, and use the DOS library command, FREE, to verify that your drive 0 diskette has at least 3 granules free.

If you are using a TRS-80 Model III, insert the diskette you've made in drive 0, go to TRSDOS Ready, and use DIR :0 to verify that your drive 0 diskette has at least 5 granules free.

3. If in step 2 you found that you don't have enough free space on your drive 0 diskette, display a directory of drive 0 to display a list of the files that are present. Since we are working with a backup diskette you may use KILL to erase any or all of the (non-system and non-invisible) files that you see listed.

4. Now, from DOS Ready or TRSDOS Ready, go into BASIC. To do it, simply type BASIC and press <ENTER>. Specify 2 for "How Many Files?" and just press <ENTER> for "Memory Size?".

5. Your display should show the "READY" prompt, followed by a ">" symbol at the start of the next line.

Checking Available Memory

Before you get under way, let's see how much memory you've got to work with. To clear away any programs that may be present in the BASIC program storage area of RAM, type NEW and press <ENTER>. Your screen should clear and the ready prompt should reappear in the upper left corner. Now enter the direct command: PRINT MEM. On my Model I the display shows:

```
READY
>PRINT MEM
38580
```

My Model III shows 38560. Note the figure displayed for you. This is the number of bytes of memory available for storage of any BASIC programs that you may write, and work space for temporary data that the programs may use.

Now let's demonstrate how the "How Many Files?" response affected the amount of memory available. Go back to DOS READY by typing CMD"S". Again, come back into BASIC as you did before, but answer the "How Many Files?" question with 3. In the last chapter we said that the answer to this question tells the system how much space to set aside for our file work areas, or "buffers." Let's check it out by entering the command, NEW, and PRINT MEM.

My Model I now says 38290. My Model III says 38200 (Your system will possibly give different numbers if you are using a different DOS version or a different memory size.)

Notice now that, since you specified 3 file buffers rather than 2, you've got less memory to play with. The difference is the amount reserved by BASIC to be ready to handle the additional file. Here are the results:

		---- MEMORY ----		
				Bytes Required
Computer	DOS	2 Files	3 Files	Per File Buffer
Model I	TRSDOS 2.3	38580	38290	290
Model III	TRSDOS 1.3	38560	38200	360

For the short programs you will be using in the next few pages you've got plenty of memory to play with, and though we have not yet discussed the reasons for multiple files, two points are worth keeping in mind for future programming sessions:

- There is no convenient way to change "how many files" while in BASIC. You must go back to DOS READY and re-enter BASIC, to change the setting.
- When you begin using larger and larger programs, your files setting can be very important to the efficient use of memory. For small programs it won't be too critical.

Writing and Saving a Program

One of the most important applications for disk files is the storage of BASIC programs. To introduce a few points, let's write a short program. To start, enter the NEW command to clear memory. Then type the lines shown below, pressing <ENTER> after each:

Figure 3.3 — *MPGCALC1/BAS*

```
 0 'MPGCALC1/BAS
10 CLS:PRINT"MILEAGE CALCULATOR":PRINT
20 INPUT"WHAT WAS YOUR ODOMETER READING, LAST FILL UP";R1
30 INPUT"WHAT WAS YOUR ODOMETER READING, THIS FILL UP";R2
40 INPUT"HOW MANY GALLONS DID YOU BUY, THIS FILL UP  ";G
50 INPUT"WHAT WAS THE PRICE PER GALLON (#.###)       ";PG
60 M=R2-R1:MG=M/G
70 PRINT"MILES DRIVEN:     ";M
80 PRINT"MILES PER GALLON: ";MG
90 PRINT"COST PER MILE:    ";PG*G/M
```

Before you record the program onto the diskette in drive 0, let's test it. To do so, enter the command RUN and answer the questions as they are asked. If you get a syntax error message, go back to the line indicated and correct it. Here's how the display might look after a test run:

```
MILEAGE CALCULATOR

WHAT WAS YOUR ODOMETER READING, LAST FILL UP? 65324
WHAT WAS YOUR ODOMETER READING, THIS FILL UP? 65611
HOW MANY GALLONS DID YOU BUY, THIS FILL UP  ? 11.3
WHAT WAS THE PRICE PER GALLON (#.###)       ? 1.289
MILES DRIVEN:      287
MILES PER GALLON:  25.3982
COST PER MILE:     .Ø5Ø7516
READY
>
```

Now record the program onto the disk in drive 0. The process to use is called "saving" a program. When you save it, you must assign a valid file name. Generally it is best to use a file name that reminds you of the purpose of the program so that, if at a later date, you look at the disk directory you will know what the file is.

For example, we will use the name "MPGCALC1/BAS". I've chosen to use the first 8 letters of the file name to indicate that the program is a miles per gallon calculator. The "1" is there because you will be writing other versions of the same program later in this chapter.

The "/BAS" is the extension. A 3-character extension in a file name is optional. In this case, it is there to remind you that the file is a BASIC language program. The "/BAS" has no meaning to the disk operating system. As far as the DOS is concerned, the file is just like any other file inventoried in the directory.

The command from READY mode is:

```
SAVE"MPGCALC1/BAS:Ø"
```

Upon pressing <ENTER>, the light on drive 0 will come on to indicate that the disk drive is operating. A few seconds later, the READY prompt will be redisplayed.

There are a few things to notice about the SAVE command. First, I added ":0" to the file name to explicitly specify that the program file is to be recorded on drive 0. Had you wanted it to go on drive 1, you could have used ":1". BASIC on the TRS-80 consistently uses this convention of colon and drive number to the immediate right of the file name in program files as well as data files. When a drive number is specified this way, the following operations are performed:

- The system searches the directory of the specified disk for the file name specified.
- If the name is not found in the directory, a new entry is created, the file is recorded, and the free granule count is reduced by the amount of disk storage consumed by the program file.
- If the name is found in the directory of the specified drive, the program is recorded over the previous contents of the file. The free granule count for the diskette may be increased or reduced, depending on whether the new file is shorter or longer than the one it overlayed.

It is not necessary to explicitly specify the drive number on which a program is to be saved. The command could have been:

```
SAVE"MPGCALC1/BAS"
```

When you don't explicitly specify a drive number the system uses the following logic:

- The directory of drive 0 is searched. If the file name is on the diskette in drive 0, the program is recorded onto drive 0, overlaying the existing file having the same name, and the free granule count is updated.
- If the file name is not in the directory for drive 0, the directory of drive 1 is searched. If the file name is on the diskette in drive 1, the program is recorded onto drive 1, overlaying the existing file on drive 1 having the same name, and the free granule count in the drive 1 diskette's directory is updated.
- If the file name cannot be found on 0 or 1, then drives 2 and 3, if they exist and they contain diskettes, are checked in the same way. If the system fails to find the file name on any of the drives, then a new directory entry is created on drive 0 if there is space enough, and the program is recorded there. If there is not enough room on drive 0, the other drives are checked.

Another thing to notice about the SAVE command is that quotes go around the file name. You have to do this because, in BASIC, file names are considered strings. (A string is a series of letters, numbers and/or special characters.) Contrast this with the file names you used when executing DOS commands. No quote marks were required or permitted.

To demonstrate that a program or file name is just like any other string you can try the following from READY mode:

```
A$="MPGCALC1/BAS:Ø"
SAVE A$
```

For more information about file names, you should see the TRSDOS section of your disk system owner's manual. Read the discussions about "File Specification," "File Names," "Drive Specification" and "Passwords."

Error Conditions With SAVE

The most common source of errors during a SAVE is carelessness. It pays to be very careful when typing the save command. If, for example, you accidentally type the file name as "MGPCALC1/BAS:0" instead of "MPGCALC1/BAS:0", the program will be saved, but with the wrong name. Then, if at a later date you want to load and run the program, a File Not Found error will occur. It will very likely require going back to DOS Ready and displaying a directory to discover the cause of your error.

Lets say that you write and save the program with the file name "MPGCALC1/BAS". At a later date you discover a program error or make a modification. After making the changes you save the program again, but this time you accidentally use "MPGCLAC1/BAS". Satisfied with your work you put away your disk. A few days later you pull it out, and load "MPGCALC1/BAS". What happened to the changes you made? Again, going back to display a directory can help you discover your error, since you'll see the two file names, MPGCALC1/BAS and MPGCLAC1/BAS, listed. In this case it is simple, but some times it's a chore to find out which version is the correct one.

I solve these problems by putting my program name and the date as a remark in the first line of the program. You may have noticed that line 0 is:

```
0 'MPGCALC1/BAS -- 09/16/82
```

I've made it a habit, just before saving, to list line 0 to remind myself of the program name I've assigned. The date in line 0 helps to answer questions about which version is the latest if I later find more than one "MPGCALC1/BAS" program on my diskette.

Disk Write Protected
Disk full
Disk I/O Error

In addition to errors that you may make in typing the SAVE command, there are three potential error conditions that can happen.

- Attempting to SAVE to a write protected disk, or a disk that is inserted upside down or backwards.

In either case, the disk drive determines that the notch on the diskette jacket is covered, preventing it from recording on it. Model III TRSDOS displays the message, "Disk write protected." Model I TRSDOS 2.3 displays a message that can be confusing in this case, "TOO MANY FILES". Your corrective action is to remove the tape covering the notch (or check the way the diskette is inserted) and try again.

- Filling up the storage space on a diskette.

The message is “Disk Full”. Only part of the program has been saved or nothing has been saved at all. Usually, though, the program name you used is recorded into the directory. It is important that you erase the partially saved program from your diskette, so your first action should be to KILL the incomplete file. BASIC has a KILL command just like the one we used in DOS Ready mode, except (like SAVE) you've got to use quotes. Here's an example of how the dialog might look on your screen:

```
>SAVE"PROG34:1"
DISK FULL
READY
>KILL"PROG34:1"
READY
>
```

Now you must find somewhere else to save your program, or you must make space for it. It is a good idea to have one or two spare formatted diskettes on hand for this purpose. Then you can save the program onto your emergency diskette, and go back to DOS to use COPY and KILL to move your programs and files around to get your program onto the diskette you want.

The the other solution is to KILL other programs and files while you are still in BASIC to make space. Let's assume, for example, that you are trying to save the program “INPUT/WI” onto drive 0 and you get a disk full message. You realize that you have a program called “VIDEOGEN/BAS” on drive 0 but you know you have other copies of it. To solve your problem, you just kill “VIDEOGEN/BAS”, and re-enter the command, SAVE“INPUT/WI:0”.

If you are not sure what other programs and files are on the diskette in question, you can display a directory without losing the BASIC program you have in memory.

With Model III TRSDOS, you can use CMD“D”. For instance, if you want to examine a directory of drive 1, from READY mode enter

```
CMD"D:1"
```

Model I TRSDOS 2.3 doesn't have that command option. Instead you have to type CMD“S” to go back to DOS Ready, display a directory of the drive in question, KILL one or more files from the diskette, and then return to BASIC by typing BASIC * and pressing <ENTER>. Now you can save your program.

The BASIC * command from DOS Ready is operable on TRSDOS 2.3 for the Model I, as well as TRSDOS for the Model III. It returns you to BASIC from DOS, without clearing the program in memory. It should be used in emergency situations only, because in some cases the program can't be retained. You should note that certain DOS commands cannot be used without destroying the BASIC program storage area. They include COPY, FORMAT and BACKUP.

Another thing to note about the “Disk Full” message is that it will be displayed on a Model III with TRSDOS 1.3 if you try to save a program to a specific drive and a diskette is not present or the drive's door is open.

- A disk I/O error occurs during a SAVE.

The message is DISK I/O error. “I/O” stands for input/output. The computer is having trouble reading from or writing to the diskette. Before the message is displayed, you may notice that the drive appears to get stuck for a few seconds with a repeating grinding or ticking sound.

The first thing you should do is to try eliminating the most common cause of a disk I/O error, an improperly inserted diskette. Check your SAVE command to verify that you specified the proper drive number. Then open the drive door, remove the diskette, and re-insert it carefully, making sure that the diskette surface is properly centered on the wheel inside the drive. Now repeat the SAVE command. I've found that if the first attempt doesn't work, the second or third try will often succeed with this procedure.

The next best remedy for a disk I/O error during a SAVE is to remove the diskette you are trying to save onto and put it into another drive if you've got one available. Generally this solution is only available if you have 3 or 4 drives. If you have a 2 drive system and are trying to save onto drive 0, you can put a spare system diskette in drive 0, put your "problem" system diskette in drive 1, and re-execute the save command with drive 1.

Another way to get around this problem is the remedy we used for a disk full condition. Save the program on a different diskette. Again, it is worthwhile to have one or two formatted "emergency" diskettes around for this purpose.

Once you've exausted the remedies we've discussed so far it's time to assess just how serious your problem is. Does the disk drive light stay on a few seconds before the "grinding" noise begins, or does it begin immediately? If the drive seems to have trouble as soon as the light comes on, the problem is probably in the directory of the diskette, or perhaps, if you are using a drive other than 0, the diskette is not formatted. This is the most serious case. You'll only be able to SAVE your program by using a different diskette or drive.

If the light stays on a few seconds before the disk I/O error appears, you can assume that the system was able to save part of the program, and then a bad spot on the diskette was encountered. One solution is to "sacrifice" another file or program you think may still be good. To do this, save the program using that file's name. This forces the system to use a part of the disk that may not be damaged. Of course you should only use this procedure if you've got a backup of the sacrificed file, if it can be easily re-constructed, or if it is not as important to you as the program you are trying to save. A variation on this remedy is to keep trying the save with different names until you are successful, or you get a disk full error. The logic behind this approach is that during each unsuccessful save, some of the diskette's storage will be used, and you may eventually get to a big enough area of disk space that is good.

Another solution for your problem, in case you suspect that part of the program is being saved before the disk I/O error is being encountered, is to delete those lines from the program that can be most easily re-typed, or those lines that you may already have duplicates of in other disk files. Before trying this desperation move, be sure to make a line printer listing. In fact, anytime you sense that trouble is possible, make a hard-copy printout.

Once you've got all or part of your program saved after fighting a disk I/O error, go back to DOS Ready and check a few things. First, you may need to clean up after your rescue efforts. This may involve killing, renaming, and copying files. Second, make sure there are no other gremlins on the diskette waiting to bite you. Attempt to make a backup of the diskette. Since the BACKUP program reads every track of the diskette containing data, it is the best way to check for problems. If you are unable to make a backup, see the discussion of BACKUP problems in Chapter 16 of this book. Once you've made a backup, use the new diskette. Discard the old one if there are any signs of scratches, smudges or dirt on its magnetic surface inside the jacket.

Before we leave the subject of error conditions during SAVE, we should look at two protection measures that can save you many hours of grief:

- SAVE often, and
- SAVE more than one file.

The first rule is important when you realize that as you type programs into your computer, they are being stored in memory. If you have a power failure or a sudden equipment malfunction, everything you've typed can be lost, but program lines that have been saved on disk are not nearly so vulnerable. We've said that when a program file already exists on disk, a SAVE to the same file name on the same diskette records the new program over the old version. Thus, if you are typing in a long program and you save after typing every 15th line, the worst that can happen if power is interrupted is a loss of the last 15 lines you typed. When power is restored, you can reload the program and retype what you've lost.

The second measure helps to protect you against problems related to the diskette and drive, and also against major errors that you may make when typing. The ideal thing to do is to use two "work" files when typing a long program, and to store them on two different diskettes and drives. Then, each time you save your work, alternate between the files. Suppose, for example, that you are typing in a 100-line program. You might want to use a procedure similar to the following:

Type in 15 lines, and save your work in a file named "WORKA" on drive 0. Type the next 15 lines and save all 30 in a file named "WORKB" on drive 1. Type in the next 15 and save all in "WORKA" on drive 0 again. Continue alternating your saves until the entire program is entered. When the entire program is in either "WORKA" or "WORKB", go to DOS Ready and rename the file to the name you wish to use, or copy and rename it onto the diskette you want to use.

With this procedure, if you encounter a problem during a save, you've got an almost up-to-date version of your work in a different file on a different diskette. It is superior to using just one file because problems during saves often destroy the contents of the file we are saving into. For instance, in the case of a disk full error, the partially saved file is usually useless. But under this method, you'll still have a loadable version of your work in the alternate file. In the case of major typing errors that might not be discovered until after a save (such as accidentally deleting lines, or accidentally typing new lines with the wrong line numbers, unwittingly replacing old lines), you still can recover by re-loading the program from the alternate file.

LOAD and RUN

By implication, the object of saving a program on disk is to have it available for use at a later date. LOAD and RUN are two commands which access programs that have been previously saved. BASIC's LOAD command replaces the program in memory, if any, with one that is in a disk file. For instance, when you enter BASIC from DOS Ready, there is no BASIC program in memory. With LOAD, you can quickly put a program in memory. After it is loaded, you can make modifications to it if you wish, or you can simply type RUN to begin execution. As an example, load the practice program, "MPGCALC1/BAS". The command from READY mode, terminated with the <ENTER> key is:

```
LOAD"MPGCALC1/BAS"
```

The disk drive light comes on, the diskette is searched, and after a few seconds, the ready prompt reappears. You can enter the LIST command to prove to yourself that the program has, in fact, been loaded. It will roll each line of the program before you on the display.

Of course, if you've been following along since the beginning of this chapter, the LOAD command in the last example just replaced memory contents with the same program lines you already had in memory. If you wish, you can enter the command NEW to clear memory of all program lines; then you can repeat the example.

Like the SAVE command, you can specify a drive number in the load file specification. Suppose you have two copies of the program "MPGCALC1/BAS". One is on drive 0 and the other is on drive 1. If you want to specifically request that the one on drive 1 be loaded, enter the command:

```
LOAD"MPGCALC1/BAS:1"
```

There are a few different variations on the RUN command. The most common one is used when you already have the program you want in memory. Simply type RUN and the program starts execution from the first line. Another variation starts execution from a line number other than the first. If, for example, you want to run a program in memory starting at line 140, type RUN 140.

The most useful variation of the RUN command fetches a program from disk and begins execution in one step. To use it from READY mode just type RUN, followed by the disk file name you want. For example, enter:

```
RUN"MPGCALC1/BAS"
```

After a few seconds the program starts, in this case by requesting your odometer reading at the last fill up. The system does three things during this command. First, it closes any data files that were open. (We've not discussed data files in BASIC yet, so just trust me on this point for now.) Then it performs the same functions that were performed with the corresponding LOAD command. Finally, it begins execution of the program.

There is another variation on the LOAD command that can be useful. It works just like the RUN command, but it does not close data files that may be open. (Again, we'll talk about reasons for this variation later.) This option is called LOAD and RUN. It is executed by following the load command with ",R". Try it with the sample program:

```
LOAD"MPGCALC1/BAS",R
```

Chaining Programs

All the examples thus far have dealt with LOAD, RUN and LOAD with the "R" option as direct commands, entered from READY mode. LOAD, RUN and LOAD,R can be used within program lines also. This capability can be very useful for breaking a large application into small program modules. Then, after execution, one program can "chain" to another, and that one to still another. That way, memory capacity is no longer a concern. Smaller programs are also easier to work with and easier for others who read them to understand.

The chaining capability also makes a method for simplifying operator training possible. If you have got several programs, you can create a "menu" for the operator. The menu program displays a list of available programs on the screen and the operator can make a selection by pressing the key indicated. For simplicity, our example will assume that you have got two programs. One is the "MPGCALC1/BAS" program that was introduced earlier. The second is a check register program "CHECKCALC/BAS". A third program could be the menu program. It might look like this:

Figure 3.4 — *MENU/BAS*

```
 0 'MENU/BAS
10 CLS:PRINT"PROGRAM MENU":PRINT
20 PRINT"ENTER <1> TO RUN MILEAGE CALCULATIONS"
21 PRINT"ENTER <2> TO DO CHECK REGISTER CALCULATIONS"
30 PRINT"ENTER <3> TO GO BACK TO DOS READY"
31 PRINT
40 INPUT"WHAT IS YOUR SELECTION";S%
41 IF S%<1 OR S%>3 THEN RUN ELSE ON S% GOTO 100,200,300
100 RUN"MPGCALC1/BAS"
200 RUN"CHEKCALC/BAS"
300 CMD"S"
```

To make this system of three programs work, first write the program "MENU/BAS" and save it. Then load and modify the programs, "MPGCALC1/BAS" and "CHEKCALC/BAS" so that, after execution they automatically re-run the menu program. A modified "MPGCALC1/BAS" program might be saved after adding a few lines at the end as follows:

```
100 PRINT"PRESS <ENTER> TO RE-RUN,"
101 PRINT"OR PRESS <M> TO RETURN TO THE MENU..."
102 LINEINPUTA$
103 IF A$="M" THEN RUN"MENU/BAS" ELSE RUN
```

The check register calculation program could be written similarily, so that, as an option, "MENU/BAS" would be run again. Line 40 in the program shown below does the job:

Figure 3.5 — *CHEKCALC/BAS*

```
 0 ' CHEKCALC/BAS
10 CLS:PRINT"CHECK REGISTER CALCULATOR":PRINT
20 INPUT"WHAT IS YOUR BEGINNING BALANCE";BA
30 PRINT"ENTER <D> FOR DEPOSIT, <C> FOR CHECK, OR"
31 LINEINPUT"ENTER <M> TO RETURN TO THE PROGRAM MENU: ";A$
32 IF LEN(A$)<>1 OR INSTR("DCM",A$)=0 THEN 30
40 IF A$="M" THEN RUN "MENU/BAS"
50 INPUT"AMOUNT";AM
60 IF A$="D" THEN BA=BA+AM ELSE BA=BA-AM
70 PRINT"NEW BALANCE: ";BA
80 GOTO 30
```

The menu program makes it so that the operator only needs to know one command after loading BASIC. The command is,

```
RUN"MENU/BAS"
```

The menu program makes loading the other programs a matter of simple key entries. The display is:

```
PROGRAM MENU

ENTER <1> TO RUN MILEAGE CALCULATIONS
ENTER <2> TO DO CHECK REGISTER CALCULATIONS
ENTER <3> TO GO BACK TO DOS READY


WHAT IS YOUR SELECTION?
```

My other book, *BASIC Faster and Better and Other Mysteries*, shows simpler and more elegant ways to construct menus. It also shows ways to pass variables from one program to another.

The ability to chain can make two or more programs look like just one to the operator, but as a programmer, you should realize that with the techniques we've discussed here, variables are not passed from one program to another. They are cleared with each new RUN command. To illustrate, the variable BA, which maintains a running total of our check book balance in our sample program, is cleared out of the system as soon as you return to the menu program. Special techniques make it possible to preserve the value of BA. Also, some disk operating systems make variable passing an option available with the LOAD and RUN commands. In these systems, you can use a command like RUN"prog-name",V if you want to preserve the contents of all variables currently in memory (where "prog-name" is the name of a program on disk).

More commonly, though, the function of passing values from one program to another is done with data files on disk. A data file can be constructed to contain your checkbook balance, or in the case of the mileage calculator, your last odometer reading. Then, if you go back to DOS Ready or you turn OFF your computer, the data can still be accessed the next time you need it. We will discuss this function of data files later.

AUTO Startups

Most, if not all, TRSDOS-compatible disk operating systems provide a capability that can make your TRS-80 perform a DOS command upon the initial power-up or reset of your system. This capability is called AUTO. The command can be a library command, such as DIR to display a directory of a diskette in a drive. It can also be a command to execute a program, such as LPC/CMD, or since the BASIC interpreter is a program, BASIC/CMD. Actually, the AUTO function can execute any one-liner, just as if you had typed it on the keyboard yourself from TRSDOS Ready or DOS Ready.

TRSDOS has a special AUTO command storage area on the system diskette. Unless you hold down the <ENTER> key, it always executes the AUTO command upon power up (after the date and time are entered with MODEL III TRSDOS) or upon reset. Under normal conditions (and on the system diskette as you received it from Radio Shack) the command in the AUTO storage area is simply the character code that corresponds to pressing the <ENTER> key. You may have noticed that if you just press <ENTER> in response to the TRSDOS Ready or DOS Ready prompt, you get TRSDOS Ready or DOS Ready again. That's what AUTO will do unless you take steps to change it.

To change the command line executed by AUTO, type AUTO and the complete command you want executed. To test it, try this:

Get your computer to display TRSDOS Ready or DOS Ready. Now type:

```
AUTO DIR :0
```

. . . and press ENTER. You've just made it so that a directory of drive 0 will be displayed each time you power up your system or press reset. Go ahead and press reset now to verify that this is the case.

Holding down the <ENTER> key upon power up (just after the time is entered with Model III TRSDOS) or just after pressing reset, temporarily disables the AUTO function. You can try it now by pressing reset and holding down <ENTER> until the DOS or TRSDOS Ready prompt displays. You'll notice that the DIR :0 is not executed.

You may have observed that throughout this book, rather than saying to just press reset to go to DOS Ready, the instructions say to press reset and hold down the <ENTER> key. This was done just in case any readers were working with a system diskette having the AUTO command activated. If you are ever in a situation where you are training an operator or writing operating instructions, be sure to specify that the <ENTER> key must be held down to get to DOS Ready.

It is easy to de-activate the AUTO function. Just type AUTO and press <ENTER>. You'll want to do it now to remove the AUTO DIR :0 we've been playing with.

In cases where you want to change the AUTO command from one thing to another, simply type AUTO and the new DOS command line you want executed. The new AUTO command line will replace the old one.

The AUTO command can be especially powerful for BASIC programmers using Model III TRSDOS. This is because of a special method for going into BASIC that we have not yet discussed. Model III TRSDOS recognizes a one-line command that will load the "how many files" and "memory size" settings and (if desired) RUN a BASIC program. Here's how you can enter BASIC if you want to specify 2 file buffers, set memory size to 60000, and RUN the program named "MENU/BAS":

```
BASIC MENU/BAS -F:2 -M:60000
```

The number following the "-F:" pre-answers the "how many files" question. The number after the "-M:" sets the memory size. The program name to be RUN follows the blank after "BASIC". The program name, files setting or memory setting may be omitted from the one-line BASIC startup, in which case, the default will be assumed. For example, if our command is:

```
BASIC -F:2
```

. . . the system goes into BASIC ready mode, reserving space for 2 disk file buffers, setting memory size to the maximum.

To make it so that your Model III TRSDOS system diskette always automatically starts up with the "MENU/BAS" program, you can enter your AUTO command as follows:

```
AUTO BASIC MENU/BAS
```

Additionally, if you want to specify that space for only 2 (rather than 3) file buffers be reserved, your AUTO command is:

```
AUTO BASIC MENU/BAS -F:2
```

When you are in BASIC on the Model I or III, it is important to note that CMD"S" always returns to DOS Ready. The AUTO command line is not executed.

There may be cases where you want a BASIC program to restart the entire system, including execution of the AUTO command line, just as if the operator had pressed the reset button. You can do it by putting two BASIC commands in a program line. As an example, we'll use line 65000:

```
65000 DEFUSR=0:PRINTUSR(0)
```

This command works like the reset button because it, in effect, tells the Z-80 processor to begin operation from memory address 0.

In situations where you want more than one TRSDOS command executed automatically, Model III TRSDOS provides the BUILD feature that lets you build a file containing a list of DOS commands. The DO library command executes each line in a BUILD file, just as if you were typing it into the keyboard. Suppose, for example, that you are using one of the line printers that requires the LPC program be installed. In addition, you want to execute a FORMS command to set the printer line width to 80 characters. Finally you want your system to go into BASIC automatically with 2 files, running the program MENU/BAS.

You can use BUILD to create a file on drive 0 that contains the necessary commands. You might want to call the file "STARTUP/BLD". From DOS ready you type:

```
BUILD STARTUP:0
```

This creates a directory entry on drive 0 with the name STARTUP/BLD. Now you can type each command that you will want automatically executed, pressing ENTER after each, and an extra ENTER after the last one to tell the system you're done:

```
LPC
FORMS (WIDTH=80)
BASIC MENU/BAS -F:2
```

Now, whenever you want to execute the 3 commands, from TRSDOS Ready you can enter:

```
DO STARTUP
```

Normally though, you'll only want these three commands executed when you power-up or reset the system. So, as the final step, you install your AUTO command line:

```
AUTO DO STARTUP
```

Your disk system owner's manual gives you more details about AUTO and (if they are available) the DO and BUILD commands. All in all, they, along with BASIC's chaining capabilities, make it so that you can construct a system diskette that makes even complex operations simple for the operator. Just insert it, press reset, and go!

Saving in ASCII

The SAVE command stores a program in a disk file in either of two different formats. Up until now you've been using the default — compressed format. The other option is ASCII format. Before you look at ASCII, let's examine the compressed format more carefully.

A SAVE in compressed format copies an exact image of the portion of memory that is storing your program text into a disk file. A one-byte hexadecimal **FF** (decimal 255) precedes the image in the disk file. A two-byte hexadecimal **0000** terminates the memory image in the disk file. The total number of bytes required in the disk file is, therefore, equal to the number of bytes occupied by your program in memory, plus 3.

The program is "compressed" because of the way that BASIC stores each program line in memory. Each program line number is stored in two bytes. Each keyword, such as PRINT, INPUT, FOR, NEXT or GOSUB, is stored as a one-byte token. That is, keywords are stored as a number from 129 to 250. (Remember, one byte can store a number from 0 to 255.) All other text, such as remarks, numeric and alpha constants, is stored byte-for-byte with the codes that, by convention, correspond to the the characters. An "A" has the one-byte code 65, a "1" is a 49, and so forth. Terminating each line is a one-byte 0.

To check it out, let's measure the length of the demonstration program, "MPGCALC1/BAS". The following illustration shows how to measure any program in memory. Your commands are those that follow the ">" symbols:

```
 READY
>NEW
```

The screen clears, and we continue . . .

```
 READY
>PRINT MEM
 38560
 READY
>LOAD"MPGCALC1/BAS"
 READY
>PRINT MEM
 38170
 READY
>PRINT 38560-38170
 390
```

The "NEW" command cleared out any program that may have been in memory so that you could get a "before" memory reading. Then you loaded the program you wanted to measure to get an "after" reading. Subtracting the memory readings gave you the amount of memory consumed by the program — in this case, 390 bytes. (The numbers you get when trying this example will differ if you typed the program differently, or if you are using a different DOS, files or memory size setting. I used TRSDOS 1.3 on a Model III, with 2 files.)

Now lets go back to TRSDOS to look at the directory. Assuming that your result, like mine, was 390 bytes, Model III TRSDOS will show you the following directory entry:

```
                Attrb  LRL  #Rec  #Grn  #Ext  EOF
MPGCALC1/BAS    N*X0   256    2     1     1   137
```

This shows that 2 records of 256 bytes were used. The EOF column shows how many bytes were used in the last record. The program should have taken 390 plus 3 bytes. When you add 256 for the first record, and 137 for the second, you get 393!

You can get some useful information from this display. First, if you subtract 137 from 256 you get 119. You can use 119 more bytes before another 256-byte record will be used. Since you know that (with Model III TRSDOS) there are 3 records per granule, you can use 119 plus 256, or 375 bytes before another granule of storage space will be required. Since TRSDOS allocates disk space in granules, the free capacity on the disk won't be reduced unless you extend the program by more than 375 bytes. Calculations such as this can be quite illuminating when you are trying to squeeze out that last byte of disk capacity! Even though your directory may indicate that 0 grans are free, you probably have more space.

An approach that is more convenient than going back to DOS and displaying a directory is to make the computation in BASIC as a direct command whenever you want to find out how many more bytes you can use before an additional granule of disk storage will be required.

If you are using Model III TRSDOS or another disk operating system that stores 3 sectors per granule type:

```
PRINT INT((B+3)/768)*768+768-(B+3)
```

If you are using Model I TRSDOS or another disk operating system that stores 5 sectors per granule, type:

```
PRINT INT ((B + 3)/128Ø)*128Ø+128Ø-(B+3)
```

In either formula, replace the B + 3 with the current length of your program in bytes, plus 3. Thus, if you've already determined that your program is 390 bytes, you can replace the B + 3 with 393's.

Following through with the Model III example, where the program length was 390 and the MEM count was 38170, the formula tells you that you have 375 bytes to play with before a SAVE will require additional disk space. Now type,

```
PRINT MEM+375
```

. . . and you get 38545. As you program along, you can PRINT MEM from time to time. As long as MEM is less than 38545 the program enhancements won't take any more disk space!

Now that we've discussed the compressed format used in normal situations with SAVE, we can look at ASCII format. In ASCII format, there is no compression of line numbers and key words. What you see on your display or on a line printer listing is what you get in your disk file. One byte of storage is used for each character or blank in the program listing. After each line is the ASCII code for "carriage return." It is hexadecimal **OD**, or decimal 13.

ASCII format almost always results in a longer disk file, but in most cases the difference is not too significant. The sample program, "MPGCALC1/BAS", saved in ASCII, is only 21 bytes longer. ASCII format also makes LOAD and SAVE operations slightly slower, since the system must make a conversion between the compressed format used in memory and the uncompressed format used on disk.

To save in ASCII format, just type the SAVE command as usual, but add ",A" after the file specification. To save "MPGCALC1/BAS" in ASCII, the command is:

```
SAVE"MPGCALC1/BAS",A
```

To specify a save on drive 1, the command is:

```
SAVE"MPGCALC1/BAS:1",A
```

LOAD and RUN procedures for files saved with the ",A" option are exactly the same as those for programs saved in compressed format. (Don't make the mistake of thinking that you use ",A" with a LOAD or RUN command.) The system automatically senses what kind of file it is loading.

Why save in ASCII? There are several reasons:

1. During program development, BASIC's "MERGE" command lets you combine a program file on disk with a program in memory. It requires the program file on disk to be in ASCII format.

2. You may want to use a word processing program such as Scripsit to edit your program or to make formatted listings. Most TRS-80 word processor programs can load BASIC programs saved in ASCII.

3. A program saved in ASCII is actually a sequential file. On occasion, you may wish to write BASIC programs that read other BASIC programs as data. These programs may do things such as making automatic modifications or formatted listings. If you are ambitious, you can even write programs that automatically write other programs based on parameters you supply. The standard routines that a "program generator" may use can be stored in ASCII files.

4. You may wish to use BASIC to create and SAVE "programs" that are never meant to be run. The built-in editing capabilites that you have with BASIC can be quite convenient for getting data into a disk file so that it can be used by other programs. As an example, you might want to use a BASIC program file as a check register, with the line numbers corresponding to check numbers. If you save it in ASCII, you can write another program that adds up the checks and balances the register for you.

5. The ability to save in ASCII can be valuable as a recovery technique in certain situations. In rare cases, equipment malfunctions can turn a program file into garbage. The line numbers may be jumbled and out of order. Saving the damaged program in ASCII and reloading it will usually make it possible for you to make the required corrections because a LOAD for an ASCII program file automatically arranges the line numbers in the proper sequence.

6. The DOS command, APPEND, can be used to combine program files if they have been saved in ASCII format. See your disk system owner's manual.

7. With Model III TRSDOS, the DOS command, LIST, can be used to display or to make printouts of your programs if they've been saved in ASCII. As an example, suppose you've saved "MPGCALC1/BAS" with the ",A" option. From TRSDOS Ready you can type:

```
LIST MPGCALC1/BAS (PRT,ASCII)
```

. . . to get a hard-copy listing on your printer. Or, you can type:

```
LIST MPGCALC1/BAS (SLOW,ASCII)
```

. . . to roll the program listing before you on your display.

Model I TRSDOS has the LIST and PRINT commands.

```
LIST MPGCALC1/BAS
```

. . . scrolls a listing of the program on your display.

```
PRINT MPGCALC1/BAS
```

. . . makes a printout on your line printer.

Error Conditions During LOAD and RUN

There are several opportunities for things to go wrong with a LOAD or RUN command. The most likely are listed below:

Code	Message
65	Bad File Name
70	File Access Denied
54	File Not Found
67	Direct Statement in File
7	Out of Memory
57	Disk I/O Error

We'll get to a discussion of the error code numbers later in this section, but first we'll cover each message in detail.

"Bad File Name" can be caused by a simple typing error or a misunderstanding of what constitutes a valid name. "File Access Denied" means that the program was saved with a password, or it has been given one with the ATTRIB command while in DOS Ready mode. You'll need the password to LOAD or RUN the program. "File Not Found" means that the program name as specified was not found on disk. These three error conditions do not erase the program currently in memory, so if they occur during the execution of a program you can design steps into your program that let the operator try again.

The "Direct Statement in File" error most often occurs when you inadvertently specify a file to LOAD or RUN that is not a BASIC program. That is, the system determines that it is not in the standard compressed format so it attempts to load it as an ASCII program file. When it finds a line that doesn't begin with a line number the error occurs.

As an example, the Model III TRSDOS diskette contains a machine-language program called MEMTEST/CMD. If, while in BASIC you enter,

```
LOAD "MEMTEST/CMD"
```

... you get the direct statement error. Or, perhaps you have a data file containing employee names. An attempt to load it will also be likely to generate the error.

Another situation in which the direct statement error can occur is when you save a program in ASCII and attempt to reload it. If it has a line that is more than 240 characters long, the error will occur. The system has a limitation in that it can only process 240 characters per line during an ASCII LOAD, RUN or MERGE. The problem usually doesn't occur because you can only type 240 characters when keying in a program line. The danger is that when editing a line, BASIC lets you get away with extending it beyond 240. After a "Direct Statement in File" error, memory will contain only those program lines up to the offending one.

An "Out of Memory" error can also occur during a LOAD or RUN. It means that the program on disk is too large for the available capacity in memory. There are three main causes for it:

- The computer the program was written on had more memory than yours does. Perhaps it was designed for a 48K TRS-80, and you only have 32K. Perhaps the program was written with a different disk operating system from the one you are using. Some operating systems leave more memory space available for BASIC programs than others.

- You specified the wrong numbers in response to "How Many Files" and/or "Memory Size", so that not enough memory is available.
- You've got too much string storage space reserved. Perhaps the program you've been using before the LOAD or RUN had a statement that cleared an unusually large amount of memory, for instance, "CLEAR 20000". You can solve the problem by typing "CLEAR 50" or "CLEAR 0" before retrying your LOAD or RUN command.
- With the Model I, the memory within the expansion interface can sometimes fail to turn ON. Often, turning the system OFF and then ON again will solve the problem. If you try this, you will need to re-do your entire power-up procedure. If you determine that this is the problem, and it persists, you should have your expansion interface serviced.

A "Disk I/O" error can be more serious. It indicates that the system is having trouble reading the diskette. First check to see that you've got your disk inserted properly. Another thing to try is to remove and reinsert the diskette. Often, just opening and closing the drive door will help ensure that the diskette is seated properly.

If these remedies don't get you beyond the disk I/O error after a few attempts the next best thing to do is to LOAD or RUN from a backup copy of your program on another diskette. If that also fails, your problem may be a dirty or misaligned disk drive. Try a different drive if you've got one.

In the event that you don't have a good backup copy of your program, you are left with the prospect of retyping and saving it, or of trying more advanced recovery techniques.

Using the Error Codes

Unless you take programming steps to avoid it, upon encountering an error condition, BASIC will display the error message, and further execution of the program (if any) will be stopped, returning you to ready mode. BASIC has an error code that is associated with each possible error condition. The error code numbers are useful when you want your program to take a specific action, depending on what the error is. From ready mode in BASIC, let's try a few things to illustrate.

First, let's try using an invalid file name. Enter the following direct command:

```
LOAD"123"
```

The display shows the message "BAD FILE NAME", since all file names must begin with a letter from A to Z. Now from Ready, type:

```
PRINT ERR
```

The display shows 128. This is the "raw" error code. TRS-80 Models I and III require that you convert the raw code to get a number that you can look up in the manuals. To do so, divide by 2 and add 1. The command should have been,

```
PRINT ERR/2+1
```

. . . giving you the true error code, 65. The appendix of your disk system owner's manual gives you a complete list of the disk error codes reported by BASIC, ranging from 61 to 70. Your *Operation and BASIC Language Reference Manual* for the Model III, and the *Level II BASIC Reference Manual* for the Model I give you the codes for non-disk errors.

Don't confuse BASIC's error codes with the TRSDOS error codes listed elswhere in your manual. The TRSDOS error codes, 0 through 41, are only useful when working directly with the disk operating system. Though they are much more specific, they don't correspond to the codes used in Disk BASIC.

When programming in a non-disk environment, error handling is not too important once you've gotten your program written and debugged. The errors you'll encounter will normally result from programming mistakes, bad typing, or situations that you did not anticipate, such as a division by zero. Disk programming is different. It is normal to expect error conditions that may be caused by the operator, like failing to insert a diskette properly, and error conditions that may result from a failure of the diskettes or drives.

BASIC provides a method for handling error conditions that occur during the execution of a program with the ON ERROR GOTO statement. Since RUN, and sometimes LOAD, are used within programs, we'll talk about this valuable statement now.

To use ON ERROR GOTO, simply assign a beginning line number for your error handling routine. The program line,

```
1Ø   ON ERROR GOTO 6ØØØØ
```

. . . tells the system that, if an error occurs, no matter what it is, to go to line 60000 for the program logic that handles it. You can have logic at line 60000 that inquires into the value of ERR so appropriate action may be taken. You can also use the value of ERL if your program routine needs to know the line number in which the error occurred.

The ON ERROR GOTO statement remains active until another ON ERROR GOTO statement changes the line number at which errors will be handled. ON ERROR GOTO 0 cancels reference to an error handling line so that BASIC's normal error processing messages will be put back into effect. A common practice is to put an ON ERROR GOTO statement just before each disk accessing command. After the command is successfully executed, or when the types of errors anticipated by the program have been exhausted, an ON ERROR GOTO 0 is used to cancel it. As an example, our mainline program logic might contain lines 10 and 20:

```
1Ø  LINEINPUT "WHAT PROGRAM DO YOU WANT TO RUN? ";FS$
2Ø  ONERRORGOTO6ØØØØ:RUN FS$
```

Now, if the system is unsuccessful in loading the program named FS$, most likely because of a bad file name or an inability to find it on disk, the program will be diverted to line 60000. At line 60000 you can handle the error condition:

```
6ØØØØ IFERR/2+1=65THENPRINT "BAD FILE NAME, PLEASE RETYPE IT!":RE
SUME1Ø
6ØØØ1 IFERR/2+1=54THENPRINT "NOT FOUND! ENTER <1> TO TRY AGAIN OR
 <2> TO RETYPE THE NAME:";:LINEINPUTA$:IFA$= "1"THENRESUME2ØELSER
ESUME1Ø
6ØØØ2 ONERRORGOTOØ
```

Notice that the RESUME statement was used in lines 60000 and 60001 to send execution back to the program line desired, depending on the error condition encountered. In this case, we only anticipated the two most likely errors, "bad file name" and "file not found". If any other type of error occurs, line 60002 will be invoked. It causes the system to revert back to normal error handling. For example, let's take the case in which the operator specifies a program name that requires a password without typing the password correctly.

The display will show the error message and line number, and the system will return to READY mode:

```
File access DENIED in 2Ø
READY
>
```

Besides using RESUME to send execution back to a line number, as in RESUME 10 or RESUME 20, you can use other variants of the resume statement:

- RESUME without a line number, causes the program to resume at the start of the line in which the error occurred.
- RESUME NEXT will cause the program to resume with the statement following the one in which the error occurred.

Automatic error handling with LOAD and RUN is not possible for some types of errors. One example is a disk I/O error encountered after a new program has begun to load. The partially loaded new program may overlay the memory occupied by the error handling logic in the program that executed the command. For other disk commands, though, all opportunities for error conditions can be handled. We will be discussing them as they arise.

Variables in BASIC

The first and most simple application for sequential file programming is saving variables on disk. Your BASIC reference manual gives a complete discussion of variables and their use in programming. For our purposes now we will highlight and review the important points about using variables in BASIC.

In BASIC, variables are named with one or two characters. The first must be a letter from A to Z. The second may be a letter or a digit from 0 to 9. Additional characters may follow the first two if you want more descriptive variable names, but the first two characters must be unique to distinguish one variable from another.

Each variable must be one of 4 different types. The type is usually specified by a symbol that follows the variable name:

Figure 3.6 — *Variable Types*

Symbol	Type	Data Stored
%	Integer	Whole numbers from -32768 to 32767
!	Single Precision	Decimal numbers up to 6 significant digits with exponents from -38 to +38
#	Double Precision	Decimal numbers up to 16 significant digits with exponents from -38 to +38
$	String	A series of up to 255 characters, each of which occupies one byte of memory and contains a code from Ø to 255. The codes may individually represent letters, digits, or symbols. Taken in combination, they may contain compressed representions of numbers and other types of data.

A$, AM$, A%, G!, G2!, ZZ#, LI%, TAX% and PRICE! are all examples of unique, valid variable names. We use variable names in BASIC as a way of telling the system that we want it to hold a given piece of information for us temporarily. When we want to use that information later within a program, we can ask for it with the name we assigned. When we want to change the information stored by a variable, we can do so at will.

When we want to do computations in a program, we can use variable names as the operands. During execution of the program, BASIC will automatically substitute the values our variables contain.

The "%", "!" and "$" symbols are not always necessary after our variable names. If we omit them, BASIC will assume all variables to be single precision, just as if we had put a "!" symbol after each. We can also control what BASIC will assume with the DEFINT, DEFSNG, DEFDBL, or DEFSTR statements. If we use the statement

```
DEFINT A-Z
```

... BASIC will assume that all variables in the program are of the integer type, unless we use a symbol to explicitly state that a particular variable is to be handled differently, or until we use a DEFSNG, DEFDBL or DEFSTR statement to change it. DEFSNG defines a particular variable or a list of variables as single precision. DEFDBL does the same for double precision. DEFSTR defines the variables specified as strings. To show an example that combines all the options, the statement:

```
DEFINT A-M : DEFSTR F : DEFDBL D,Z
```

. . . makes BASIC assume that all variables starting with the letters A through M are integers. Then it defines all variables beginning with the letter F as strings. Finally, it defines variables beginning with D and Z as double precision. All other variables will be single precision. Notice that the DEFDBL D changes the specification for variables starting with D from integer to double precision.

Variables in BASIC can be classified another way: simple variables and array variables. Up until now, we've been looking at simple variables. Array variables are used to store lists and tables in memory. This is done with one or more subscripts, enclosed in parentheses, following the variable name and type symbol.

As an example, A#(0), A#(1), A#(2) and A#(3) might be used as a table to store 4 double precision amounts. They all have the same name, but a different subscript. AD$(1), AD$(2), AD$(3) and AD$(4) might be used to store the four lines of a customer's address.

Arrays are powerful ways of storing data because we can design program statements that perform repetitive operations on their elements. To print a mailing label with the data in the AD$ array, we can say:

```
FOR X = 1 TO 4
LPRINT AD$(X)
NEXT
```

. . . rather than saying:

```
LPRINT AD$(1)
LPRINT AD$(2)
LPRINT AD$(3)
LPRINT AD$(4)
```

The shortcut method of using a variable as a subscript becomes more valuable with very large arrays. When arrays contain more than 11 elements, 0 through 10, we must dimension them so BASIC can set aside enough memory. This is done with the DIM statement:

```
DIM AM#(1ØØ)
```

. . . dimensions the AM# array so that it will hold 101 double precision numbers.

```
DIM NA$(32)
```

. . . dimensions the NA$ array so that it will hold 33 strings.

Arrays may also be doubly or triply dimensioned. The statement:

```
DIM I%(5,4)
```

. . . sets up an integer array, I%, so it will hold data as if it were in a table of 6 rows and 5 columns (or 5 rows and 6 columns, depending on how you wish to process it). The following program statements show how the contents of the I% array could be displayed on the screen in row and column format:

```
FOR ROW=Ø TO 5
FOR COLUMN=Ø TO 4
PRINT I%(ROW,COLUMN),
NEXT COLUMN
PRINT
NEXT ROW
```

We could also use the following logic to display the entire contents of the array as one long list:

```
FOR X=Ø TO 5
FOR Y=Ø TO 4
PRINT I%(X,Y)
NEXT Y
NEXT X
```

Similarly, a triple dimension array can be used. Suppose we sell 8 styles of shirts, each of which comes in 4 sizes and 2 colors. To keep a count of each shirt in each size and color, we could dimension an integer array as follows:

```
DIM CO%(7,3,1)
```

To access the inventory count for style 3, size 1 and color 2 we could say:

```
PRINT CO%(2,Ø,1)
```

Notice that we used the number, minus 1, for each subscript. It's the old problem of reconciling the real world, where we usually start counting with 1, and the computer world, where we often start with 0. The first element in any array is always 0. We can, if we wish, choose to ignore the zero element, and use the 1 element as our first, but memory will be wasted.

With most versions of BASIC available for TRS-80 computers, when a program is RUN, no variables are active. As variables are introduced into the program, they are added to a variable list that BASIC maintains in memory. We use program statements to load variables with the values we want. The most common way that variables get their values is by using program statements that put a variable name to the left of an equals sign, and an expression, or constant, to the right, as in the following examples:

```
TI$="INVENTORY STATUS REPORT"
A1%=-12345
C!=1.4432
B#=A#+C#/2
CD%=INSTR("YN",A$)
AM%=BM%
```

A statement that uses an equal sign to load a variable is called an assignment statement. The next most common way to load variables is with input statements. They let the operator supply values during execution of a program. This is done with the INPUT and LINEINPUT statements:

```
10 INPUT"WHAT IS YOUR LAST NAME";LN$
10 INPUT J%
10 LINEINPUT"PRESS <ENTER> TO CONTINUE...";A$
10 LINEINPUT"TYPE THE NAME OF THE PROGRAM YOU WANT: ";FD$
10 LINEINPUT Z$
```

I arbitrarily used line 10 for each example. The line numbers were included in the examples to emphasize that INPUT and LINEINPUT can only be used within BASIC programs. It is illegal to use them as direct commands from ready mode.

INPUT may be used with any variable type. One advantage of "quick and dirty" programs is that INPUT statements may allow the entry of more than one variable at a time, as in the following example:

```
10 INPUT"ENTER LAST NAME, FIRST NAME, AND YOUR AGE";L$,F$,AG%
```

The operator may type the three data items, pressing ENTER after each, or they may all be typed at once, separated by a comma, and terminated after the final item with <ENTER>. Since the comma is used as a separator in INPUT statements, it cannot be used within the data unless the data entered by the operator is enclosed in quotes. The operator must also use quote marks if a string to be input contains leading blanks or colons. Otherwise, the INPUT statement will edit out all blanks to the left of the first character, and all data to the right of, and including, the first colon.

If the operator attempts to enter something other than a number when an INPUT statement specifies an integer, single, or double precision variable, the system will display "?REDO" to enforce a numeric entry. If the operator enters too many data items, the system will display "?EXTRA IGNORED".

LINEINPUT or LINE INPUT is an enhancement that disk BASIC provides to extend the capabilities provided by BASIC in the ROM of the Models I and III. LINEINPUT may be used to assign values to strings only, and unlike INPUT, it can accept only one entry at a time. In sophisticated programs it is more useful than INPUT because it gives the programmer more control over the video display and it eliminates pre-editing. You have more control over your display because no "?" prompt is printed and no "EXTRA IGNORED" or "REDO" messages are shown.

Since pre-editing is eliminated, your string will contain exactly what the operator typed, including leading blanks, quotes, commas, and colons. This can be important, for instance, when you want the operator to type a file name and drive number such as MPGCALC1/BAS:1.

We said that LINEINPUT must be used with string variables. We can, however, use it to load numeric variables if we use the VAL function. We LINEINPUT the number into a string and take the VAL of the string. This two-step process is valuable if you want to allow more than one data type in a single statement. Here's an example of a short program that will add together as many numbers as you care to give it. When you are finished, enter "E" to print the final total:

```
1Ø LINEINPUT"ENTER THE AMOUNT, OR PRESS <E> TO END...";A$
11 IF A$<>"E" THEN A!=VAL(A$) : T!=T!+A! : GOTO1Ø
12 PRINT "TOTAL: ";T!
```

Before you learn sequential file programming, it is important to get familiar with INPUT and LINEINPUT. The disk commands that retrieve data from a sequential file correspond very closely to their keyboard entry counterparts.

DATA statements are a third way to load values into variables. They let you put a list of data items anywhere within your program. As we'll see, data statements are very analogous to sequential files in that BASIC must read each item from first to last. They also are similar to INPUT statements in respect to the way that BASIC strips leading blanks and uses commas as separators, unless quotes surround the data item.

Here is an example of how a data statement might be used to load the months of the year into an array:

```
1Ø DATA JANUARY,FEBRUARY,MARCH,APRIL,MAY,JUNE
11 DATA JULY,AUGUST,SEPTEMBER,OCTOBER,NOVEMBER,DECEMBER
2Ø DIM MY$(12)
3Ø FOR X=1TO12
4Ø READ MY$(X)
5Ø NEXT
6Ø PRINT"TODAY IS ";MY$(VAL(TIME$));" ";MID$(TIME$,4,2)
```

The function of this sample program is to display the month name and day, based on the contents of TIME$.

If the date contained in TIME$ is 05/15/82 the program displays:

```
TODAY IS MAY 15
```

Notice that line 40 is executed 12 times when this program is RUN. Each time, BASIC moves its pointer to the next item in the data statement, and loads the corresponding element of the MY$ array. This is much more efficient than having 12 separate assignment statements:

```
MY$(1)="JANUARY" : MY$(2)="FEBRUARY" : .   .   . : MY$(12)="DECEMBER"
```

Also, notice that the data statements are on two different lines, 10 and 11. Actually, data statements may be anywhere in your program and may be spread over as many lines as you wish. In this case, you could have put all the data elements in line 10 or in 12 separate lines.

One error that programmers sometimes make is to put a comma at the end of a data statement line. Notice in our sample that no comma terminates line 10. If there had been one, our 6th month would have been loaded as a null string, and months 7 through 12 would have been wrong.

An important technique that is often used with data statements is to combine different types of information. As an illustration, suppose we wanted to load two arrays. The MY$ array is to contain the months of the year, and the ND% array is to contain the number of days in each month. The program could be modified as follows:

```
1Ø DATA JANUARY,31,FEBRUARY,28,MARCH,31,APRIL,3Ø,MAY,31,JUNE,3Ø
11 DATA JULY,31,AUGUST,31,SEPTEMBER,3Ø,OCTOBER,31,NOVEMBER,3Ø,DE
CEMBER,31
2Ø DIM MY$(12),ND%(12)
3Ø FOR X=1TO12
4Ø READ MY$(X),ND%(X)
5Ø NEXT
6Ø PRINT"TODAY IS ";MY$(VAL(TIME$));" ";MID$(TIME$,4,2)
61 PRINT"THERE ARE";ND%(VAL(TIME$));" DAYS IN THIS MONTH"
```

Now the READ statement in line 40 loads two variables at a time. If you use a list of variable names to be loaded, a READ statement can load any number of variables as long as your list fits in a program line and the data statements are coordinated properly.

Our data statement illustrations thus far have assumed that the data was to be read only once during the execution of a program, that it was to be loaded into one or more arrays, and that we knew exactly how many data items were to be read. There are some cases where it is not convenient or efficient to make all these assumptions. To handle those cases we will need to know a little more about DATA statements.

Whenever you want your BASIC program to go back to start reading from the first data item again you may use the RESTORE statement. It, in effect, resets a pointer, so the next READ statement will return the first data item.

If your program attempts a DATA statement READ and no more items remain in the data list, an "OUT OF DATA" error (code 4) will occur. If line 30 in the example had requested 13 repetitions, the error would have occurred after the 12th, given the data list shown.

We can use the "OUT OF DATA" error to our advantage in cases where we don't wish to use arrays to store the data items, or in situations where, due to frequent modifications of the program, it is inconvenient to keep track of how many data items you have. A special subroutine can be helpful. I call this one the "Read-Data" subroutine. It occupies lines 900 through 904. Each time you want to read one data item, or a set of them, you can call the subroutine. Instead of aborting the program with an out of data error, it returns a value in the variable, ED%. ED% is 0 if the subroutine was successful in getting the data. ED% is returned as –1 if an out of data error occurred. To use it you must replace the list of variables in line 902 with the variables you want to use to receive the data.

```
9ØØ 'READ-DATA SUBROUTINE
9Ø1 ED%=Ø:ONERRORGOTO9Ø4
9Ø2 READ A1,A2:'REPLACE WITH THE READ STATEMENT YOU REQUIRE
9Ø3 ONERRORGOTOØ:RETURN
9Ø4 IFERR/2+1=4THENED%=-1:RESUME9Ø3ELSE9Ø3
```

You can construct many useful programs that implement the READ-DATA subroutine. I recommend that you put your program logic in lines 0 through 899 and start your data statements at line 1001. Here's a program that illustrates how the subroutine might be used in two ways. First, it displays the data list. Second, it requests operator entries from the keyboard and searches the data list. It does both of these tasks without regard to how long the data list is, and without using arrays. The data set used in our example is a list of account numbers and descriptions for the accounts, as might be found in a general ledger, budgeting or accounts payable system. AC$ is used to store the most recent account number read from the data statements. NA$ stores the account name that corresponds to the account number.

Figure 3.7 — *DATASUB/DEM*

```
Ø 'DATASUB/DEM
1ØØ 'DISPLAY DATA DEMONSTRATION ******************************
11Ø CLS:RESTORE
13Ø GOSUB9ØØ:IFED%THEN2ØØELSEPRINTAC$,NA$:GOTO13Ø

2ØØ 'SEARCH DATA DEMONSTRATION *******************************
21Ø RESTORE:LINEINPUT"WHAT ACCOUNT NUMBER DO YOU WANT? ";A$
22Ø GOSUB9ØØ:IFED%THENPRINT"NOT FOUND...";:GOTO2ØØ
23Ø IFAC$<>A$THEN22ØELSEPRINT"FOUND:":PRINTAC$,NA$:GOTO2ØØ

9ØØ 'READ-DATA SUBROUTINE ************************************
9Ø1 ED%=Ø:ONERRORGOTO9Ø4
9Ø2 READ AC$,NA$
9Ø3 ONERRORGOTOØ:RETURN
9Ø4 IFERR/2+1=4THENED%=-1:RESUME9Ø3ELSE9Ø3

1ØØØ'DATA LIST ***********************************************
1ØØ1 DATA 511.ØØ,TELEPHONE
1ØØ2 DATA 512.ØØ,GAS
1ØØ3 DATA 513.ØØ,ELECTRIC
1ØØ4 DATA 52Ø.ØØ,MEDICAL
1ØØ5 DATA 521.ØØ,REPAIRS
```

Notice that the account numbers "511.00","512.00" and so forth, are handled as strings. You can easily handle the same data items as as numbers with the following changes:

- Change all references to "AC$" to "AC!".
- Change "LINEINPUT" to "INPUT" in line 210.
- Change "A$" to "A!" in lines 210 and 230.

The point is that when a number is encountered in a data statement the READ command doesn't care whether you assign it to a string variable or a numeric variable. This is a useful characteristic of data statements, and as we shall see, a similar principle applies to input from sequential files.

The data is independent of the operating logic. Once you've designed the arrangement of your data, your program is free to do with it what you please, and you can use the variable names you wish.

You must be more careful when you use numeric variables. A READ statement that attempts to load a numeric variable with a series of characters that can't be interpreted as a number will result in a syntax error. Also, you'll run into problems if a number is encountered which is larger than permitted by the variable type you've chosen. If you attempt to READ a number less than -32768 or greater than 32767 into an integer variable, you will get an overflow error.

We've used DATASUB/DEM to show the principles of data statements, READ, and RESTORE. Otherwise, DATASUB/DEM really doesn't perform any useful purpose, but it is a handy program to have available on disk. By changing the data statements that start at line 1000, and re-writing the program lines before line 900, you can use it for many purposes.

BASIC's editing capabilities allow you to add, change and delete program lines, thus giving you a convenient way to get information into your computer. When used with LOAD and SAVE for a program that contains DATA statements, you have a "primitive" method for storing many kinds of information on disk.

We'll be using the DATASUB/DEM program as a "shell" program in several ways later in the book. In its role as a "shell" program, we will be changing the data statements and re-writing the program logic.

Why Data Files?

Now that we've discussed a few things about variables and data storage in BASIC, we can get a better appreciation for why we want to use data files in the first place. Data file programming adds a measure of complexity to your work, and for many applications you can manage without them. So, before we dive into the techniques of programming data files, let's review the reasons for them.

The first reason for data files is "permanence." All variables are normally cleared when you go from one program to another in BASIC or when you shut down the computer for the day. In the absence of data files, if you want to go back to a previously run program, you have to re-enter all of the information, even if some of it is the same information you used before. Programs have to rely upon what is input from the keyboard. Each time, you have opportunities for operator errors, and, of course, you waste time making redundant entries.

DATA statements within a BASIC program can provide permanence, assuming you use a SAVE command to put the program on disk. Indeed, DATA statements can be a useful substitute for data files, but you soon come upon limitations.

One disadvantage is that additions, deletions or changes to DATA statements cannot be done automatically. To expect an unsophisticated operator to do it accurately might be asking too much. There are many opportunities for errors.

On the other hand, you can write programs with data files that prompt the operator through each entry and verify validity. They make it more convenient to add, delete, change and re-sequence the data that is used in an application.

Another reason for data files is that memory capacity is limited. Even with the smallest diskette, the storage capacity on disk is more than 5 times the available space in memory when a BASIC program is running. The external and removable data storage provided by disk files frees precious memory for the complex logic that your program may require. Your only limitation is the capacity of the drives you select and the number of diskettes you use.

The final reason is "program/data independence." This principle refers to the idea of storing data separately from the programs that operate on it. Program/data independence is important in two ways. First, it allows you to write one program or several different programs that use the same data files. You can break one big program into smaller, more easily managed, modules. The data can be more easily managed because changes need only be made in one place to keep everything up to date. Secondly, program/data independence makes it possible to write and sell standard programs. Each user of the same program can develop his own data files. No change to the program itself is necessary to "customize" the system for each owner.

Secrets of Sequential Files

We will begin our study of file programming in BASIC with sequential files. Many people find that sequential file programming is easy to learn because there are very few steps involved. Once you learn a few commands, you can do many things with sequential files. As you get more familiar with them, you'll be able to appreciate their value, but you'll also begin to get a realization of their limitations. Sequential files are just one tool in your bag of tricks. This chapter will show you how and when to use them.

Saving Variables On Disk

In the last chapter, we talked about the different types of variables that you may use in a BASIC program. Without the capability for using external files, the data that's stored in those variables has to come from keyboard input statements or data statements that are built into the program itself.

Programs that just do calculations rarely require data files. When you sit down at the computer, you type in the information requested, and the answers are displayed or printed for you. Take, for example, a program that computes an amortization schedule for a mortgage loan. You just type in the principal, the interest rate, the number of periods and the monthly payment. The computer prints out the results of its calculations, and you're done. Most game programs fall into the same category. You turn ON the computer, play the game, and turn OFF the computer when you're done. No information needs to be carried forward from one session to the next.

Let's consider a class of program that's a little more complex:

- A calculation program that is used periodically, where you want to carry forward the results from one session to the next. The mileage calculator in the previous chapter is a good example. The ending odometer reading for one session is the beginning odometer reading for the next. That information could be stored on disk.

- A game program in which you wish to keep a list of the highest scores attained, or in which you want to be able to store all data for a game in progress so it can be resumed at a later date.

- A calculation program that requires a large number of data entries. You might want to save the input data on disk so when you come back later to make a change or two for a recalculation, you don't have to re-enter everything. Examples are tax computation and tax planning programs, construction estimating, and investment analysis.

- A program of any type that is written to satisfy a large number of diverse users. To eliminate re-answering questions such as "What's your company name?", "Does your system have a line printer?" and "What options do you want?" each time the program is started, it may be desirable to get this information once, store it on disk and load it automatically at the beginning of each session.

- A series of programs that are chained together. When you go from one program to another, you may want to pass along the contents of certain BASIC variables. Temporarily storing them on disk is a way to get around the problem of variable clearing when a RUN statement is executed.

All of these situations have something in common. A BASIC program has loaded one or more variables with data and you would like to recover that data for use in a subsequent session. Unfortunately, each time you RUN a program, in effect, all numeric variables are cleared to 0 and all string variables are erased. You can SAVE programs, but there's no automatic, one-step way to "save" the contents of some or all of our variables. You have to use data file programming. Sequential file techniques provide you with the most easily programmed way to "save" the contents of the variables you want to preserve for the next run.

Sequential file programming is quite similar to printing. One way to save variables for future use is to make a hard-copy printout on a line printer. Suppose, for example, that your program has information stored in SS$, ND%, NA$, ST$ and TI#. You would like to remember the contents of those variables for the next run. To save the data on paper you can say:

```
LPRINT SS$
LPRINT ND%
LPRINT NA$
LPRINT ST$
LPRINT TI#
```

Perhaps your paper shows:

```
341-34-3432
 2
BILL JACKSON
CA
 23252.35
```

As an alternative, and more simply, you can issue the command:

```
LPRINT SS$, ND%, NA$, ST$, TI#
```

In that case your sheet of paper might show:

```
341-34-3432     2              BILL JACKSON   CA
23252.35
```

You also have the option of eliminating the spaces between the data you LPRINT:

```
LPRINT SS$; ND%; NA$; ST$; TI#
```

That would give the following printout:

```
341-34-3432 2BILL JACKSONCA 23252.35
```

With each printout method (except perhaps the last) when you start your next session at the computer your input job is simplified because all the information you'll need is printed on paper.

A sequential file works in much the same way, but it stores the contents of BASIC variables on disk rather than paper. A line printer starts at the top of a page, works its way down to the bottom, and goes to a new page if necessary. It is a sequential output device. In the case of a sequential file, BASIC starts accepting the data "printed" to it. When 256 bytes have been accepted, it turns on the disk drive and records that data into the first 256-byte sector of the file. As your program continues sending data to the file, BASIC continues accumulating and writing the data to subsequent disk file records in 256-byte batches. When you tell BASIC you have finished giving it data to store in the file, it writes anything it has received but not yet recorded, and you've got a sequential file!

You should be familiar with the PRINT, LPRINT, INPUT and LINEINPUT statements for the video display, line printer and keyboard. Sequential files have statements that work almost exactly the same, but for disk files. And, like their counterparts, you've got some flexibility in the way that the data is separated or spaced.

The process of saving the contents of your variables on disk has 3 steps:

1. Open a sequential file for output.
2. Print to the file the contents of each variable you want stored.
3. Close the file.

The process of reading back the contents of the variables you saved on disk also has 3 steps:

1. Open the sequential file for input.
2. Input the contents of each variable you stored.
3. Close the file.

We'll look at the commands for each, then we'll look at a few program examples.

OPEN"O"

The first thing you need to do when you want to store information in a sequential file is to open it. Opening a sequential file for output is like putting a piece of paper in your line printer (or electric typewriter), aligning it to the top of page and turning it ON. There are a few things you have to decide when you program an OPEN statement.

In the case of a sequential file, you have to specify which of up to 15 file buffers you want the program to use, just as you might need to decide which printer to use if you had more than one connected to your computer. This is where the response supplied for the "How many files?" question upon entering BASIC affects you. We mentioned that BASIC accumulates data that is being sent to a sequential file, and doesn't actually record the data it's received onto the disk until your program has given it 256 bytes, or until you've told it you are done. File buffers are the waiting areas in memory where this data is accumulated. If, upon entering BASIC, you specify that you will need 3 file buffers, you will have three waiting areas, numbered 1, 2 and 3. In such a case, your program is capable of working with three different files at once. As you get into more complex operations, you'll see the reasons for multiple files. For now, assume that you're only using one file, in buffer 1.

You also have to decide what file name to use for storing the data, and what disk drive to store it on. This is like deciding what kind of paper to insert in your printer and deciding where to keep it when the printout is complete.

In a nutshell, you only need to decide two things when opening a file for sequential output:

1. The file buffer number to use.
2. The file name to use. Optionally, you can specify a disk drive number.

To open a sequential output file named "STORAGE/SEQ" on drive 0 using file buffer 1, the command is:

```
OPEN"O",1,"STORAGE/SEQ:Ø"
```

If you wish, you can use variables to specify the file buffer number and/or the file name and drive. As an example, to open a sequential file for output whose name is stored in the string FO$, and using a file buffer specified by the integer variable BF%, the statement is:

```
OPEN"O",BF%,FO$
```

The "O" in the open statement can also be replaced by a string variable name. The "O" is the code for the file mode. In this case, you are opening a sequential file in "output mode". That is, you want data to go "out" of the computer and onto a diskette. Later you'll see that (for sequential files) "I" is "input mode" and "E" is "extend mode". If you use a variable, such as MD$, you can use the same program line to handle all open commands. It might have the following statement:

```
OPEN MD$,BF%,FS$
```

The open statement can be put anywhere in a program. The only requirement you need to be aware of is that another file must not already be open in the same file buffer. A file buffer can be used for only one file at a time!

The file name and drive specification is identified and handled the same way as a BASIC program file spec is. For that matter, your constraints in selecting a name and drive for a sequential file are the same as for any file using the TRSDOS disk operating system. To make it obvious to you what you're looking at when you examine a directory, you might want to use an optional identifer like "/DAT" to remind yourself that the file name you see listed is a data file. Or, as some suggest, you might use "/TXT", meaning text file. I prefer to use "/SEQ" to identify a file as being of the sequential type. Of course, you don't need a file name extension at all.

To conclude, the OPEN"O" statement does the following things:

- If not already present, it records the file name you specify in the directory of the disk in the drive you specify.
- It assigns a particular area in memory as a 256-byte buffer or "waiting area" for data that will be sent to the disk file.
- It presets to 0 an end-of-file pointer in memory. That pointer is a number which (like a book mark) keeps track of how many records of data have been sent to disk, and how many bytes are still in the buffer, awaiting storage on disk. As an operator or programmer you are usually unaware of the status of the end-of-file pointer while you're outputting data, since it is maintained invisibly and automatically by the system.

PRINT#

Now that you've opened your sequential file for output, what do you do with it? You do the same thing we do with a printer after it's turned ON — send it data.

The command used to send data to a sequential file is PRINT#. For now we'll just talk about numeric data from integer, single precision and double precision variables. Here are some examples:

```
PRINT#1,A!
```

. . . sends the contents of A! to the sequential file that has been opened for output as file 1.

```
PRINT#2,A%;B%;C#
```

. . . sends the contents of all the variables listed to the sequential file that has been opened for output as file 2. Notice that the numeric variable names are separated with semicolons. Each number, when recorded to disk, will be followed by a blank and preceded by a "–" or blank for the sign, just as if it is being printed on the display. You can also use commas between the variable names, but that separates the data into 16-byte fields, just as a comma following a print statement separates displayed data into 16-byte fields on the video display. There's rarely any reason to use a comma in this way for sequential file output.

The same variables can be sent to an output file with separate PRINT# statements:

```
PRINT#2,A% : PRINT#2,B% : PRINT#2,C#
```

You can, if you wish, use constants and expressions as well as variables:

```
PRINT#1, 1ØØØ ; A%*2 ; A#+1
```

You might also want to store the contents of an array in a sequential file. This use for sequential files is quite common. Here's how you might store the contents of the first 10 elements of the A# array:

```
FOR X=1 TO 1Ø
PRINT#1,A#(X)
NEXT
```

It's important to note that just the contents of the variables (or the results of BASIC's evaluation of the expressions) go into the file. The file does not store the names of the variables or the formulas of the expressions. In sequential file programming the watchword is "sequence." If you send data out to the disk file in a specific sequence, you'll get it back later in that same sequence. If C# is the third variable you wrote to the sequential file, when you read it back, the third item of data you get will be whatever the contents of C# were when you outputted it. If you stored ZA% as the 5th item in the file, your input program will have to read past the first 4 items to get to it.

If you've opened a sequential file for output, and a file having the same name was already on the disk when you started, all the data that is written to the file with your PRINT# statements is being recorded over the previous contents of that file. Once you CLOSE your output file, as explained next, the contents of your previous file are, for all practical purposes, gone forever.

CLOSE

Now that you've outputted data to a sequential file, you have to take a final step. You have to close the file. Assuming you have opened and printed data to file 1, the command is:

```
CLOSE 1
```

If you are using BF% to remember what file buffer number is being used, you can also say:

```
CLOSE BF%
```

If you want to close any and all files that are currently open, simply say:

```
CLOSE
```

There are a few other ways that you can specify the files to be closed with a CLOSE command. They are listed in your *Disk System Owner's Manual.*

The CLOSE command performs some important functions where sequential output files are concerned:

- CLOSE tells the disk operating system that there is no more data to be outputted. In response, the DOS writes any remaining data waiting in the disk buffer onto the diskette.
- CLOSE updates the end of file pointer in the directory for the file being used. Say for example, that between the OPEN and CLOSE you've written four 256-byte records of data, plus a final record containing only 38 bytes of data. The CLOSE statement records these statistics so that the end of file can be known when you read the file back later.
- CLOSE frees the buffer so that, if your program requires it, another file can be opened.

All files are closed automatically if you RUN or LOAD a different program, when you do a NEW, edit the program in memory, or execute an END statement. If the diskette is removed, there's a power failure, or if you exit to DOS Ready without a CLOSE, all of the important functions CLOSE would normally perform remain undone.

OPEN"I"

Once data is stored on disk in a sequential file, you need a way to get it back. Perhaps the same program that recorded the data needs it again for a subsequent run. On the other hand, maybe you've got a different program, or several different programs that need the same data. The OPEN"I" statement gets your program ready to read a sequential file.

OPEN"I" works like OPEN"O". You've got to decide which file buffer you want to use, what disk file name, and (optionally) what drive. There's one big difference. The file name you specify has got to be on the disk already, and if you specify a drive, the file name has to be in the directory of the diskette in that drive. Remember, you can only use a given file buffer for one file at a time. You can't change the mode for a file in mid-stream. Once you've opened it, and until you close it, a sequential file is either output or input — not both.

Here's a sample open for input statement:

```
OPEN"I",2,"STORAGE/SEQ"
```

The "I" specifies that you are opening a sequential file in input mode. The 2 tells BASIC that you want to use file buffer 2. "STORAGE/SEQ" is the name of the file you want to input from. The drive number is not specified, so the name must be in the directory for one of the disks that you have on-line, or the OPEN will fail.

As with OPEN"O", you can use variables for any of the parameters in the open statement:

```
OPEN MD$, BF%, FS$
```

. . . opens a file, using the mode specified by MD$, the file buffer specified by BF% and the file specification (name and drive) stored in FS$.

INPUT#

When you've opened a sequential file in input mode, you want to retrieve the data it contains. Data is coming "in" to BASIC variables from the disk file. INPUT# works almost exactly like its keyboard counterpart, INPUT. It is even more similar to a READ command with data statements, but there still are differences.

Let's say, for instance, that a program has created a sequential file that has three numbers stored in it. Depending on the way they were outputted by a PRINT# statement, those numbers, as they are stored on disk, may be separated by blanks or carriage return characters.

If the PRINT# statement outputted a series of variables or numeric expressions, each separated by a comma, anywhere from 1 to 15 blanks will separate each number in the file, depending on the length of the variable stored.

If the PRINT# statement used semicolons to separate the numeric expressions, a single blank separates each number and its sign in the file.

If you used more than one PRINT# statement in building the output file, a carriage return character follows each number stored with a PRINT#. A carriage return is represented by 1 byte whose ASCII value is 13. (The <ENTER> key generates the same code.)

There are other ways that you can force certain types of delimiters to be between the numbers in the file. For example.

```
PRINT#1,A%;",";B%;",";C#
```

. . . will insert commas between the numbers in the file.

The INPUT# command doesn't care whether the numbers in an input file are separated by blanks, carriage returns, or commas. As long as they are separated properly, each will be inputted separately.

Assuming you've opened the input file in buffer 1, the INPUT# command to get the first three numbers, assigning them in sequence to A%, B%, and C# is:

```
INPUT#1,A%,B%,C#
```

Notice that, just like you did with the PRINT# statement, you specify the buffer number. Unlike the PRINT# statement, the list of items must contain variable names only, and they must be separated by commas.

If the input file contains " 3111 4322 1432 ", after the INPUT# statement shown above, A% contains 3111, B% equals 4322, and C# is 1432.

We also can divide our input into separate BASIC statements:

```
INPUT#1,A%
INPUT#1,B%
INPUT#1,C%
```

As separate statements, you can insert program logic between each input. Perhaps, in the example above, you want to do some calculations or printing after getting A% but before getting B%.

It is important to realize that you don't need to use the same variable names that you used when you outputted the file. The variable types don't even need to be the same, as long as the variable types you're using in the input statement have the capability to store the type of data that's coming in. A statement such as:

```
INPUT#1,G#,L#,B#
```

. . . could accept the same data.

As another thought, suppose the program is only interested in the third number, and you want to store it as G#. You have to read the first two numbers, but the input statement can be arranged to ignore them:

```
INPUT#1,G#,G#,G#
```

This statement works because you're only interested in the final value of G#.

Loading an array from a sequential file is another way that INPUT# statements might be applied. The following logic loads 10 elements of the A# array:

```
FOR X=1 TO 1Ø
INPUT#1,A#(X)
NEXT
```

At this point we are discussing sequential files as a way to store and retrieve numeric variables. We're assuming that the program routines doing the input know exactly how many items to expect in the file and in what order they will be found.

The INPUT# statement will result in an error message, INPUT PAST END, if you attempt to input more items than you stored in the file. We'll talk more about that later when we deal with situations in which your program doesn't know how much data to expect.

Here's another thing to keep in mind. Many times it's convenient, during testing and debugging to work with files using immediate commands in response to BASIC's READY prompt. You can use OPEN#, CLOSE#, and PRINT# without a line number. INPUT# is an exception. INPUT# must be executed as a statement in a BASIC program. If you attempt any INPUT from READY mode, you get an "ILLEGAL DIRECT" error.

Closing the Input File

The CLOSE command for a sequential input file is programmed exactly as it is for a sequential output file. With an input file, though, there is no information that needs to be recorded to disk upon closing, so the CLOSE command is not nearly as critical. Its main value is to free the buffer so another file can use it. Nevertheless, it's a good programming practice to always use a close statement when access to a particular file is completed.

A common requirement is to re-read the data in a particular sequential file more than once during a given program run. The only straightforward way to accomplish a re-read from the beginning is to CLOSE the file and OPEN it in input mode again.

Saving Variables, a Demonstration

A simple program will show how the ideas we've been discussing so far can be applied. The mileage calculator program introduced in the previous chapter could be made more automatic if it were able to remember the prior odometer reading for us. We also could get more information from the program if it would keep a running average of our mileage performance. A sequential file is an ideal way to save the variables that are required because it makes our programming job so simple.

MPGCALC2/DEM is the mileage calculator program, modified to take advantage of sequential disk file storage. To program it, take the MPGCALC1/ BAS program and add lines 11 through 13 and 100 through 2050. The complete listing is shown below:

Figure 4.1 — *MPGCALC2/DEM*

```
Ø 'MPGCALC2/DEM
1Ø CLS:PRINT"MILEAGE CALCULATOR":PRINT
11 GOSUB1ØØØ:IFLEFT$(YN$,1)<>"Y"THEN2Ø
12 PRINT"ODOMETER READING, LAST FILL UP, WAS:        ";R1
13 GOTO3Ø
2Ø INPUT"WHAT WAS YOUR ODOMETER READING, LAST FILL UP";R1
3Ø INPUT"WHAT WAS YOUR ODOMETER READING, THIS FILL UP";R2
4Ø INPUT"HOW MANY GALLONS DID YOU BUY, THIS FILL UP  ";G
5Ø INPUT"WHAT WAS THE PRICE PER GALLON (#.###)      ";PG
6Ø M=R2-R1:MG=M/G
7Ø PRINT"MILES DRIVEN      ";M
8Ø PRINT"MILES PER GALLON: ";MG
9Ø PRINT"COST PER MILE:    ";PG*G/M
1ØØ PRINT
11Ø PRINT"MILES TO DATE:    ";MT+M
12Ø PRINT"GALLONS TO DATE:  ";GT+G
13Ø PRINT"AVERAGE MPG:      ";(MT+M)/(GT+G)
14Ø PRINT
15Ø GOSUB2ØØØ
16Ø END

1ØØØ INPUT"LOAD DATA FROM DISK?  (Y=YES, N=NO)        ";YN$
1Ø1Ø IFLEFT$(YN$,1)<>"Y"THENRETURN
1Ø2Ø OPEN"I",1,"MPGCALC2/DAT:Ø"
1Ø3Ø INPUT#1,R1,MT,GT
1Ø4Ø CLOSE1
1Ø5Ø RETURN

2ØØØ INPUT"RECORD DATA TO DISK?  (Y=YES, N=NO)        ";YN$
2Ø1Ø IFLEFT$(YN$,1)<>"Y"THENRETURN
2Ø2Ø OPEN"O",1,"MPGCALC2/DAT:Ø"
2Ø3Ø PRINT#1,R2;MT+M;GT+G
2Ø4Ø CLOSE1
2Ø5Ø RETURN
```

MPGCALC2/DEM uses a sequential file named "MPGCALC2/DAT" and stores it on drive 0. Use of this file for input and output is done in the same program. Subroutine 1000 handles the input. Subroutine 2000 handles the output. Upon startup, you can tell the program to get the information as of the last run, and when you've entered the current information, you can save it on disk for the next time.

The MPGCALC2/DEM program only stores three numbers on disk: the closing odometer reading, a sum of the total miles driven to date, and a sum of the total gallons used to date. Notice how the PRINT# statement in line 2030 outputs the numbers. It writes the current odometer reading as the variable R2. It writes the new mile total as an expression, adding the prior mile total, MT, to the current number of miles, M. The gallon total is handled the same way. Notice that the number outputted as R2 is inputted as R1 in line 1030.

As with most programs of this type, you've got to allow for two possibilities:

1. Perhaps the user is running the program for the first time, so there won't be any input data available on disk. In that case, keyboard entry routines have to be provided.

2. What if the operator makes an error? He must be able to end a run with the option of *not* storing the new data on disk.

Here's how MPGCALC2/DEM interacts with the operator and how the program might appear on the video display after the program's been run a few times:

```
MILEAGE CALCULATOR

LOAD DATA FROM DISK?   (Y=YES, N=NO)        ? Y
ODOMETER READING, LAST FILL UP, WAS:          28ØØ
WHAT WAS YOUR ODOMETER READING, THIS FILL UP? 322Ø
HOW MANY GALLONS DID YOU BUY, THIS FILL UP  ? 18.5
WHAT WAS THE PRICE PER GALLON (#.###)       ? 1.299
MILES DRIVEN       42Ø
MILES PER GALLON   22.7Ø27
COST PER MILE:     .Ø572179

MILES TO DATE:     222Ø
GALLONS TO DATE:   88.5
AVERAGE MPG:       25.Ø847

RECORD DATA TO DISK?  (Y=YES, N=NO)        ? Y
```

Strings in Sequential Files

I purposely avoided using string variables in any of the sequential file output or input examples thus far. Strings can be a bit trickier in sequential files. If you output strings without being aware of the limitations, sooner or later you may get unexpected results when trying to do your input.

To see some of the problems quickly, consider the following short program:

```
1Ø INPUT"WHAT'S YOUR NAME";NA$
2Ø PRINT"HELLO ";NA$
```

A response without leading blanks, commas, or colons works as planned:

```
WHAT'S YOUR NAME? LEWIS ROSENFELDER
HELLO LEWIS ROSENFELDER
```

If our response has leading blanks, they are stripped by the INPUT statement:

```
WHAT'S YOUR NAME?    LEWIS ROSENFELDER
HELLO LEWIS ROSENFELDER
```

A response having a comma in it gives you more trouble:

```
WHAT'S YOUR NAME? ROSENFELDER, LEWIS
?Extra ignored
HELLO ROSENFELDER
```

BASIC took the comma as a delimiter. It thought you were trying to enter two answers to the same question. Had the input statement been set up to receive data for two string variables, it would have put everything beyond the comma into the second variable in our list. As an operator at the keyboard, you can get around the last problem if you type double quotes around your response:

```
WHAT'S YOUR NAME? "ROSENFELDER, LEWIS"
HELLO ROSENFELDER, LEWIS
```

I've shown these keyboard input examples because many of the same problems and solutions exist for INPUT# and PRINT# statements with sequential files. Consider the following program:

```
1Ø CLEAR 1ØØ                'CLEAR ALL VARIABLES
2Ø OPEN"O",1,"TESTFILE"     'OPEN A FILE FOR OUTPUT
3Ø A$="APPLE":B$="BOAT"     'LOAD 2 STRINGS WITH DATA
4Ø PRINT#1,A$;B$            'OUTPUT 2 STRINGS
5Ø CLOSE                    'CLOSE THE FILE
6Ø CLEAR 1ØØ                'CLEAR ALL VARIABLES
7Ø OPEN"I",1,"TESTFILE"     'OPEN SAME FILE FOR INPUT
8Ø INPUT#1,A$,B$            'INPUT 2 STRINGS
9Ø PRINT A$,B$              'PRINT THEM
1ØØ CLOSE                   'CLOSE THE FILE
```

When you run the program, it does fine until it gets to line 80. The program terminates with an "INPUT PAST END" error in 80. If, at that point, you PRINT A$, you will see that A$ contains "APPLEBOAT". B$ contains nothing. The PRINT# statement in line 40 failed to give BASIC any way to separate the two strings you outputted. The INPUT# statement in line 80 took everything into A$, so when it went to get B$, it was already at the end-of-file.

Now change line 40 so it outputs A$ and B$ in two separate PRINT# statements:

```
4Ø PRINT#1,A$:PRINT#1,B$     'OUTPUT 2 STRINGS
```

With this change the program gives the expected result:

```
APPLE         BOAT
```

By separating the PRINT# statements, you forced Disk BASIC to put a delimiter after each string. The delimiter it automatically used was ASCII code 13, a carriage return. As mentioned before, each PRINT# statement generates a carriage return terminator after the last expression listed.

Suppose, though, that you want to use just one PRINT# statement. You can force your own delimiter by outputting a comma between the two string variables. Change line 40 to:

```
40 PRINT#1,A$;",";B$          'OUTPUT 2 STRINGS
```

Now, the INPUT# statement in line 80 treats the data stored on disk, "APPLE,BOAT", as two strings. The program works as it should. Alternately, you could imitate the way that two PRINT# statements would store adjacent strings by using ASCII character 13 as a delimiter. This substitution in line 40 gives the same successful result:

```
40 PRINT#1,A$;CHR$(13);B$     'OUTPUT 2 STRINGS
```

If the string you want to output already contains a comma or a CHR$(13), you can run into problems, just as you might with a keyboard input statement. Consider the following change:

```
30 A$="AP,PLE":B$="BOAT"      'LOAD 2 STRINGS WITH DATA
```

You want "AP,PLE" to go out as one string, but again, you find that the INPUT# statement in line 80 takes AP as one string and loads it into A$, and it takes "PLE" as another string, which it loads into B$. The program never gets around to reading "BOAT". You can solve the problem just as you would with a keyboard input statement. Enclose the A$ string with quote marks when you output it. A double quote is ASCII code 34.

```
40 PRINT#1,CHR$(34);A$;CHR$(34);CHR$(13);B$
```

If there were also the possibility that B$ contained a comma, you'd want to enclose it with double quotes too.

We've used the preceding examples to show some of the things you've got to watch out for when inputting and outputting strings for sequential files. A "try-it-out" program like the one above is the best way to get familiar with what will work and what won't for the particular types of data you intend to store. The Disk BASIC section of your *Disk System Owner's Manual* gives you a more technical discussion of how PRINT# stores data on disk and how INPUT# statements evaluate the delimiters.

Numbers and Strings

When you output numeric and string expressions to the same sequential file you have more opportunities for unexpected results, but you also have opportunities for easy solutions to some of the problems we just discussed.

It's important to remember that a sequential file just stores a stream of data with various kinds of delimiters to separate the data. There is no explicit indication within the file about what kind of data is being stored. We saw that integer numbers outputted to a sequential file could be read back as single precision or double precision numbers. Numbers can also be read back as strings if they are separated properly by commas or carriage returns.

When an INPUT# statement is retrieving data to be put into two numeric variables, a blank is accepted as a delimiter between two streams of digits. If your file contains "1234 5678 " followed by a carriage return,

```
INPUT#1,A%,B%
```

. . . will load A% with 1234 and B% with 5678.

If you attempt to use string variables as input, BASIC will take the whole stream into the first variable.

```
INPUT#1,A$,B$
```

... will load A$ with "1234 5678". Unless there's more data in the file, an "INPUT PAST END" error will result, and B$ won't get loaded. On the other hand, if the input statement is:

```
INPUT#1,A%,B$
```

. . . A% will be loaded with 1234 and B$ will be loaded with "5678". This leads to a programming trick that can simplify things.

When you want to output a list of numeric and string expressions with one PRINT# statement, it's easiest if you put all your numeric expressions first, followed by your string expressions. You still need to provide for proper separation in the string expressions, though. As an example, let's suppose that you want to output the contents of A$, T#, B$, L% and C$:

```
PRINT#1,T#;L%;A$;",";B$;",";C$
```

... uses the sequence that will make your input job easiest. You can input the same data later with:

```
INPUT#1,T#,L%,A$,B$,C$
```

If you had some reason to store them in the other sequence, the statements would have to be:

```
PRINT#1,A$;",";T#;B$,",";L%;C$

INPUT#1,A$,T#,B$,L%,C$
```

If you wanted to output them as numeric and string expressions, and later input them all as strings, the statements might be:

```
PRINT#1,A$;",";T#;",";B$;",";L%;",";C$

INPUT#1,A$,T$,B$,L$,C$
```

If you output the variables with separate print statements, the separation job is taken care of for you:

```
PRINT#1,A$:PRINT#1,T#:PRINT#1,B$:PRINT#1,L%:PRINT#1,C$
```

Now, as long as your strings don't have internal commas or carriage returns, you can input as you wish without too much concern:

```
INPUT#1,A$,T$,B$,L$,C$
```

or,

```
INPUT#1,A$,T#,B$,L%,C$
```

We've shown how a string variable can be used to input a numeric, but what happens when your input statement requests numeric data and BASIC finds non-numeric information next in the file. For keyboard input, you get the "?Redo" message if non-numerics are entered when your program is attempting to load numeric variables. In the case of input from a file, you don't get a message. If a number is to be input into an integer, single precision, or double precision variable, and BASIC finds anything other than a blank, "–", decimal point or numeric digit as the next character, the variable will be loaded with zero.

Another situation can occur. Suppose that the next number in the file is 503433 and your input statement is:

```
INPUT#1,A%
```

Here you are trying to input an integer variable, but the next number in the file is too large. In this case you will get an "Overflow" error.

Carefree I/O With LINE INPUT#

Perhaps the discussions thus far have left you a little confused. I've got to admit that when output files combine strings and numerics, I have to stop and think awhile. Sometimes I've got to do a little experimentation to make sure the output and input will go as I planned. There is a solution, though, that can simplify all your sequential file programming. I call it the "carefree method."

LINE INPUT# is an alternative to the INPUT# statement. Just as LINEINPUT simplifies keyboard input programming, LINE INPUT# can simplify sequential disk file programming. With LINE INPUT#, you don't have to be concerned about strings that contain quotes, commas, or leading blanks. From any point in an input file, LINE INPUT# gets everything up to the first carriage return character encountered, or if 255 bytes exist without a carriage return, it gets the first 255 bytes.

There are some limitations:

- You can only use LINE INPUT# to load string variables. It cannot load integer, single, or double precision variables directly.
- You cannot input a list of different variables with a single LINE INPUT# statement. LINE INPUT# can load only one string variable at a time.
- If a LINE INPUT# encounters a carriage return (ASCII 13) immediately preceded by a line feed (ASCII 10), the carriage return character is not taken as a terminator.

Here's the carefree method, which I use in 90 percent of my sequential file programming:

When outputting to a sequential file . . .

- Use PRINT# statements for numeric or string expressions at will, but limit each PRINT# statement to one expression. (Forget about using lists of expressions.)
- You are free to output anything, as long as it doesn't contain an ASCII 13.

When inputting from a sequential file . . .

- Use LINE INPUT#. Then, if you want to use the data you retrieved as a numeric, use the VAL function to convert it.

To make these ideas clear, let's suppose you want to store the contents of L$, M$, L#, and M% in a sequential file. Your output statements (assuming you are using buffer 1) are:

```
PRINT#1,L$:PRINT#1,M$:PRINT#1,L#:PRINT#1,M%
```

After you open the file for input later, you can put the data back into the same variables with:

```
LINE INPUT#1,A$ : L$=A$
LINE INPUT#1,A$ : M$=A$
LINE INPUT#1,A$ : L#=VAL(A$)
LINE INPUT#1,A$ : M%=VAL(A$)
```

As you can see, the carefree method is not quite as frugal with program memory, but very little thought needs to go into the way the data is stored in the file. Your only concern is the sequence. Notice that I'm inputting everything as A$. Though everything above was shown as separate input statements, the idea of using one variable as an input variable lends itself very well to the development of an input subroutine.

The PRINT# statements could also be put into one subroutine, using A$ as the output variable. Suppose your output subroutine were line 56100:

```
56100 PRINT#1,A$:RETURN
```

Then we could say:

```
A$=L$:GOSUB56100
A$=M$:GOSUB56100
A$=STR$(L#):GOSUB56100
A$=STR$(M%):GOSUB56100
```

The advantages of using standard input and output subroutines may not be apparent here, but as you get into error handling and multiple files, there can be some attractive benefits.

The manuals say that the space in LINE INPUT is optional and that you can use LINEINPUT as one word if you wish. Forgive me, if you will, but I've gotten into the habit of programming it as two words. It seems that, several times a few years ago I ran into programming situations with "LINEINPUT#" that didn't work properly. When I changed the statements to "LINE INPUT#", they did work. I may have been seeing things, and I can't reconstruct the exact situation (it may have been an older version of Disk BASIC), but I've stuck with the separating space ever since, just to be sure.

In the pages that follow we will cover more examples that use LINE INPUT#. It's a handy statement with sequential files.

Sequential Files and DATA Statements

There are many similarities between sequential input files and DATA statements. Both programming devices will feed a BASIC program with a stream of numbers or strings. With both, there is no specification of what type of information is stored — that depends on how it is read, or input. Both programming methods rely on a sequential organization of data. Without special methods, you cannot use random access. Your program has to take the data as it's organized, from beginning to end.

Often, data statements are used simply to load information into arrays. At other times, they may be used to provide lists of numbers that are to be POKEd into certain areas of memory so machine-language USR subroutines can be used. In certain situations, it can be profitable to store the information commonly handled with data statements in a sequential file:

- If data statements occupy more than a few hundred bytes in a program, and the data is primarily numeric, a sequential file can save memory. A sequential file takes a fixed amount of memory, unrelated to the amount of data that is to be handled. Once the file is closed, that memory may be used to operate another file.

- When data is changeable from time to time, a sequential file can be a better answer, because when changes are made, no modifications need to be made to the program. A payroll tax table, which might only be changed every few years, is a good example.

- When data has to be provided to meet various needs, but the data for all the needs anticipated is not required in any one session, one or more sequential files on disk can give a faster and more memory-efficient answer. Having the payroll tax data for each state in a different sequential file on disk, and only loading the data for the state being computed is an example. A report printing-program that is able to handle various report formats is another.

Let's consider the case of a USR subroutine that must be POKEd into memory. The usual way is to store the numbers in data statements. Shown below are data statements for the 40-byte delete-array subroutine that is discussed in Appendix 2. The routine in line 10 reads the data statements, POKEing the values into 40 byes of protected memory starting at **FFOO**:

```
1Ø FOR X%=Ø TO 39 : READ A% : POKE &HFFØØ+X,A% : NEXT

1ØØØ 'DATA STATEMENTS FOR THE DELETE-ARRAY USR SUBROUTINE
1ØØ1 DATA 2Ø5,127,1Ø,17,4,Ø,183,237,82,229,7Ø,43,78,43,43,43
1ØØ2 DATA 227,9,35,229,235,42,253,64,183,237,82,229,193,225
1ØØ3 DATA 2Ø9,4Ø,2,237,176,235,34,253,64,2Ø1
```

The data statements in lines 1000 through 1003 take 154 bytes of program memory to specify information that is to be stored in 40 bytes. You could load all those numbers from a sequential file and save the 154 bytes of program memory. Assuming those numbers are in a file named DELARRAY/SEQ, our logic in line 10 could be:

```
1Ø OPEN"I",1,"DELARRAY/SEQ":FORX%=ØTO39:INPUT#1,A%
   :POKE&HFFØØ+X%,A%:NEXT:CLOSE1
```

The data statements are no longer necessary in the program, and once file 1 is closed, you can open it again for another purpose. This example loads a relatively small amount of data. When you have hundreds of numbers to be POKEd or loaded into an array, the technique can be really worthwhile.

Now you may be wondering how to get the data into a sequential file. Perhaps you've got programs with data statements in them and you'd like to put all the data in sequential files. The DATAOUT/DEM program makes it easy for you. Simply enter the data you want to write into a sequential file as DATA statements, starting preferably at line 1001 or higher. If you have a BASIC program written already, merge the DATA statements from that program into the DATAOUT/DEM program. (The line numbers of your DATA statements are unimportant, as long as they don't conflict with the logic from lines 100 to 120 and 900 to 904.)

The DATAOUT/DEM program is based on DATASUB/DEM that was presented in Chapter 3. You don't need to know how many data items there are. It automatically senses the end for you. You will need to replace the output file name in line 100 to specify the name and drive you want the data stored on. DATAOUT/DEM here is shown as it would be used to get the numbers for the DELARRAY USR subroutine into a sequential file named "DATAFILE" on drive 0. You will, of course, want to delete lines 1001 through 1003 and put in your own data statements.

Figure 4.2 — *DATAOUT/DEM*

```
Ø 'DATAOUT/DEM
1ØØ OPEN"O",1,"DATAFILE:Ø"    'NOTE: CHANGE NAME AS NEEDED
11Ø GOSUB 9ØØ : IF ED% THEN 12Ø ELSE PRINT#1,A$ : GOTO 11Ø
12Ø CLOSE1:END

9ØØ 'READ-DATA SUBROUTINE
9Ø1 ED%=Ø:ONERRORGOTO9Ø4
9Ø2 READ A$
9Ø3 ONERRORGOTOØ:RETURN
9Ø4 IFERR/2+1=4THENED%=-1:RESUME9Ø3ELSE9Ø3

1ØØØ'DATA LIST - NOTE: REPLACE DATA AS NEEDED ****************
1ØØ1 DATA 2Ø5,127,1Ø,17,4,Ø,183,237,82,229,7Ø,43,78,43,43,43
1ØØ2 DATA 227,9,35,229,235,42,253,64,183,237,82,229,193,225
1ØØ3 DATA 2Ø9,4Ø,2,237,176,235,34,253,64,2Ø1
```

Detecting End of File

Up until now we've assumed that we know exactly how many items of data the sequential input file contains. This can be a valid assumption if you are storing a specific list of variables, a certain number of items to be loaded into an array, or a constant number of codes to be POKEd.

As you get into more complex applications for sequential files, it's not always possible to know before-hand how many items the file contains. The problem you encounter is that if you attempt to continue input after the last item in a file has been read, the program can terminate with an "Input Past End" error.

EOF stands for end of file. The EOF function is used only with files opened for sequential input. For any file buffer from 1 to 15, as long as it has been opened in "I" mode, the EOF function is equal to 0 if we have not yet input the last item. It equals –1 if you have reached the end of the file. The EOF function requires you to provide a numerical expression to specify the file buffer you want to check.

As you may know, –1 is the usual way to indicate a true condition, and 0 indicates a logical false condition. Because of this, if BF% is the file buffer you're using, you can simply say,

```
IF EOF(BF%) THEN ....
```

After the "THEN", we can send the program flow to the routine that is to be executed after every item in the file has been input. Normally, the next operation in this case will be a CLOSE for the file.

You can also handle the logic the opposite way:

```
IF NOT EOF(BF%) THEN INPUT#BF%, .....
```

or,

```
IF NOT EOF(BF%) THEN LINE INPUT# BF%, .....
```

or,

```
IF EOF(BF%)=Ø THEN LINE INPUT# BF%, .....
```

Suppose you have a sequential file, "DATAFILE/TXT", and you don't know how many data items it has. You want to read and print each item in the file. Here are a few program lines that could handle the process, assuming you are using file buffer 1:

```
1Ø CLEAR 3ØØ                'WE NEED TO CLEAR AT LEAST 255
2Ø OPEN"I",1,"DATAFILE/TXT" 'OPEN A FILE FOR INPUT
3Ø IF EOF(1) THEN 1ØØ       'CHECK FOR END OF FILE
4Ø LINE INPUT#1,A$          'INPUT NEXT ITEM OF DATA
5Ø PRINT A$                 'PRINT IT
6Ø GOTO 3Ø                  'REPEAT
1ØØ CLOSE 1                 'END OF FILE, SO CLOSE IT
11Ø END                     'END THE PROGRAM
```

INPUT# statements that specify a list of variables to be input cannot fully benefit from the EOF function. Your program can check for EOF just before the list is input, but if BASIC fails to find enough items to supply the list, you will get an "Input Past End" error. Normally, though, this isn't a problem, because if you've taken care in outputting the data, your input program will know how it's organized.

Programmers sometimes wish to build their own end of file indicators into a sequential file by storing a data item that has a specific value that can be tested for. As an example, suppose that a file contains a list of numbers to be loaded into the A# array. Those numbers could be input as strings, until a special string indicates that the last number has been retrieved. The example below uses the special string, "EOD" to indicate that the last number has been received:

```
1Ø CLEAR 3ØØ                'WE NEED TO CLEAR AT LEAST 255
2Ø DIM A#(499)              'ALLOW FOR A MAX OF 499 ELEMENTS
3Ø OPEN"I",1,"DATAFILE/TXT" 'OPEN A FILE FOR INPUT
4Ø X=Ø                      'X WILL INDICATE ELEMENT NUMBER
5Ø LINE INPUT#1,A$          'INPUT NEXT ITEM OF DATA
6Ø IF A$="EOD" THEN 1ØØ     'JUMP IF END OF DATA
7Ø A#(X)=VAL(A$)            'LOAD NUMBER INTO ARRAY
8Ø X=X+1:GOTO5Ø             'REPEAT
1ØØ CLOSE 1                 'END OF FILE, SO CLOSE IT
11Ø ....                    'CONTINUE PROGRAM HERE
```

Your own end of data indicators can be especially useful when a single sequential file contains more than one series of data. Perhaps the file contains an unknown quantity of integers to be loaded into one array, followed by an unknown quantity of double precision numbers to be loaded into another. A special indicator can separate the two groups of items. When that indicator is identified by the program, the logic can branch to the routine that handles the second group.

Another approach is to store a number in the file that tells the program how many items follow. This method is quite valuable when you are loading arrays because it makes it possible for your program to dimension the applicable arrays with just enough elements to hold the information that will be input. The result is an elimination of wasted memory, and in most cases, a faster running program.

LOF and LOC

LOF stands for length of file. The LOF function tells you the length of any sequential file that is open for input or output. The length is expressed in terms of 256-byte records. The LOF for a particular file is the record number of the last record in the file, which may or may not contain 256 active bytes. Like the EOF function, you specify the file buffer number as the parameter:

```
LOF(1)
```

. . . returns the ending record number for the file that has been opened as file 1.

The LOF function can be important for sequential files opened in output mode as a way to anticipate and prevent disk full conditions. Let's say that before entering BASIC you looked at a directory of the disk and determined that you'd have space to store a file of 30 records. If you've got a large amount of data to output to a sequential file, you can have your program check the LOF after each PRINT# statement. When it approaches 30, you can flash a message to the operator to indicate that the disk is about to fill up. This allows the program to be terminated gracefully without a loss of data.

The LOC function is not available with TRSDOS 2.3 on the Model I. On TRSDOS 1.3 for the Model III, and other versions of Disk BASIC for the TRS-80, LOC tells your program the current location within a file. For sequential files, it specifies which 256-byte physical record is being used. Upon opening, the LOC of a file is 0. As soon as you input or print the first data item, the LOC is changed to 1. Once the total data (including any delimiters) input or output exceeds 256 bytes, the LOC changes to 2. For sequential files opened for output, the LOC is always equal to the LOF. The LOC function can be more valuable to you in a sequential file that has been opened for input. If you know the LOF, you can use the LOC to gauge how far you've read through the file so far, and you can tell about how much more data (in 256-byte increments) remains to be input.

As an example, suppose you are inputting from a long sequential file that has been opened as file 3. At any point during the process you can have your program print an estimate of your current position:

```
PRINT"WE HAVE READ ABOUT ";1ØØ*LOC(3)/LOF(3);"% OF THE FILE"
```

It's not very common to see LOF and LOC used in sequential file programming, but you might want to try these ideas.

OPEN"E"

Model III Disk BASIC provides another open mode for sequential files. OPEN"E", or "OPEN EXTEND" is similar to OPEN"O", but it adds data onto the end of an already-existing sequential file. Once the file is opened in extend mode, the programming is identical — with PRINT# statements. BASIC with Model I TRSDOS 2.3 does not offer this feature. It is available for the Model I, though, with most of the other operating systems that can be purchased.

OPEN"E" requires that the file name already be on disk. If it's not, a "File Not Found" error will result. The first time you use the file, it must be opened in output mode. All subsequent opens for the file are done with OPEN"E" or OPEN"I". When you want to clear the data the file contains and start over, you can use an OPEN"O" again.

OPEN"E" is useful in applications where you would like to accumulate data over more than one session at the computer. Say, for example, that you wanted to keep a history of the gasoline prices paid and gallons purchased. You could modify the MPGCALC2/DEM program so after each run this information could be added onto the end of a history file. Perhaps you could call it MPGCALC2/HIS. Then you could write a sequential file input routine which would list each purchase made since the history file was first created.

If your version of Disk BASIC doesn't have the OPEN"E" feature, there are ways to achieve the same results. One way to add new data to an old sequential file is to output the new data to a file with a different name, then use the TRSDOS library command, APPEND to combine the two files into one. As our discussion of sequential files continues, other ways will become apparent.

Sequential File Errors

If your programming experience up until now has not involved disk files, you probably have not had much reason to concern yourself with the error trapping statements that BASIC provides. Disk file programming is different. Error conditions are an everyday concern. You've got to deal with diskettes that may not be inserted properly, drives that may not be ON, file names that might not be present, and a dozen or so other conditions that can occur. In this section we will look at error trapping as it applies to sequential files.

The ON ERROR GOTO statement is the primary tool for error trapping. Its use was discussed in the previous chapter. Let's look at the most prevalent error conditions that occur with sequential files.

During an OPEN"I", the following errors are most likely:

Code	Message
53	Bad file number
54	File not found
56	File already open
57	Disk I/O error
65	Bad file name
68	Too many files
7Ø	File access denied

"Bad file number", if it occurs, can be a programmer's error or an operator's error. It means that your program is trying to open a file in a buffer that has not been reserved. As an operator's error, the specification for "How many files" was wrong when BASIC was entered. The number specified, if any, was too small. Perhaps ENTER was pressed in response to the "How many files?" question, so space for 3 file buffers was automatically reserved, and your program is attempting to open a file 4 or higher. As a programming error, you may have provided for an AUTO startup from TRSDOS that didn't anticipate the number of files your program would require, or, somehow, the OPEN statement has gotten a number that is not in the valid range.

"File not found" occurs if you attempt to open a file that is not in the directory of the drive specified. If no drive number was specified in your open statement, a "File not found" error occurs if the file is not on the directory of any of the disks currently on line. Another cause is a file name that is typed incorrectly. Perhaps the operator intends to open a file named "DATAFILE", but types "DATAFLIE" instead. TRSDOS 1.3 on the Model III also returns the error message if a diskette is not present, or the door is open on the drive specified. (TRSDOS 2.3 on the Model I gives "Too many files" in either of these situations.)

"File already open" is usually a programmer's error. It means that the disk drive buffer number you've specified in your open statement is already in use for another file. A file buffer can be used for only one file at a time. To use the same buffer for another file, or the same file in another mode, you have to CLOSE it before you can do an OPEN. Many programmers protect themselves from this common error by routinely putting a close statement right before an open statement. Here's an example, assuming file 2 is to be opened:

```
CLOSE2 : OPEN"I",2,"DATAFILE/SEQ"
```

If the file in buffer 2 is not already open, the close statement is harmless. If there is a file that's open, it's closed, and the subsequent open statement can execute successfully.

"Disk I/O Error" is potentially the most serious of the errors that might be encountered during an OPEN"I". It means that the drive is having trouble reading the diskette, or it senses a problem on the diskette. Most disk I/O errors at open time are caused by a diskette that is not properly inserted in the drive. Perhaps the inner surface is not properly centered on the spindle, or maybe the diskette is in upside down. It's a good idea to provide routines in your programs that anticipate this, allowing the operator to "Press ENTER to retry".

In the case of an OPEN where the drive number is not specified, the disk I/O error may not have anything to do with the file that's being opened. Say, for example, that the file specification is "INVEN/SEQ" and the file resides on drive 2. To find it, the disk operating system first searches the directories of drives 0 and 1. If an unformatted disk is in drive 1, or if the directory in 0 or 1 is damaged, the error may occur before the system ever gets to drive 2. It is important to watch the lights on the drives to see just what's happening.

Another possible cause of a disk I/O error is a drive that is misaligned, dirty or otherwise malfunctioning. That may not be too bad if the problem occurred since the file you are attempting to open was created. You can get the drive fixed or replaced, but if the hardware problem was present when the file was outputted, it may be unreadable by this or any other drive. The only question to be answered then is, "Do you have a backup?"

Of course, smudges and dirt on the diskette can be the cause. Quite possibly the damage is magnetic. If just one bit is wrong, the system's parity calculation can be thrown off, and the file will be considered unreadable. Sometimes the problem is a damaged directory. That can be caused by hardware and media malfunctions, but it can also be caused by such things as swapping diskettes in the drives while a file is open and then closing the file.

"Bad file name" errors occur when your program attempts to use a name that's not legal. It's more likely if your program allows the operator to enter the file name to be used, and you don't provide ways to verify that the name is proper before the open. See your *Disk System Owner's Manual* for discussion of the constraints placed on you in selecting file names.

As far as I know, the "Too many files" error won't occur with an OPEN"I" using Model III TRSDOS. It can occur with TRSDOS 2.3 on the Model I. It seems to be a catch-all for things like disk drive doors that are left open, or requests for drives that are not connected to the system. If this error occurs when a file is to be opened, check to make sure the drives are connected properly and getting power.

“File Access Denied” occurs when the file being opened for input has been assigned a password. Your open statement either omitted the password, or used the wrong one.

OPEN“O” and OPEN“E” are subject to all of the same errors that can result with an OPEN“I”. The main exception is OPEN“O”, which never results in a “File not found” error. Here are the others that might occur:

Code	Message
62	Disk Full
68	Too Many Files
69	Disk Write Protected

With Model III TRSDOS, “Disk Full” occurs during an open if there is no more space available in the disk directory to store a new file name. The error message for Model I TRSDOS 2.3 in this case is “Too Many Files”.

Just because “Disk full” does not occur when you open a file for output doesn’t mean that there are any free granules of file storage space available. BASIC doesn’t check whether or not there’s enough space to store the data you output until it’s first faced with the need to record data. That may not be until you’ve output 256-bytes or you’ve attempted to close the file.

“Disk Write Protected” means that the notch on the diskette is covered, usually with a piece of tape, so that the drive is prevented from recording on it. This error message gives you the chance to ask yourself if you really want to write on that diskette. If so, you can remove the tape covering the notch, re-insert it, and try your open command again.

Model III TRSDOS allows a software write protect, through the WP library command. To remove write protection from all drives in the system, you can go to TRSDOS and enter the command, WP. Software write protect gives the same error indications in BASIC as a physical write protect does.

During INPUT# and LINE INPUT# you may encounter any of the following errors:

Code	Message
53	Bad file number
55	Bad file mode
57	Disk I/O error
63	Input past end

“Bad file number” will occur if your input statement specifies a file number that is greater than the number of files you specified when you entered BASIC. Its meaning and correction is the same as it is during an OPEN.

“Bad file mode” is a programming error. Your program is probably attempting to input from a file that was opened in “O”, “E” or “R” mode.

The causes and corrections for disk I/O error during input are much the same as they are during an open. In this case, BASIC has sensed a bad spot in the file and it cannot continue. It is often helpful to provide a retry capability because subsequent attempts often solve the problem.

“Input past end” means that you’ve gotten to the end of file, but your program is still trying to do an INPUT# or LINE INPUT# command. The best solution for this error is to avoid it by testing the EOF function before executing your INPUT# or LINE INPUT#. It's discussed elsewhere in this chapter.

Input past end can occur unexpectedly if you are inputting from a file that wasn't closed properly after it was opened for output. If, for instance, an output sequential file was created during the last session, but it was not closed because of a power failure or operator carelessness, the end-of-file pointer failed to get updated. Now, when you attempt to input from it, the end of file pointer is still 0, or whatever it was the last time the file was closed successfully.

The errors resulting from PRINT# statements are listed below:

Code	Message
53	Bad file number
55	Bad file mode
57	Disk I/O error
62	Disk Full
69	Disk Write Protected

"Bad file mode" can result when you attempt to do a PRINT# if you opened the file in any mode other than "O" or "E".

Disk I/O errors can occur when BASIC attempts to write the buffer you've been filling to disk. It's good to provide a retry capability. Remember that disk writes are delayed until the buffer is full or the file is closed. The attempt to write the information to disk may determine whether dozens of items output earlier in your program will get recorded. If you can't overcome the I/O error, you probably won't be able to close the file either. Usually, the only practical solution is to start the program over, working from backup files or diskettes if necessary.

"Disk full" errors can also be serious when PRINT# statements are being executed. In statements outputting more than one expression, it's not always possible to tell what the last item successfully recorded was. About the only way is to close the the file and read it back to see what you've got. The best way to handle disk full errors is prevention. Plan your requirements before you start.

Operator-Specified File Names

A disk file name can be up to 8 letters and/or digits, as long as the first is a letter. An optional slash followed by up to 3 letters or digits may form a file name extension. A period followed by up to 8 letters or numbers may follow the name and extension as a password. Finally, the drive may be specified with a colon and drive number. All in all, a valid file name (or "file specification") may be a string of up to 23 characters in length.

Most often file names are hard-coded into the programs that create and access the files. That is, somewhere in in the program logic, the programmer has pre-defined the file names that will be used. All the examples in this chapter so far have shown hard-coded file names. The casual operator may be unaware what names are being used, and cannot change them without modifying the program.

Many programming situations call for a capability that lets the operator specify one or more of the file names to be used, in whole or in part. If you can be sure an experienced and "aware" operator will be using the program you write, you don't need to take too many precautions in your programming. On the other hand, you can't always make that assumption.

A program that accepts a file name from the operator must allow for some common situations:

- An invalid file name is entered. Perhaps it begins with a number instead of a letter, or maybe the operator used one or more characters that are not permitted.

- When an input file name is being specified, perhaps the operator made a typo. If the name is misspelled, the program won't be able to find it on disk.

- When an input file name is entered, perhaps it isn't in the directory of any of the disks that are on line. Again, it won't be found.

- When an output file name is entered, perhaps the operator types the name of a valuable program or data file that is already on disk. When your program opens the file with the name specified in output mode, important information is, in effect, erased.

- When an input file name is entered, a valid name may be selected that exists on the disk, but, perhaps it's not the right kind of file for the purpose at hand. An example is when the operator is given the opportunity to request an input file that contains investment data, but, in error, an input file that contains something entirely different, such as payroll data, is selected.

There are many solutions that prevent these potential problems. We'll discuss a few of them here.

Constructing File Names

One approach you can use, instead of having your program ask the operator for the whole file name in one entry, is to make your program construct a file name based on a series of operator entries. Here's an example:

```
1Ø LINEINPUT "ENTER THE 5-LETTER FILE NAME: ";A$
11 IF A$<"Ø"ORLEN(A$)>5THEN1ØELSENA$=A$
2Ø LINEINPUT "WHAT DISK DRIVE NUMBER (Ø-3)  ";A$
21 IF LEN(A$)<>1ORINSTR("Ø123",A$)=ØTHEN2ØELSEDN$=A$
3Ø FS$="INV"+NA$+"/DAT:"+DN$
4Ø PRINT"OPENING FILE: ";FS$ : OPEN"I",1,FS$
```

This example illustrates how a file name might be constructed to access files on disk that contain information on investments. The programmer has decided that file names are to begin with "INV" and end with "/DAT". The operator is to supply one to five letters or numbers to differentiate multiple investment files on the same disk. As far as the operator is concerned, file names are limited to 5 letters or digits. In line 11, the program verifies that at least one character, and no more than 5 have been entered. The IF test will be true for all filenames whose first character is a valid character of zero or higher in ASCII value. Line 30 constructs the file name by concatenating "INV" plus the identifier the operator has supplied plus "/DAT:" and the drive number (validated in line 21) the operator has requested. Besides making the purpose of the files readily apparent when a directory is displayed, the "INV" prefix serves to force the file name to begin with a letter.

Here are some examples of valid file names that the above routine might generate:

```
INVØØ1/DAT:Ø    INVABCD1/DAT:2    INV21/DAT:Ø
```

You can see that the routine doesn't anticipate every possible error. The operator is not prevented from putting blanks or special characters within the name. That's why it's important to let the operator see what file name has been constructed, so if there's a problem, it can be identified. Line 30 displays the file specification, as stored in FS$, just before the file is opened.

I once used an idea similar to this in an inventory program I wrote for a retailer. The inventory was divided into departments, each department having its own set of 3 files. As far as the user was concerned, departments had to be identified with 2-digit numbers. On disk, the file names were constructed as "DEPT" plus the two digits supplied, plus an extension. DEPT01/MAS, DEPT01/IND, and DEPT01/SEQ held the data for department 1; DEPT23/MAS, DEPT23/IND, and DEPT23/SEQ held the data for department 23, and so forth.

Output File Controls

Assuming that you've done some preliminary checking on the validity of a file name provided by the operator, there are some other controls and protections you can incorporate into your OPEN routine.

Remember, a sequential file opened in output mode destroys the contents of any pre-existing file having the same name on the drive selected. Sometimes you'll want to warn the operator that this is going to happen with the name that has been supplied.

How does your program know if the file name is already on disk? Simply provide logic that attempts to open the file for input before it's opened for output. If the file is not found, we know it's okay to proceed. If the file is found on the disk, you can give the operator an opportunity to abort.

Here is some logic that you might want to implement to make sure a file name entered by the operator is valid for output. You may want to change the line numbers and fancy it up for your own programs or make it into a standard subroutine. It shows some of the things you might want to allow for in a file that is to be opened for output:

```
1Ø ONERRORGOTOØ : LINEINPUT"ENTER THE FILE NAME AND DRIVE: ";FS$
2Ø ONERRORGOTO25 : OPEN"I",1,FS$ : CLOSE1
21 LINEINPUT"FILE'S ALREADY ON DISK. USE IT? (Y=YES) ";A$
22 IF A$="Y" THEN 3Ø ELSE1Ø

25 IF ERR/2+1=54 THEN LINEINPUT"FILE'S NOT ON DISK. CREATE IT? (
Y=YES) ";A$ : IF A$="Y" THEN RESUME 3Ø
26 IF ERR/2+1=65 THEN PRINT"BAD FILE NAME!":RESUME1Ø
27 PRINT"DISK ERROR, CODE";ERR/2+1:RESUME1Ø

3Ø ONERRORGOTO35 : OPEN"O",1,FS$ :ONERRORGOTOØ:GOTO4Ø
35 IF ERR/2+1=69 THEN PRINT"DISK IS WRITE-PROTECTED!":RESUME 1Ø
36 PRINT"DISK ERROR, CODE";ERR/2+1:RESUME1Ø

4Ø 'PROGRAM CONTINUES HERE IF FILE'S BEEN OPENED SUCCESSFULLY
```

Line 10 is where the operator enters the file name to be opened for output. Line 20 opens the file in input mode to see if it is already present on disk. An error code of 54 means that the file isn't on disk. In some cases, this may alert the operator that the wrong file name has been typed, so a chance to re-enter the file name is given. No error code means that the file *is* on the disk. This will alert the operator, who knows that it shouldn't be there yet, that something may be amiss. Again the program gives the opportunity to abort and retype the file name.

Line 26 traps a bad file name, such as one that doesn't begin with a letter, or one that contains illegal characters. It sends the program back to line 10 where the name can be retyped. Other errors (such as diskettes that are inserted improperly, unformatted diskettes and drives that aren't working) fall through to line 27. Here, instead of trying to anticipate every possible situation, our program displays the error code and resumes at 10 to give a chance to correct the situation and try again.

Once the operator has been informed whether or not the file name is already on disk and the possible error conditions have been eliminated, the routine goes to line 30, where the file is finally opened in output mode. Here there is one other potential condition that must be allowed for — a write protected disk. In that case, the program goes all the way back to 10, where the operator can enter another file name and drive, or uncover the notch on the diskette to try again.

Input File Controls

Like output files, input files are subject to bad file name errors; errors resulting from improperly inserted diskettes; unformatted or damaged diskettes; or malfunctioning drives. These types of conditions can be handled in much the same way. A more common problem that your program should anticipate is the possibility that the file name requested for input is not in the directory. This may be because the wrong disk has been inserted, or perhaps it's due to a typing error.

Here's an example of the logic that might be used to open a sequential input file with error handling:

```
50 ONERRORGOTO0 : LINEINPUT"ENTER THE FILE NAME AND DRIVE: ";FS$
60 ONERRORGOTO65 : OPEN"I",1,FS$ :ONERRORGOTO0:GOTO70

65 IF ERR/2+1=54 THEN PRINT"FILE NOT FOUND! ":RESUME50
66 IF ERR/2+1=65 THEN PRINT"BAD FILE NAME!":RESUME50
67 PRINT"DISK ERROR, CODE";ERR/2+1:RESUME50

70 'PROGRAM CONTINUES HERE IF FILE'S BEEN OPENED SUCCESSFULLY
```

Internal Identifiers

Just because an input file name has been found on a disk and opened successfully, it doesn't necessarily mean it's the right file. As long as protection and password status permits, BASIC can open any existing file in input mode. If the file was previously written in any mode other than sequential output, the data your program will find in the file will probably be totally inappropriate. You will get a small measure of protection if you use file names that give sufficient identification, but sometimes you'll want to really make sure your program is (pardon the expression) "idiot proof."

One way to verify that a file opened for input is the right one is to record an identifier as the first data item. Let's say, for example, that the diskette in drive 1 contains many programs and files. Among them are several files that have player statistics for teams in a football league. We want to allow the operator to use any file name, but you want to make sure he doesn't access a file that is for something else. The problem can be solved by designing the output routine so the first item in a football statistic file is the string

"FBSTATS", for example. That can be accomplished by a executing the program statement:

```
PRINT #1,"FBSTATS"
```

. . . as the first output command after the file is opened for output.

For the program that inputs the file, use an open input routine like the one shown in lines 50 through 67 in the prior example. Just add line 70 for a final validation, before actual input is begun:

```
7Ø LINE INPUT#1, A$ : IF A$<>"FBSTATS" THEN PRINT "THIS IS
NOT A FOOTBALL STATISTICS FILE!": CLOSE 1 : GOTO 5Ø
```

The popular spread-sheet program, VISICALC, uses this idea in files that it creates for use by other programs under its standard Data Interchange Format (DIF). A DIF file is a sequential file whose first item is the string "TABLE", meaning that it stores tabular (row and column) data, stored in a particular format.

LIST and PRINT

Information in sequential files is stored as ASCII-coded text. This means one byte is used for each digit, letter, number or character, and no special conversions or decompressions are required to interpret the data. It's worthwhile at this point to call your attention to certain library commands that are available from TRSDOS Ready mode.

Model III TRSDOS 1.3 lets you display or print the contents of a sequential file without writing a special program. Simply go to TRSDOS Ready, and enter the following:

```
LIST filespec (ASCII,SLOW)
```

Replace "filespec" with the name of the sequential file you want displayed. If you want to have a printout of the contents of your sequential file, turn your line printer ON and enter:

```
LIST filespec (ASCII,PRT)
```

Again, replace "filespec" with the name and drive of the file you want listed.

If you leave out the "ASCII" parameter, the file is listed or printed in a different format. Each byte, including the characters used as delimiters, is shown. Below each byte, the ASCII value, from 0 to 255 is shown.

TRSDOS 2.3 has two library commands that perform the ASCII LIST functions. From DOS Ready, simply enter:

```
LIST filespec
```

. . . replacing "filespec" with the name of the sequential file. Its contents will be displayed and rolled up on your video. To get a printout on your line printer, the library command with TRSDOS 2.3 is:

```
PRINT filespec
```

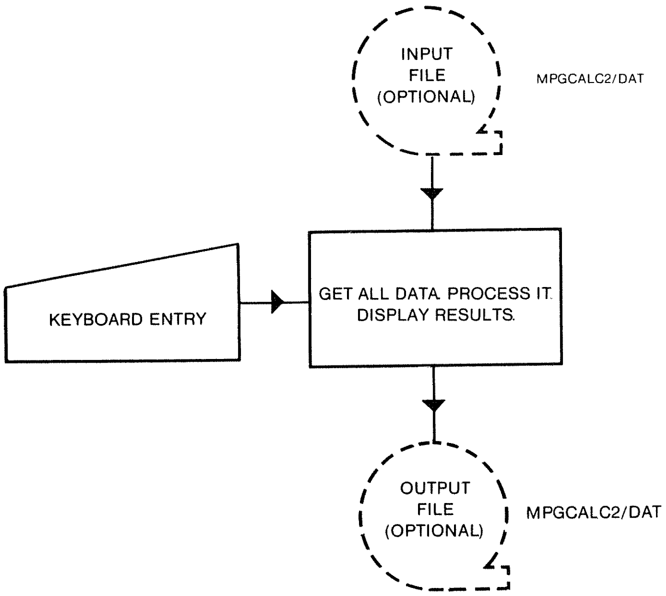
If you've got Model I TRSDOS 2.3 and you want a detailed, byte-for-byte interpretation of a sequential file that shows the ASCII value of each byte, including delimiters, use the BASIC program, DISKDUMP/BAS. It is provided on the TRSDOS 2.3 diskette.

These PRINT and LIST commands are handy when you want to see exactly how the data is organized in a sequential file. If you LIST an unknown file of any type and it appears to show jumbled, meaningless letters, numbers and graphics blocks, you are probably looking at a non-sequential file. A sequential file, on the other hand, will be easy to read. The data items will roll neatly across the video screen.

External Storage Concepts

Up to this point, we've looked at sequential files as ways to store data that is to be loaded into memory. We've outputted the contents of certain variables or arrays, so that they could be inputted by other programs, or other routines in the same program in subsequent sessions. We've used sequential files in ways that are analogous to the way SAVE and LOAD are used with programs. We have, in effect, saved data from variables, and loaded the data back. The typical operation is illustrated by the sample program, MPGCALC2/DEM. Only one file (MPGCALC2/DAT) holds the data, and it may be input in its entirety at the start, and/or output in its entirety at the end. Flowchart 1 shows the process:

Figure 4.3 — *Flowchart 1*

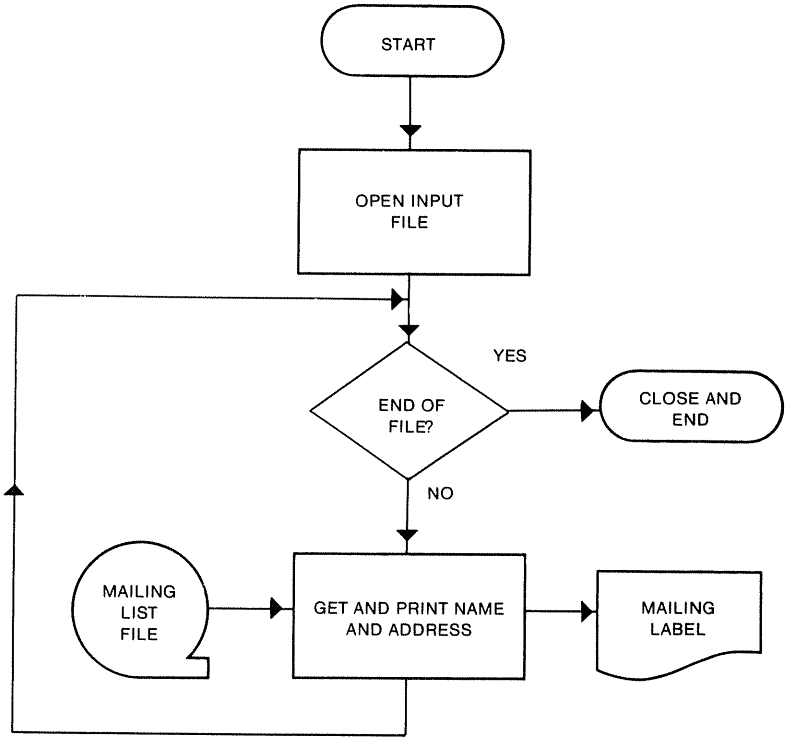


The save-load approach to sequential files is very important and useful, but it is just one way to use them. What happens when you wish to store more data in a file than can be stored in memory at any one time? Since you can't load all the information into variables and arrays, you have to read through the file, using the data you retrieve as you find it. This "external" approach to sequential files is like a buffet. Instead of having the "waiter" bring the whole "meal" at once, the program goes through a "buffet" line, picking and choosing the data it wants to "eat."

Say, for example, that you want to print a mailing list from a sequential file that contains a series of names and addresses. Your program opens the file for input, reads in the first name and address, and prints it. Then it reads the next name and address, and prints. The process is repeated until the end of file is reached. You don't care how many names and addresses you have because you're not loading them all into memory at once.

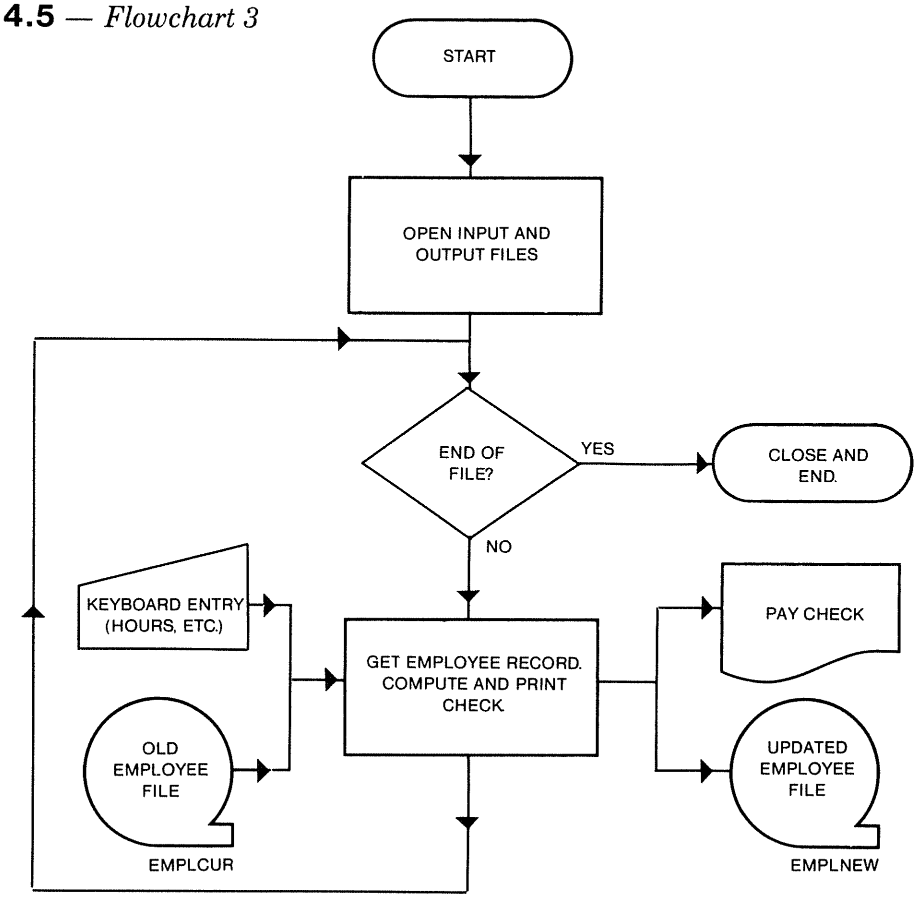
That kind of process is illustrated by Flowchart 2:

Figure 4.4 — *Flowchart 2*



Suppose that it's payroll time. Your program has a sequential file containing all the employee names, their pay rates, their tax codes and totals for quarter-to-date and year-to-date earnings. The program inputs data for the first employee, and you enter the number of hours worked. Then the program computes the withholding taxes and prints a check. Before going on to the next employee, an exact copy of the first employee's data is written to an output file, but in the output file the to-date earnings, tax and deduction totals have been updated. Now, one by one, the program inputs data for each subsequent employee in the file, prepares a check, and outputs the employee's information to the other file. That process is illustrated by Flowchart 3.

Figure 4.5 — *Flowchart 3*


START
OPEN INPUT AND OUTPUT FILES
END OF FILE?
YES
CLOSE AND END.
NO
KEYBOARD ENTRY (HOURS, ETC.)
GET EMPLOYEE RECORD. COMPUTE AND PRINT CHECK.
PAY CHECK
OLD EMPLOYEE FILE
EMPLCUR
UPDATED EMPLOYEE FILE
EMPLNEW

In Flowchart 3 the input file uses the file name "EMPLCUR", meaning that it contains current employee information. The output file is named "EMPLNEW", meaning that it has the updated employee information. Not shown is a file called "EMPLOLD". That file contains the employee data for the prior pay period. Before the next payroll run, a few operations have to be done so the "EMPLNEW" becomes "EMPLCUR":

1. KILL EMPLOLD, it is no longer needed.
2. RENAME EMPLCUR-> EMPLOLD. Now EMPLOLD is the backup file, in case you need to reconstruct the last payroll.
3. RENAME EMPLNEW -> EMPLCUR. The EMPLCUR file now has the latest payroll data to be used as input for your next run.

Under this scheme, the disk needs to be able to store 3 versions of the payroll file, EMPLOLD, EMPLCUR and EMPLNEW. In the event the program has trouble reading EMPLCUR during any payroll run, you can recover by copying EMPLOLD to EMPLCUR, and redoing last period's payroll.

If you have more than one drive, you can eliminate the kill and rename operations. You can have one file, named "EMPLOYEE" stored on three different diskettes. During the operation of the program, EMPLOYEE is opened for input on drive 1, and EMPLOYEE is opened for output on drive 2. The disk in drive 2 is used in drive 1 for the next run. The disk that was in drive 1 is held as backup, and another diskette, previously held as backup, can now be used in drive 2 for the new output.

The scheme is showed below, with the three diskettes required labeled A, B and C.

	Drive 1 Input File	Drive 2 Output File	Offline Backup File
First run	EMPLOYEE (A)	EMPLOYEE (B)	EMPLOYEE (C)
Next run	EMPLOYEE (B)	EMPLOYEE (C)	EMPLOYEE (A)
Next run	EMPLOYEE (C)	EMPLOYEE (A)	EMPLOYEE (B)
Next run	(Repeat "first")		

Many flows like these can be designed with sequential files. One of the beauties of sequential file programming is that input and output can be handled in separate files. If anything happens during operation of the program that prevents you from continuing, such as disk I/O errors, power failures or operator errors, you still have your input file. As long as the system is designed so the file(s) used to create another file are retained until the file created is successfully used as an input file itself, automatic backup protection is built into the system. Whenever a problem occurs, you can rerun the program. If the problem occurs because an input file is unreadable, you can go back and rerun the program that created the input file. Of course, along with the advantages, are disadvantages. Since two versions of the same file or files must be online during a program run, the disk storage capacity requirements can be quite demanding.

Sequential File Copy

Copy is the most fundamental operation in programs designed around multiple sequential files. Many sequential file programming schemes are based on copying. Our payroll program example, for instance, copied each employee's data from one file to another. As it did so, it changed and updated certain totals, but the operation was essentially a copy operation.

Let's look at the logic that copies one file to another. COPYSEQ/BAS copies all data from any sequential file to any other. It's up to the operator to supply the name and drive file specifications for both files. To avoid clouding the issue, no error handling is provided in this short program:

```
Ø 'COPYSEQ/BAS
1Ø CLEAR 5ØØ          'CLEAR ENOUGH SPACE FOR SF$,DF$,A$
2Ø LINEINPUT"ENTER SOURCE FILE NAME AND DRIVE:      ";SF$
21 OPEN"I",1,SF$      'OPEN SOURCE FILE IN BUFFER 1
3Ø LINEINPUT"ENTER DESTINATION FILE NAME AND DRIVE: ";DF$
32 OPEN"O",2,DF$      'OPEN DEST   FILE IN BUFFER 2
4Ø IF EOF(1) THEN 8Ø  'IF END OF SOURCE FILE THEN END
5Ø LINE INPUT#1,A$    'ELSE GET NEXT ITEM FROM SOURCE FILE
6Ø PRINT #2,A$        'OUTPUT IT TO DESTINATION FILE
7Ø GOTO 4Ø            'REPEAT FOR NEXT ITEM
8Ø CLOSE 1,2          'CLOSE BOTH FILES
9Ø PRINT"DONE":END    'REPORT THAT PROCESS IS COMPLETED, END
```

To make COPYSEQ into a payroll check writing program, lines 50 and 60 could call subroutines for the input and output of several items of data, and between them, line 55 could be added to call a subroutine that does the calculations and prints a check.

Line 50, for example, might call subroutine 1000. Its function is to read all the data for the next employee in the file.

```
1000 'READ AN EMPLOYEE RECORD
1010 LINE INPUT#1,NA$         'GET EMPLOYEE'S NAME
1020 LINE INPUT#1,AD$(1)      'GET FIRST LINE OF HIS ADDRESS
1021 LINE INPUT#1,AD$(2)      'GET SECOND LINE OF HIS ADDRESS
1022 LINE INPUT#1,A3$(3)      'GET THIRD LINE OF HIS ADDRESS
1050 LINE INPUT#1,SS$         'GET HIS SOCIAL SECURITY NUMBER
1060 LINE INPUT#1,MS$         'GET MARRIED/SINGLE CODE
1070 INPUT#1, HR!             'GET HOURLY RATE
1080 INPUT#1, QG#             'GET GROSS EARNINGS THIS QUARTER
1090 INPUT#1, YG#             'GET GROSS EARNINGS YEAR TO DATE
1100 'Note -- other items might be inputted here
1110 RETURN
```

Line 60 might call subroutine 2000. Its function is to write the employee record to the output file:

```
2000 'WRITE AN EMPLOYEE RECORD
2010 PRINT#1,NA$              'WRITE EMPLOYEE'S NAME
2020 PRINT#1,AD$(1)           'WRITE FIRST LINE OF HIS ADDR.
2021 PRINT#1,AD$(2)           'WRITE SECOND LINE OF HIS ADDR.
2022 PRINT#1,A3$(3)           'WRITE THIRD LINE OF HIS ADDRESS
2050 PRINT#1,SS$              'WRITE HIS SOCIAL SECURITY NO.
2060 PRINT#1,MS$              'WRITE MARRIED/SINGLE CODE
2070 PRINT#1, HR!             'WRITE HOURLY RATE
2080 PRINT#1, QG#             'WRITE GROSS EARNINGS THIS QTR
2090 PRINT#1, YG#             'WRITE GROSS EARNINGS YTD
2100 'Note -- other items might be outputted here
2110 RETURN
```

Line 55 might be a GOSUB 3000. In line 3000 we could provide logic to do the following:

- Display employee's name on the screen for operator verification. If that employee didn't work this week, the operator could press a key, causing a return from the subroutine, so that the unmodified data would be copied to the output file.

- Request the number of hours the employee worked from the operator. Multiply the hours by the rate, stored in HR!, to get the gross pay for the current check.

- Compute the taxes and deductions, based on the gross pay, the married-single code and the earnings to date.

- Print the payroll check for the employee.
- Add totals to the quarter-to-date and the year-to-date variables.
- Return to the main program, where the updated data is copied to the output file.

As you can see, this payroll operation is basically a copy from one file to another, with a few computations and data changes between the input and output.

Logical Records

The group of data items associated with each employee is called a logical record. Sequential files are quite flexible. They let you have logical records of any length, with any number of data items in them. The logical records in a sequential file need not be of a uniform length. If the name and address for one employee is longer than the name and address for the next, there's no problem, and there's no wasted space for the employees with shorter names.

The number of data items from one logical record to the next can be variable also. As an example, let's make an improvement to the payroll program. Suppose you want to keep a history of the checks that have been written to each employee. To make it simple, let's suppose you want to record the date, check number, and net amount. At the end of the quarter you want to print a payment history by employee, and then clear all checks from the file so it doesn't get too full.

The number of data items in each employee record will be variable because some employees will not have been on the job as long as others, and there may be vacation periods when checks are not written for certain employees. When logical records have variable numbers of data items, you have to provide some way to let the system know when the last item has been input. For this payroll example, we will keep a dummy item to signify that the last check history item has been read. That item will be input when a check date is being expected, so let's make it a check date of 99/99/99 to indicate that the last check has been read.

The modification to the system is simple. Assume that, after the current check has been written, DT$ stores the check date, CN! stores the check number, and NA# stores the net amount. If no check was written, let's assume that PD$=“N”. Here's the logic, which could be put in subroutine 4000. Line 65 could be a GOSUB 4000.

```
4000 'ADD CURRENT CHECK TO HISTORY FOR CURRENT EMPLOYEE

4010 LINE INPUT#1,A$           'GET DATE OF CHECK
4011 IF A$="99/99/99"THEN4020  'JUMP IF LAST CHECK'S BEEN READ
4012 PRINT#2,A$                'COPY CHECK DATE TO OUTPUT FILE
4013 FOR X=1TO2                'FOR DATE AND AMOUNT DATA
4014 LINE INPUT#1,A$           'READ...
4015 PRINT#2,A$                'AND COPY TO OUTPUT FILE
4016 NEXT                      'REPEAT
4017 GOTO 4010                 'GO COPY NEXT HISTORICAL CHECK

4020 IF PD$<>"Y"THEN 4040      'SKIP IF NO NEW CHECK THIS TIME
4030 PRINT#2,DT$               'OUTPUT DATE OF CURRENT CHECK
4031 PRINT#2,CN!               'OUTPUT CHECK NUMBER USED
4032 PRINT#2,NA#               'OUTPUT NET AMOUNT PAID
```

```
4Ø4Ø PRINT#2,"99/99/99"        'WRITE END-OF-CHECKS INDICATOR
4Ø5Ø RETURN                    'RETURN TO MAIN PROGRAM
```

Notice that lines 4010 through 4017 just copy the data for any prior checks that may be on file for the employee. That information is of no use to the check writing program, so a working variable, A$, can be used for each data item. Once you've copied the last check to the output file, you are in the proper position to record today's check into the output file. Finally, the program writes the end-of checks indicator for the current employee and returns to the main program, where it will repeat the process until the end of the input file is reached.

As you can see, this process makes the output file longer than the input file, because information about a new check is inserted between each employee record. At the end of each quarter you will want to print out all the employee records. Then you'll want to zero the quarter-to-date totals and delete the check histories. That could be done by a separate program. It would have the following flow:

1. Open input and output files.
2. If end of input file, close both and end.
3. Read next employee's name, address, YTD and QTD amounts, etc., as is done by subroutine 1000.
4. Change quarter-to-date total to 0.
5. Output the same data to the output file, as is done by subroutine 2000.
6. Read and ignore all check data for the employee, until a check date of "99/99/99" is found.
7. Write the "99/99/99" string to the output file.
8. Repeat from step 2.

By not copying the check dates, numbers and amounts in step 6, we (in effect) deleted them from the output file.

Other Two-File Operations

As you can see, an entire payroll system can be built around the fundamental operation of copying one file to another. Since, for many companies, all employees are paid every week in the same sequence, payroll can be viewed as an operation that might be well suited to sequential file programming. The design of a program that makes the operator go through the entire file in sequence also helps to avoid accidentally forgetting to pay an employee. Over the past few years, I've seen several payroll systems that successfully used this two-file approach.

There are some disadvantages to the two-file approach. Any time you want to make a change to the information in any logical record the whole file has to be copied. Say, for example, that an employee's address changes. Your program's got to read and copy the file up to that employee's record, change the information, output the updated record, and copy everything beyond the updated record to the output file.

Adding new records is also a process that involves reading the whole file. Here's how the "add a record" procedure could be arranged. First, the computer operator accumulates data sheets on all the new employees, and arranges those data sheets in alphabetical order.

Now the program starts. The first name is entered, and the program reads and copies all employee records that come before the name alphabetically. Then the other information on the new employee is entered and output. The name for the next new employee is entered, the system copies the files up to that point, and the process is repeated. After the last new employee has been added, the remaining data, if any, from the old payroll file is copied to the new payroll file, and the process is complete.

Employee deletions can also be done. For payroll, though, you need to keep information on all employees who worked until the end of the year, so you can print the required year-end reports. The usual approach in such a situation is to have another data item in each employee record. That item tells whether or not the employee has been terminated. If so, programs such as the check writing program can just copy the terminated employee's data from the input file to the output file each run, without stopping for the operator to make an entry. At the end of the year, a program can automatically zero the year-to-date totals. As it does so, it copies active employees only to the output file. Employees with a termination code of "YES" are read and ignored, so they're, in effect, deleted.

Random Files — Removing the Mysteries

We discussed sequential files in the previous chapter. The other main classification of files that Disk BASIC handles is random files. Random files are also sometimes called "direct" files. The TRS-80 Model II BASIC manual uses that term. For our purposes, both terms describe the same thing and are interchangeable.

The main difference between random files and sequential files is that random files are based on the idea of uniform-length records. Sequential files, as they're handled invisibly and automatically by Disk BASIC and the underlying disk operating system, are based on standard 256-byte records, but as programmers and users, we're hardly aware of it. We look at sequential files as a stream of data items, which may be (but usually are not) of uniform lengths. Each data item is separated from the preceding item by a delimiter of some kind: a comma, a carriage return character, or in the case of numbers, a blank. Because of this, the only practical way to access any data item in a sequential file is to read through all the information that precedes it. With random files we assume that each record in the file is a fixed length: 10 bytes, 32 bytes, or whatever we've decided. Knowing that every item is the same length, our program can compute exactly where to go to get, for instance, the 343rd item or record. Likewise, to record data, our program can compute exactly where to go in the file to write the 688th item.

This brings us to another big distinction between the two types. Once a random file has been opened, we can use it for input or output or both. Within a given program session, for example, we might be extending a random file by recording new records at positions 821 and 823, inquiring into the contents of record 612, and changing the contents of record 311. We can do all these operations without any pre-determined sequence, hence the name "random." Our program goes directly to the records we request by number. That's where we get the alternative name, "direct."

The idea of doing input and output anywhere within a file gives random another advantage over sequential files. You'll remember that, in the payroll example, to change any item of data in the sequential file, you had to copy the whole thing in the process. To make the application work you had to have enough disk space to hold the old version of the file as well as the new version. You don't have that requirement when you want to make changes to the contents of a random file. You save disk space because everything is done in place. The input file and output file are one and the same.

Random files can also result in more efficient use of disk space because compression techniques are more feasible. To illustrate, an integer number that may take up to 6 bytes in a sequential file can be stored in 2 bytes in a random file. This is made possible by the fact that random files don't use delimiters to separate data. All data items are stored based on position within the file and length. Since you don't have to reserve special characters for delimiters, each byte in a data field can have any one of 256 different values. The system doesn't get confused just because one of the bytes used in the compressed representation of a number happens to be of the same value that represents a carriage return, as it would in a sequential file.

To summarize, random files can be more useful than sequential files simply because they give you more control over what will be stored in a disk file, and how.

As we'll see, the added control can make your data storage less vulnerable to operator errors and power failures. Your ability to control the process can make your programs faster and easier to use. That's not to say that sequential files don't have a place in advanced programming. They have very important applications. The important thing is to use the best tools for the task at hand.

Are random files more difficult to learn and use than sequential files? You'll have to be the judge of that. My contention is that, to do very simple things, such as saving a few variables on disk, sequential files are easier to learn and implement. To do things that are just as elementary, random files are easy, but they take a little planning. To do things that are more complex, random files are much easier.

OPEN"R"

The first thing your BASIC program must do to use a random disk file is to OPEN it. The OPEN"R" statement opens a random file. (The TRS-80 Model II can use either OPEN"R" or OPEN"D" for the same purpose. There's no difference. We'll use the OPEN"R" option for compatibility with everyone.)

The format for the OPEN"R" statement is:

```
OPEN"R",BF%,FS$
```

BF% is an integer that specifies the number of the file buffer you wish to use. It may be any number from 1 to 15, depending on how you entered BASIC. If you answered the "How Many Files?" question with 6, BF% may be any number from 1 to 6. If you answered it with 2, BF% may be 1 or 2. If you took the default (by just pressing ENTER), space is reserved for 3 file buffers numbered 1, 2 and 3.

FS$ is a string that provides the file specification or "filespec". The filespec is the 8-character name, optional 3-character extension and optional 8 character password for the file you wish to use. Your constraints on selecting and constructing a filespec to identify a random file are exactly the same as they are with sequential files or any other files that are handled by the disk operating system you are using.

Here are some examples for OPEN commands:

```
OPEN"R",2,"CUSTFILE/DAT:1"
```

. . . opens a file named "CUSTFILE" on disk drive 1, using file buffer 2. Perhaps that file stores, or will store, information about the customers of your company. If the file is already on drive 1, it will now be ready for your BASIC program to use. If it's not there yet, Disk BASIC tells the disk operating system to create a directory entry on the disk in drive 1. After it has taken a second or so to create it, your BASIC program can begin writing and reading data in the file.

```
OPEN"R",1,"MEMBERS"
```

. . . opens a random file named "MEMBERS". The file will use buffer 1. Since we didn't specify a drive, the disk operating system will search through the directory of each disk that is on-line. If it finds the name "MEMBERS", it will open the existing file on the disk where it resides. If it doesn't find any directory entry for "MEMBERS", it will create a directory entry and open the file on drive 0, as long as it's not write-protected.

Here are some other examples of open statements for random files. Remember, you can use variable names or constants for each parameter:

```
OPEN"R",1,"A"
OPEN"R",2,"DF3Ø43/CRS:2"
OPEN"R",1,"SECRETS.LEWIS"
OPEN"R",3,"MASTER/MS:3"
OPEN"R",PF%,A$+":1"
OPEN"R",BF%,"INVEN:"+MID$(STR$(DR%),2)
OPEN"R",BF%,NA$+"/"+EX$+"."+PW$+":"+MID$(STR$(DR%),2)
```

The chapter on sequential files probably gave you a little insight as to the purpose for opening files. To summarize it in one sentence, an OPEN statement relates a specific file name on a specific disk drive to a specific area in memory, for the purpose of handling the transfers of data between your BASIC program and a diskette.

As with sequential files, a file buffer can handle only one file at a time. Before you can do an OPEN, the buffer you specify must be available. If that buffer has already been opened and assigned to a disk file, you cannot open it again, until you close it first. BASIC watches for this potential error and reports "File already open" if you attempt to do so.

There is another option that the OPEN random statement provides. It relates to what are called "variable length records". We'll discuss that later.

The FIELD Statement

We've said that an open statement assigns a specific area in memory to handle the transfer of data between your disk file and your BASIC program. That area of memory is called the file buffer. Each file that is open has its own buffer. For all of the OPEN"R" statements we've illustrated thus far, a file buffer is assigned that has a capacity of 256 bytes. We don't usually care about the exact address of a file buffer in memory. We just specify the file buffer number, and Disk BASIC uses whatever memory address it has assigned for that file.

We've said that a disk file buffer is a waiting area in memory. The FIELD statement gives a way to "point" certain string variables, so the data they contain is the data contained in the disk buffer. To put information into the waiting area, we put information into the strings that are "fielded" into the disk buffer. To get information from the waiting area, we just use the contents of the strings we've "fielded" for the file buffer.

Once a BASIC program has loaded data into the waiting area, you have a command that can record the information contained there to any desired record position in the disk file. If you want to retrieve information from a disk file, there is a command that gets information from any record position in the file. That information arrives in the same waiting area where the BASIC program is free to use it.

To see how this works, let's open a random file on drive 0, using buffer 1. (Unlike sequential files, all statements related to random files may be entered as direct commands without line numbers.)

First, to make sure we are all starting from the same point, enter the following commands:

```
CLOSE
KILL"TESTFILE:Ø"
```

Now you can be sure that buffer 1 is available, and that there is not already a file on drive 0 by the name of "TESTFILE". Enter the open statement next:

```
OPEN"R",1,"TESTFILE:Ø"
```

Now, until you close it, you can access TESTFILE through buffer 1.

Let's say you want to store 3 strings in the first record of TESTFILE. First, you have to point those strings to the file buffer (or waiting area) that has been assigned to the file. To do this, you have to decide how long each string will be. Let's say the first string is a name, NA$, and you want to allow 24 characters for it. The second string is a street address, A1$, and it also is to be up to 24 characters long. The third string, A2$, will hold the city, state, and zip, and you want to allow 28 characters for that.

The field command is:

```
FIELD 1, 24 AS NA$, 24 AS A1$, 28 AS A2$
```

Now, whatever data the first 24 bytes of the file buffer contains, NA$ contains also. Whatever the next 24 bytes of the file buffer contains, A1$ contains that data also. The next 28 bytes of buffer 1 correspond to the contents of A2$. That adds up to 76 bytes in the fielded buffer. The remaining 175 bytes in the buffer are not easily accessible at this point because you haven't given BASIC a way to transfer data to and from that area.

Before you go on, let's clear up some concepts about string storage in memory. As with any variable, BASIC uses a 1 or 2 letter variable name to identify any string. The names used in the example were NA, A1 and A2. The "$" indicator after each identified them as strings, as opposed to integer, single, or double precision variables.

BASIC keeps a list of all variables used in a program, so it can look up the data each variable stores, and change that data when necessary. For string variables, BASIC stores the variable names along with all the others in the list, but it stores the data each string contains elsewhere in memory. Next to each string variable name in its list, BASIC keeps a length indicator and address pointer. The length indicator is a number from 0 to 255 that tells how long the string is. The address pointer is a number from 0 to 65535 that tells where in memory the data for that string starts. All this is invisible to the casual programmer, and BASIC provides no automatic way to see where it has stored the data for any particular string.

The data for all normal string variables is stored in an area near the top of memory, called string storage memory. You control how much string storage memory is available with your CLEAR statement. CLEAR 1000, for example, reserves 1000 bytes of memory for storing any number of strings, as long as their combined lengths don't exceed 1000 bytes. (BASIC also uses this area as a work area when it needs to construct strings according to commands you give it.) You can find more information about string storage in *BASIC Faster and Better and Other Mysteries*.

Now back to fielding. When you give BASIC the command to field a particular file buffer, and you provide a list of string variable names and lengths, you are telling it to use the memory that has been assigned to a file buffer (rather than string storage memory) to store the data that those variables contain. File buffer memory is an entirely different area. It resides just above the area where the disk operating system and BASIC interpreter are stored, and just below the area where your program lines are stored.

LSET

Any string that has been fielded into a file buffer is called a "field variable." In the example, NA$, A1$ and A2$ are field variables. The problem that you encounter now is how to get data into a field variable. Can we just say, NA$="RADIO SHACK" and expect "RADIO SHACK" to be put into the buffer area of memory?

The answer is no! You've got to say LSET NA$="RADIO SHACK". The three commands, (which you can enter now,) to put an address in the buffer for file 1 are:

```
LSET NA$="RADIO SHACK"
LSET A1$="ONE TANDY CENTER"
LSET A2$="FORT WORTH, TEXAS 761Ø2"
```

Let's review what you've done so far. You have opened a random file in buffer 1. You have fielded it to cause 3 strings to overlay the memory area used for the first 76 bytes of the buffer. The field statement also caused BASIC to put three variable names identifying those strings in its variable list. Lastly, you used the LSET command to put data into those strings, which really resulted in putting data into the disk file buffer.

Before you go on, try this:

```
PRINT LEN(NA$),LEN(A1$),LEN(A2$)
```

The display shows 24 24 28. Even though you put data into each variable that was shorter than the lengths you fielded them to hold, the strings remained the lengths you fielded them for.

Now try this statement:

```
PRINT NA$;"X" : PRINT A1$;"X" : PRINT A2$;"X"
```

The display shows:

```
RADIO SHACK                 X
ONE TANDY CENTER            X
FORT WORTH, TEXAS 761Ø2     X
```

The data you intended to put in the strings is there, but blanks have been padded onto each string to make it the length you fielded it for. Here's where LSET is different from an ordinary assignment statement:

```
LSET NA$ = string expression
```

. . . replaces the prior contents of NA$ with whatever series of bytes that string expression represents, but it doesn't change the length of NA$. Instead it puts the string expression in the leftmost side of NA$ and pads the rest with blanks. If the string expression turns out to have a length longer than the length you've fielded NA$ to have, the excess bytes are truncated from the right side to fit it into NA$. In the variable list, the length indicator and address pointer for NA$ is unaltered.

```
NA$ = string expression
```

. . . is entirely different from the LSET statement we described. First of all, the length of NA$ always becomes the length of the string expression that's assigned to it. Secondly, the address pointer stored with NA$ in the variable list is usually changed by such a statement. Usually the address pointer will point to a location up in string storage memory. If the string expression is a constant enclosed in quotes in a line somewhere in the BASIC program, the address pointer for NA$ points to that location within program storage memory.

The important thing to realize, is that if you assign a field variable a value without using LSET (or, as we'll see later, RSET) that variable is de-fielded. It no longer overlays the buffer. It becomes a normal, everyday, string variable. You can't use that variable name to transfer data to and from the buffer again until or unless you execute another field statement listing that variable.

PUT

Now that you have data in the buffer, how do you get it into the disk file? You use the PUT statement to tell Disk BASIC to copy the contents of the buffer to a specific record position in the file on the diskette.

For now, every record in the file is 256 bytes. A PUT statement copies all 256 bytes in the buffer for a particular file into a 256-byte record on the diskette. You specify the record by number. Record 1 is the first 256-byte record, record 2 is the second, and so forth. Let's write the name and address that you've stored in the buffer for file 1 into the disk file you've opened as TESTFILE on drive 0. To store it in record 1 of the file the command is:

```
PUT 1, 1
```

Go ahead and enter the PUT statement. You'll see the light on disk drive 0 come on for a second and then go off. You've recorded the data in the buffer into record 1 of the file. The first "1" in the PUT statement specified the buffer number. The second "1" specified the record number.

Now let's put another name and address in record 2 of the file. First, look at NA$, A1$, and A2$:

```
PRINT NA$, A1$, A2$
```

They still contain the strings that specify the address for Radio Shack. The content of the buffer has not changed, and the variables that were used to field it still are pointed into the buffer. The PUT statement just duplicated the information in memory onto the disk.

Enter the following commands to put another address into the buffer:

```
LSET NA$ = "IJG INC."
LSET A1$ = "126Ø WEST FOOTHILL BLVD"
LSET A2$ = "UPLAND, CALIFORNIA 91786"
```

Now record the new contents of the buffer as record 2 in the file:

```
PUT 1 , 2
```

The disk drive light comes on briefly, and the data is recorded to disk.

GET

The GET statement is the opposite of PUT. It retrieves a 256-byte record from a disk file and copies it into the memory buffer. Any variables that you have fielded into the buffer will contain whatever was in the record on disk.

If you've been following along at your computer keyboard, your field variables, NA$, A1$ and A2$ still contain the address for IJG Inc. Let's recall the address for Radio Shack. Remembering that you stored "Radio Shack" in record 1, the command is:

```
GET 1,1
```

Go ahead and enter it. You've told BASIC to get record 1 from file 1. The first "1" in the statement specifies the file number. The second 1 is the record number you want. The GET command made the light on drive 0 come on briefly while the diskette was accessed.

To see what you've gotten, enter:

```
PRINT NA$ : PRINT A1$ : PRINT A2$
```

The name and address for Radio Shack has been recalled for you. Actually it has been brought back into the buffer. The field variables, NA$, A1$, and A2$, acting like little windows to the buffer, let you see what the buffer contains.

Now enter:

```
GET 1, 2
PRINT NA$ : PRINT A1$ : PRINT A2$
```

The address for IJG has been returned; but looking at it, you notice that you've stored the wrong information. IJG is no longer at 1260 West Foothill Blvd. You should have recorded the current address, 1953 West 11th Street. How do you make the change to the address field, leaving the name and city-state fields unaltered?

You've done the first step by bringing IJG's record into the buffer. Now you want to change the contents of A1$. The statement is:

```
LSET A1$ = "1953 West 11TH STREET"
```

Now when you say:

```
PRINT NA$ : PRINT A1$ : PRINT A2$
```

. . . you see that buffer has the correct current address:

```
IJG INC.
1953 WEST 11TH STREET
UPLAND, CALIFORNIA 91786
```

You've modified the buffer; but, as yet, you haven't recorded the change to disk. The command is:

```
PUT 1, 2
```

You've recorded the change into the second record of the file you opened as TESTFILE on drive 0.

CLOSE

You've stored two names and addresses in the file. Let's say you are done for now. The CLOSE command closes the file. In effect, CLOSE locks in everything you've recorded. Just as in sequential files, the disk operating system has to write an end-of-file pointer in the directory of the diskette containing the file. The file in the example is 2 records long. The CLOSE command records that fact. It also frees the buffer so that it can be used for a different file.

Enter the command:

```
CLOSE 1
```

You see the light for drive 0 come on briefly, and now file 1 has been closed. (Since you didn't have any other files open, all files are closed.) Now that it's no longer open you can't use FIELD, PUT, or GET for file 1 until you re-open TESTFILE or another file in that buffer.

Go ahead and try it:

PUT 1,1 and GET 1,1 result in the error message, "Bad file mode". If you repeat the FIELD command shown above, you get the error message "Bad file number". PUT, GET, and FIELD can be only used for files that have been opened, and currently are open in random mode.

The LOF Function

When you are working with a random file, there are many times when you will need to know statistics about the file. Disk BASIC provides you with certain functions that give you or your program the required information.

The LOF function tells how long a file is. That is, LOF tells the number of the last record in the file. This is important to know for three purposes:

- You might want to know the length of the file you've built so you can judge if the diskette is about to fill up.
- You might want to know the length of the file so you can design a program that gets every record in the file from the first to the last, for the purpose of displaying data from each, making printouts or updating the contents of each record. The LOF function gives the information needed to program a FOR NEXT loop that gets each record, one by one.
- You might want to know the number of the last record in the file, so, if you want to extend the file by adding new records, you know where to start — at the record number just beyond the last.

LOF(1) returns the number of the last record in the file that has been opened as file 1. In general, LOF(BF%) returns the length for any file that has been opened in file buffer BF%. Here are some BASIC statements that show how the information might be used:

```
Ø 'RANDOM1/DEM
1Ø OPEN"R",1,"TESTFILE:Ø"
2Ø PRINT"LENGTH OF THE FILE IS";LOF(1)
3Ø PRINT"NEXT RECORD SHOULD BE ADDED AT";LOF(1)+1
31 IF LOF(1)=Ø THEN 9Ø          'SKIP REMAINDER IF EMPTY
4Ø FIELD 1, 24 AS NA$, 24 AS A1$, 28 AS A2$
5Ø FOR R = 1 TO LOF(1)          'FOR EACH RECORD IN THE FILE
6Ø GET 1,R                      'GET IT
7Ø PRINT NA$                    'PRINT THE NAME FIELD
8Ø NEXT                         'REPEAT UNTIL DONE
9Ø CLOSE : END                  'CLOSE AND END
```

If you've been following the examples so far, go ahead and run this short program. You'll see the following:

```
LENGTH OF THE FILE IS 2
NEXT RECORD SHOULD BE ADDED AT 3
RADIO SHACK
IJG INC
```

By the way, a filename can be on disk and have an LOF of 0. The LOF of a file is zero if:

- It's a new file and no records have been PUT in it yet.
- It's a file that was opened, then closed, and no PUT commands were ever given for it.
- It's a file that was opened, and PUT commands were given, but (because of a power failure, premature removal of the diskette, or an improper return to DOS) the file was never closed. Hence, the statistics for that file were never recorded in the directory.

Notice that in line 31 we provided logic to skip over the FOR NEXT loop if the length of file was zero. This prevents an attempt to get the first record if none exists. (FOR NEXT loops in TRS-80 BASIC's don't allow for 0 iterations. Unless you jump over the loop, a FOR NEXT loop will be executed at least once.)

EOF and a Simplified GET

The EOF function works just like it did for sequential files. If you open a file whose LOF is 0, the EOF is –1, meaning that you are at the end of the file. If you open a file whose LOF is 2, the EOF is 0 after you get the first record, and –1 after you get the second. The –1 condition indicates that your last GET command has retrieved the last record in the file, or it has tried to retrieve a record beyond the end of file.

The EOF function is, like LOF, expressed with the file number between parentheses. EOF(PF%) tells the EOF status (0=False, –1=True) of the file you've opened as file number PF%. You can simply say:

```
IF EOF(PF%) THEN GOTO ....
```

... which is translated, "If you're at the end of the file that's been opened as file PF%, then go to ..."

To see how EOF works, let's write another short program:

```
0 'RANDOM2/DEM
10 OPEN"R",1,"TESTFILE:0"
20 FIELD 1, 24 AS NA$, 24 AS A1$, 28 AS A2$
30 IF EOF(1) THEN 70          'JUMP IF END OF FILE
40 GET 1                      'GET NEXT RECORD IN FILE
50 PRINT NA$                  'PRINT THE NAME FIELD
60 GOTO 30                    'REPEAT FROM LINE 30
70 CLOSE : END                'CLOSE THE FILE AND END
```

Your display should show:

```
RADIO SHACK
IJG INC.
```

Notice the simplified GET command in line 40. If you specify just the file number and don't specify the record number in a GET statement, it is translated to mean: "get the next record in the file." When you open a file, the next record is 1. After any GET or PUT command, the next record is the record that follows the last one gotten or written.

The LOC Function

The LOC function tells your current location in the file after a GET. LOC(1) gives the record number of the last record gotten in file 1. LOC is not available with the BASIC provided with TRSDOS 2.3 on the Model I. It is applicable to Model III BASIC under TRSDOS 1.3. If you are using a different disk operating system or BASIC, try it out to see if you've got the capability. The last program example can be modified by adding line 41 to test and demonstrate LOC:

```
41 PRINT"RECORD #";LOC(1)      'PRINT CURRENT RECORD #
```

Now, when you run the program it gives:

```
RECORD # 1
RADIO SHACK
RECORD # 2
IJG INC.
```

LOC is useful when your program needs to see where it is in a file after a GET. It's also useful after a PUT, but you've got to be aware of the rules. Here are some examples, assuming you've opened a random file in the buffer specified by BF%:

```
GET BF%
```

... gets the record number equal to LOC(BF%) + 1. LOC(BF%) is incremented by 1 after the GET.

```
GET BF%,R%
```

. . . gets record number R%. LOC(BF%) is equal to R% after the get.

```
GET BF%,LOC(BF%)
```

. . . re-reads the last record gotten back into the buffer. This is useful if you've been LSETting data into fields for a data change to a record and you notice you've made a mistake before you do the PUT. It restores the previous contents of the buffer.

```
PUT BF%
```

. . . records the current contents of the buffer into the record number specified by LOC(BF%)+1. After the PUT, LOC(BF%) is incremented by 1. (This is the simplified form of the PUT statement. It corresponds to the simplified version of the GET statement.)

```
PUT BF%,LOC(BF%)
```

. . . records the current contents of the buffer into the record number specified by LOC(BF%). After the put, LOC(BF%) remains the same. This form is very handy. It is used to rewrite the buffer to the same record, usually after you've made a change to some of the data in the buffer.

Storing Numbers — STR$ and RSET

The idea of storing numbers in a random file is sometimes one of the most difficult things for a beginning disk programmer to grasp. There are many ways to store numbers in a random file, and none of the ways need be very hard to understand.

Up through this point we've created a random file that just stores 3 strings in each record: a company name, address, and city-state-zip. Those three strings are only taking 76 bytes of each 256 byte record, so let's use some of the remaining 170 bytes to store some numbers. For a simple example, suppose that you want to keep a record of how many purchases you've made from each company, and the total dollar value of those purchases.

You immediately run into a problem. A FIELD statement only uses strings as field variables. You have to find some way to store the numbers you wish to use as strings. BASIC provides several ways to convert numeric variables to strings. The one you're most familiar with probably is the STR$ function.

For any numeric variable, (A%, A! or A#), STR$(A%), STR$(A!) or STR$(A#) converts that variable into a string. Thus, if NP% equals –12, STR$(NP%) generates the string, "– 12". If NP% equals positive 12, STR$(NP%) generates the string " 12".

The VAL function is the opposite. For any string, A$, VAL(A$) is the numeric value of the string.

Thus,

```
A!=134.43
A$=STR$(A!)
B!=VAL(A$)
```

... takes the number in A!, stores a string representation of it in A$, and converts A$ back into the same number and stores it in the numeric variable B!.

To modify the file, you have to decide just how many digit positions to allow for each numeric field. Let's say that you want 3 positions for the number of purchases. That will give you a range between "–99" and "999". Let's provide 7 positions for the dollar value of those purchases. Assuming you'll need a decimal, that will let you go to "9999.99" or "–999.99". You'll field the number of purchases as NP$, and the total dollar value as DV$.

First, since you are adding new numeric fields to the file, you want to make sure they start out as 0. Type NEW, then enter and run this short program to create the new fields and initialize them to 0.

```
Ø 'RANDOM3/DEM
1Ø OPEN"R",1,"TESTFILE:Ø"
2Ø FIELD 1, 24 AS NA$, 24 AS A1$, 28 AS A2$, 3 AS NP$, 7 AS DV$
3Ø IF LOF(1)=Ø THEN 1ØØ       'SKIP IF FILE IS EMPTY
4Ø FOR R = 1 TO LOF(1)        'FOR EACH RECORD IN THE FILE...
5Ø GET 1,R                    'GET IT
6Ø RSET NP$=STR$(Ø)           'NUMBER OF PURCHASES = Ø
7Ø RSET DV$=STR$(Ø)           'DOLLAR VALUE = Ø
8Ø PUT 1,R                    'RE-WRITE THE RECORD TO DISK
9Ø NEXT                       'REPEAT FOR NEXT RECORD
1ØØ CLOSE : END               'CLOSE AND END
```

Notice lines 60 and 70. We used a new command, RSET. RSET is similar to LSET. It gives you a way to get data into a field variable. The difference is that RSET puts the results of the string expression into the rightmost side of the field and pads or truncates on the left. If the string expression is shorter than the field length, blanks are padded on the left. Thus, after line 60, NP$ contains " 0". (Two blanks and a 0.) After line 70, DV$ contains "   0". (Six blanks and a 0.) You should use RSET in this case because numbers usually look better when they are aligned against the right side of a field or column. You'll probably find very few other uses for RSET in disk programming.

Now let's write a program that lets you make inquiries and changes to the NP$ and DV$ fields. Enter the NEW command to clear program memory, then type in the following lines:

```
Ø 'RANDOM4/DEM
1Ø OPEN"R",1,"TESTFILE:Ø"       'OPEN THE FILE
2Ø FIELD 1, 24 AS NA$, 24 AS A1$, 28 AS A2$, 3 AS NP$, 7 AS DV$
3Ø INPUT"RECORD NUMBER";R       'GET RECORD NUMBER FROM OPERATOR
4Ø IF R<1 THEN 2ØØ              'END IF Ø RECORD NUMBER ENTERED
5Ø IF R>LOF(1)THEN3Ø            'DON'T ALLOW ACCESS PAST EOF
6Ø GET 1,R                      'GET IT
7Ø PRINT NA$                    'PRINT COMPANY NAME
8Ø NP%=VAL(NP$):DV!=VAL(DV$)    'CONVERT FIELDS TO NUMERIC
9Ø PRINTUSING"YOU MADE ### PURCHASES TOTALLING $####.##";NP%;DV!
1ØØ INPUT"CURRENT PURCHASE";CP! 'GET AMOUNT OF THIS PURCHASE
11Ø IFCP!=ØTHEN3Ø               'IF NO PURCHASE, GOTO 3Ø
12Ø NP%=NP%+1                   'INCREMENT NUMBER OF PURCHASES
13Ø DV!=DV!+CP!                 'ADD TO DOLLAR VALUE
14Ø RSET NP$=STR$(NP%)          'PUT NEW NUMBER IN BUFFER
15Ø RSET DV$=STR$(DV!)          'PUT NEW DOLLAR TOTAL IN BUFFER
16Ø PUT 1,R                     'WRITE BUFFER TO DISK
17Ø GOTO 3Ø                     'BACK TO 3Ø FOR ANOTHER RECORD
2ØØ CLOSE : END                 'CLOSE AND END IF DONE
```

After you've typed in the program, run it. Notice that line 50 forces you to enter a record number that is not beyond the end of the file, so you're assured of getting a record with data in it. Line 40 gives you a way to end the program. You just enter "0" in response to the "RECORD NUMBER?" prompt.

When you first request record 1, the following is shown on the video display:

```
RECORD NUMBER? 1
RADIO SHACK
YOU MADE   Ø PURCHASES TOTALLING $   Ø.ØØ
CURRENT PURCHASE?
```

Now you can enter the dollar amount of the item you purchased. If you enter 0, nothing is changed on the disk. The program jumps back to "RECORD NUMBER?" If you enter a dollar amount, the program adds 1 to the number of purchases field and the amount of the current purchase to the dollar value field. The new information is recorded to disk, and you are returned to the "RECORD NUMBER?" prompt. Enter 29.95 for your current purchase. Now when you recall record 1, it shows:

```
RECORD NUMBER? 1
RADIO SHACK
YOU MADE   1 PURCHASES TOTALLING $  29.95
CURRENT PURCHASE?
```

Go ahead and play with the program. You can jump back and forth to any record in your file. When you are done, be sure to enter 0 for the record number to close the file and end properly. Even if you don't close the file there is no danger of losing any data in this program. Because the file is not extended in length, the disk operating system will just be recording the same EOF pointer when the CLOSE is done.

A Better Way to Store Numbers

Using the STR$ and VAL functions is one way to store numbers on disk, but it's usually not the best. In addition to STR$, Disk BASIC provides 3 functions that convert numbers to strings:

```
A$=MKI$(A%)
```

. . . converts the integer number stored in A% to a 2-byte string, A$.

```
A$=MKS$(A!)
```

. . . converts the single precision number stored in A! to a 4-byte string, A$.

```
A$=MKD$(A#)
```

. . . converts the double precision number stored in A# to an 8-byte string, A$.

After any of the statements listed above, A$ contains a compressed representation of a number. If you follow any one of them with a command like:

```
PRINT A$
```

. . . you will not see the number. Instead you might see letters, digits, special symbols, and graphics characters. In its compressed format a number is not printable in any way that makes sense. It is only when you convert it back that you can print it or do arithmetic operations on it.

Disk BASIC gives you 3 functions that convert 2, 4 and 8-byte strings back to numbers:

```
A%=CVI(A$)
```

. . . converts the 2-byte string, A$, (or the 2 leftmost bytes of A$), back to an integer, A%. A$ must be at least 2 bytes long.

```
A!=CVS(A$)
```

. . . converts the 4-byte string, A$, (or the 4 leftmost bytes of A$), back to a single precision number, A!. A$ must be at least 4 bytes long.

```
A#=CVD(A$)
```

. . . converts the 8-byte string, A$, (or the 8 leftmost bytes of A$), back to a double precision number, A#. A$ must be at least 8 bytes long.

Of course you can substitute any variable names into the expressions shown above, as long as the types and lengths are correct.

To see how these concepts apply, let's modify the file so the numbers are stored in compressed format. The number of purchases field, NP$, can now be 2 bytes long, rather than 3. Those 2 bytes can store any integer number, from –32678 to 32767. We'll make the dollar value field, DV$, a 4-byte string in MKS$ format. That will give you the capability to store decimal numbers of up to 6 significant digits. Thus, the range is now –9999.99 to 9999.99.

Since you're changing the way numbers are stored, you have to run a setup program to initialize the number fields to 0 under the new format. You can write the conversion program by making a few modifications to RANDOM3/DEM. Let's call the new program, RANDOM5/DEM:

```
Ø 'RANDOM5/DEM
1Ø OPEN"R",1,"TESTFILE:Ø"
2Ø FIELD 1, 24 AS NA$, 24 AS A1$, 28 AS A2$, 2 AS NP$, 4 AS DV$
3Ø IF LOF(1)=Ø THEN 1ØØ      'SKIP IF FILE IS EMPTY
4Ø FOR R = 1 TO LOF(1)       'FOR EACH RECORD IN THE FILE
5Ø GET 1,R                   'GET IT
6Ø LSET NP$=MKI$(Ø)          'NUMBER OF PURCHASES = Ø
7Ø LSET DV$=MKS$(Ø)          'DOLLAR VALUE = Ø
8Ø PUT 1,R                   'RE-WRITE THE RECORD TO DISK
9Ø NEXT                      'REPEAT FOR NEXT RECORD
1ØØ CLOSE : END              'CLOSE AND END
```

The lines we changed are 20, 60 and 70. Line 20 does the fielding, so we changed the lengths of NP$ and DV$. Lines 60 and 70 put zeros into the number fields. Notice we are using LSET and MKI$ now. Since the MKI$ function of a number always results in a 2-byte string, and we want to put that string into a 2-byte field, it doesn't really matter whether we use LSET or RSET. The result is the same. My rule is, "Always use LSET unless you have a special reason for using RSET." That keeps things simple.

After you've run RANDOM5/DEM, go ahead and modify RANDOM4/DEM into RANDOM6/DEM as shown below:

```
Ø 'RANDOM6/DEM
1Ø OPEN"R",1,"TESTFILE:Ø"      'OPEN THE FILE
2Ø FIELD 1, 24 AS NA$, 24 AS A1$, 28 AS A2$, 2 AS NP$, 4 AS DV$
3Ø INPUT"RECORD NUMBER";R      'GET RECORD NUMBER FROM OPERATOR
4Ø IF R<1 THEN 2ØØ             'END IF Ø RECORD NUMBER ENTERED
5Ø IF R>LOF(1)THEN3Ø           'DON'T ALLOW ACCESS PAST EOF
6Ø GET 1,R                     'GET IT
7Ø PRINT NA$                   'PRINT COMPANY NAME
8Ø NP%=CVI(NP$):DV!=CVS(DV$)   'CONVERT FIELDS TO NUMERIC
9Ø PRINTUSING"YOU MADE ### PURCHASES TOTALLING $####.##";NP%;DV!
1ØØ INPUT"CURRENT PURCHASE";CP! 'GET AMOUNT OF THIS PURCHASE
11Ø IFCP!=ØTHEN3Ø              'IF NO PURCHASE, GOTO 3Ø
12Ø NP%=NP%+1                  'INCREMENT NUMBER OF PURCHASES
13Ø DV!=DV!+CP!                'ADD TO DOLLAR VALUE
14Ø LSET NP$=MKI$(NP%)         'PUT NEW NUMBER IN BUFFER
15Ø LSET DV$=MKS$(DV!)         'PUT NEW DOLLAR TOTAL IN BUFFER
16Ø PUT 1,R                    'WRITE BUFFER TO DISK
17Ø GOTO 3Ø                    'BACK TO 3Ø FOR ANOTHER RECORD
2ØØ CLOSE : END                'CLOSE AND END IF DONE
```

The only lines that have changed are 20, 80, 140 and 150. Line 20 is changed for the new fielding of NP$ and DV$. Line 80 is changed to convert NP$ and DV$ to numeric variables that we can easily work with. Lines 140 and 150 are changed so that the new values, after a purchase, are written into the disk buffer in compressed format.

To Unpack or Not To Unpack

If you are staying at a hotel for a few days, do you unpack your luggage? Or do you leave your luggage packed, and just get what you need when you need it. That's the question we'll ask here, but in regard to compressed numbers in random files.

Notice that, in the RANDOM6/DEM program we used 2 variables for each number. NP$ was used as a field variable, and NP% was used as a numeric variable to identify the same number. DV$ and DV% both contained two representations of the same numeric value. In line 80 we did what's called unpacking. We converted the fielding strings to numeric so that the data would be easy to work with. Lines 140 and 150 did the packing. They converted the numbers back to strings so they could be LSET into their fields in the buffer.

In random file programming, you may find that many times it's easier and faster to only unpack numbers when you need them. You also can save memory because you don't have so many variables floating around. To show this alternative approach, you can delete lines 80, 120, and 130 from RANDOM6/DEM. Then you can change lines 90, 140, and 150 as shown below:

```
9Ø PRINTUSING"YOU MADE ### PURCHASES TOTALLING $####.##";CVI(NP$);CVS(DV$)

14Ø LSET NP$=MKI$(CVI(NP$)+1)     'PUT NEW NUMBER IN BUFFER
15Ø LSET DV$=MKS$(CVS(DV$)+CP!)   'PUT NEW DOLLAR TOTAL IN BUFFER
```

Now the program no longer has the NP% and DV! variables. Notice in particular, line 140, where we add 1 to the 2-byte MKI$ field and put it right back in the buffer, and line 150, where we add the contents of CP! to the DV$ field in the buffer all in one step.

This technique is especially valuable in programs where there are a large number of data fields. To unpack them all when it's likely that only a few will be used can slow down a program. Unless our program is going to do extensive and repetitive calculations on numeric data from the buffer, for greater speed it's often best to let the buffer be our storage area.

Add, Change, Inquiry

Through this point we've showed how to store information in random files, but we haven't, as yet, written a complete program that adds records, calls them back for display on the screen, or allows for changes to any data item in any record.

In this section we'll present a demonstration program that does just that. The program is called RANDOM7/DEM. We'll be using the same file on disk, TESTFILE:0, and data will be stored in the same format:

NAME	24 Bytes	String
ADDRESS	24 Bytes	String
CITY, STATE, ZIP	28 Bytes	String
NUMBER OF PURCHASES	2 Bytes	MKI$ integer
DOLLAR VALUE OF PURCHASES	4 Bytes	MKS$ single precision

The program illustrates some of the problems in making it easy for an operator to make additions and changes. In trying to keep it short, it is not quite as operator friendly as it could be, but everything is prompted on the screen so everything can be done without additional instructions.

The program has three main routines, starting at lines 1000, 2000 and 3000. They are selected by number from a menu that's programmed at line 100. The first routine adds new records to the end of the file. New record numbers are assigned automatically. The second routine recalls any record in the file by number. Once you've recalled a record for display, you can make changes to any of the fields.

The third routine displays all the active records and their contents on the screen. Finally, a fourth selection is provided that closes the file and ends the session.

RANDOM7/DEM is designed so, for the most part, the same logic handles common functions. The same input lines are shared for adding new records and making changes. The same display logic is used in the add, change, inquiry, and list routines. Three main subroutines make this possible.

Subroutine 11000 is the data entry subroutine. Subroutine 12000 displays the current contents of the buffer. Subroutine 13000 allows changes to the current contents of the buffer.

Figure 5.1 — *RANDOM7/DEM*

```
Ø 'RANDOM7/DEM
1 CLEAR 5ØØ                       'CLEAR MEMORY FOR STRINGS
2 CLS: GOSUB 1ØØØØ                'CLEAR SCREEN, GO OPEN THE FILE

1ØØ 'MAIN MENU ROUTINE
1Ø1 CLS : PRINT"RANDOM FILES DEMONSTRATION" : PRINT
11Ø PRINT"<1> ADD NEW RECORDS TO THE FILE"
12Ø PRINT"<2> INQUIRIES AND CHANGES BY RECORD NUMBER"
13Ø PRINT"<3> LIST FILE CONTENTS ON VIDEO DISPLAY"
14Ø PRINT"<4> CLOSE THE FILE AND END"
15Ø PRINT
19Ø LINEINPUT"ENTER THE NUMBER OF YOUR SELECTION: ";A$
191 A%=VAL(A$):IFA%<1ORA%>4THEN1ØØ
192 ON A% GOTO 1ØØØ,2ØØØ,3ØØØ,2ØØ

2ØØ CLS : PRINT"CLOSING FILE"    'TELL OPERATOR
21Ø CLOSE 1                      'CLOSE IT
22Ø END                          'END OF PROGRAM

1ØØØ 'ADD NEW RECORDS AT END OF FILE
1Ø1Ø CLS : OP$="A"                'CLEAR SCREEN, OPTION=ADD
1Ø2Ø R=LOF(1)+1                   'REC NUMBER IS NEXT PAST EOF
1Ø3Ø PRINT"RECORD";R              'SHOW WHAT RECORD WE'LL ADD
1Ø4Ø GOSUB 11ØØØ                  'GO GET DATA FOR BUFFER
1Ø5Ø GOSUB 13ØØØ                  'GO ALLOW CHANGES
1Ø6Ø PRINT"RECORD IT? (Y=YES)    :"; 'ALLOW FINAL ABORT OPTION
1Ø7Ø LINEINPUTA$                  'GET RESPONSE
1Ø8Ø IFA$="Y" THEN PUT 1,R        'WRITE BUFFER TO DISK IF YES
1Ø9Ø PRINT"ADD MORE? (Y=YES)     :"; 'LET OPERATOR QUIT
11ØØ LINEINPUT A$                 'GET DECISION
111Ø IFA$<>"Y"THEN1ØØELSE1ØØØ     'BACK TO MENU OR NEXT ADD

2ØØØ 'INQUIRY BY RECORD NUMBER WITH OPTIONAL CHANGES
2Ø1Ø IF LOF(1)=Ø THEN 1ØØ        'CAN'T INQUIRE IF Ø RECORDS
2Ø11 CLS : PRINT"THERE ARE";LOF(1);"RECORDS IN THE FILE"
2Ø2Ø PRINT"ENTER NUMBER YOU WANT, OR Ø TO RETURN TO THE MENU:"
2Ø3Ø INPUT"RECORD NUMBER: ";R    'GET DESIRED RECORD NUMBER
2Ø4Ø IF R=ØTHEN1ØØ               'RETURN TO MENU IF Ø ENTERED
```

```
2Ø5Ø IF R>LOF(1)THEN2ØØØ         'ENFORCE VALID ENTRY
2Ø6Ø GET 1,R                     'GET IT
2Ø7Ø CF$=""                      'CLEAR CHANGE FLAG
2Ø8Ø GOSUB 13ØØØ                 'GO DISPLAY, ALLOW CHANGES
2Ø9Ø IF CF$="Y"THEN PUT 1,R      'REWRITE IF CHANGES MADE
21ØØ GOTO 2ØØØ                   'GO ALLOW ANOTHER INQUIRY

3ØØØ 'LIST FILE CONTENTS ON THE SCREEN
3Ø1Ø IF LOF(1)=ØTHEN 1ØØ         'CAN'T LIST IF Ø RECORDS
3Ø2Ø CLS                         'CLEAR THE SCREEN
3Ø3Ø FOR R=1 TO LOF(1)           'FOR EACH RECORD IN THE FILE
3Ø4Ø PRINT"RECORD";R             'PRINT RECORD NUMBER
3Ø5Ø GET 1,R                     'GET RECORD INTO BUFFER
3Ø6Ø GOSUB 12ØØØ                 'DISPLAY BUFFER CONTENTS
3Ø7Ø NEXT                        'REPEAT FOR NEXT RECORD
3Ø8Ø LINEINPUT"PRESS <ENTER>";A$ 'STOP WHEN DONE
3Ø9Ø GOTO1ØØ                     'GO BACK TO MENU

1ØØØØ 'OPEN AND FIELD SUBROUTINE
1ØØ1Ø FS$="TESTFILE:Ø"            'STORE FILE NAME IN FS$
1ØØ2Ø PRINT"OPENING FILE: ";FS$   'INFORM OPERATOR
1ØØ3Ø OPEN"R",1,FS$               'OPEN IT
1ØØ4Ø FIELD1,24ASNA$,24ASA1$,28ASA2$,2ASNP$,4ASDV$
1ØØ5Ø RETURN

11ØØØ 'DATA ENTRY SUBROUTINE FOR ADDITIONS & CHANGES
11Ø1Ø LINEINPUT"NAME:           ";A$ 'ENTER THE NAME
11Ø11 LSET NA$=A$                    'STORE IT IN BUFFER
11Ø12 IFOP$="C"THENRETURN            'RETURN IF CHANGE MODE
11Ø2Ø LINEINPUT"ADDRESS:        ";A$ 'ENTER THE ADDRESS
11Ø21 LSET A1$=A$                    'STORE IT IN BUFFER
11Ø22 IFOP$="C"THENRETURN            'RETURN IF CHANGE MODE
11Ø3Ø LINEINPUT"CITY,ST,ZIP:    ";A$ 'ENTER CITY, STATE, ZIP
11Ø31 LSET A2$=A$                    'STORE IT IN BUFFER
11Ø32 IFOP$="C"THENRETURN            'RETURN IF CHANGE MODE
11Ø4Ø LINEINPUT"# PURCHASES:    ";A$ 'GET NUMBER OF PURCHASES
11Ø41 IFABS(VAL(A$))>32767THEN11Ø4Ø 'ENFORCE INTEGER NUMBER
11Ø42 LSET NP$=MKI$(VAL(A$))         'STORE IT IN BUFFER
11Ø43 IFOP$="C"THENRETURN            'RETURN IF CHANGE MODE
11Ø5Ø LINEINPUT"DOLLAR VALUE: ";A$ 'GET TOTAL DOLLAR VALUE
11Ø51 LSET DV$=MKS$(VAL(A$))         'STORE IT IN BUFFER
11Ø52 IFOP$="C"THENRETURN            'RETURN IF CHANGE MODE
11Ø9Ø RETURN                         'RETURN TO CALLING ROUTINE

12ØØØ 'DISPLAY BUFFER SUBROUTINE
12Ø1Ø PRINT"NAME:          ";NA$    'PRINT THE NAME FIELD
12Ø2Ø PRINT"ADDRESS:       ";A1$    'PRINT THE ADDRESS FIELD
12Ø3Ø PRINT"CITY,ST,ZIP: ";A2$      'PRINT THE CITY,STATE FIELD
12Ø4Ø PRINT"PURCHASES:  ";CVI(NP$)'PRINT NUMBER OF PURCHASES
12Ø5Ø PRINT"VALUE:       ";CVS(DV$)'PRINT DOLLAR VALUE
12Ø6Ø PRINT                         'PRINT A BLANK LINE
12Ø7Ø RETURN                        'RETURN TO CALLING ROUTINE
```

```
13ØØØ 'CHANGE BUFFER SUBROUTINE
13Ø1Ø CLS : OP$="C"              'CLEAR SCREEN, OPTION=CHANGE
13Ø2Ø PRINT"RECORD";R            'SHOW WHAT RECORD NUMBER
13Ø3Ø GOSUB12ØØØ                 'SHOW CURRENT BUFFER
13Ø4Ø PRINT"ANY CHANGES? (Y=YES) :"; 'ANY CHANGES TO BE MADE?
13Ø41 LINEINPUTA$                'GET RESPONSE
13Ø42 IFA$<>"Y"THENRETURN        'RETURN IF NO MORE CHANGES
13Ø5Ø PRINT"CHANGE WHAT FIELD?"  'PRINT CHANGE MESSAGE
13Ø51 PRINT"1=NAME  2=ADDR  3=CITY,ST  4=PURCHASES  5=VALUE"
13Ø52 LINEINPUTA$                'GET DECISION
13Ø53 A%=VAL(A$):IFA%<1ORA%>5THEN13Ø51
13Ø6Ø ONA%GOSUB11Ø1Ø,11Ø2Ø,11Ø3Ø,11Ø4Ø,11Ø5Ø
13Ø7Ø CF$="Y"                    'CHANGE FLAG = YES
13Ø8Ø GOTO13ØØØ                  'GO RE-DISPLAY CHANGED DATA
```

Creative Fielding

As we discussed earlier, the FIELD statement uses one or more strings as a bridge between a program and a disk buffer. When a disk record is gotten, the strings fielded into the buffer automatically contain a copy of whatever is the disk record. Since the field statement is a "template" of sorts, based on the relative positions and lengths of the strings fielded into the buffer, the data in the random file maintains a uniform organization from record to record. You simply LSET data into the desired fields, and it goes to the right place in the disk record when the PUT is executed.

When you look at what's really happening with a field statement, you see that BASIC is just changing the length indicators and data address pointers for the string variables you field. Each item in a FIELD statement specifies a field length. The combined lengths of the fields in a fielding statement must not exceed 256 bytes. If they do, BASIC returns a "Field overflow" error.

Creative use of the FIELD statement can give you some special advantages. Before we start our in-depth look at fielding, let's look at the field statement you've been using in the demonstration programs so far:

```
FIELD 1, 24 AS NA$, 24 AS A1$, 28 AS A2$, 2 AS NP$, 4 AS DV$
```

The "1", you'll recall, refers to the file buffer number. Since the examples used only one file, you used buffer 1.

Fielding Bugaboos

The spaces in the field command are optional. You could have said:

```
FIELD1,24ASNA$,24ASA1$,28ASA2$,2ASNP$,4ASDV$
```

Deleting the spaces saves memory, which can be a major consideration in larger programs. There are certain situations, though when you'll need to put a space in here or there. Suppose you wanted to call the first address line AD$, and the city, state field, CS$. You'd have to leave a space before CS$:

```
FIELD1,24ASNA$,24ASAD$,28AS CS$,2ASNP$,4ASDV$
```

Can you see why the space is necessary before any field variable name that starts with “C”? “ASC” is a reserved word in BASIC that is used to identify the ASCII function. There are a few others you have to watch out for. TR$, for example, when it follows AS without a blank, gives ASTR$. STR$ is a reserved word. The result is a syntax error.

The field lengths in the example are constants. You can also use integer variable names to specify the field lengths:

```
LF%=2Ø
FIELD1,LF%ASE$,LF%ASF$,LF%ASG$
```

This statement gives buffer 1 three fields, E$, F$ and G$, of 20 bytes each. It works fine, but if you fail to separate your variable name from the “AS”, you get a type mismatch:

```
LF=2Ø
FIELD1,LFASE$,LFASF$,LFASG$
```

BASIC thinks you are using a string variable, LFASE$, when you should be using a numeric variable. One solution is the following:

```
LF=2Ø
FIELD1,(LF)ASE$,(LF)ASF$,(LF)ASG$
```

If you run into type mismatches or syntax errors in your field statement, you may have to do a little experimenting. These “bugaboos” with fielding catch even quite experienced programmers from time to time, so watch out for them.

String Arrays as Fields

We used NA$, A1$ and A2$ to field the address. One thing to be aware of is that it's perfectly all right to use array variables in fielding statements. Since the lines of an address are often used together, you might get some advantages by fielding the buffer as follows:

```
FIELD1,24ASAD$(1),24ASAD$(2),28ASAD$(3),2ASNP$,4ASNV$
```

Then to print an address from the buffer you would say:

```
FOR X=1 TO 3 : PRINT AD$(X) : NEXT
```

If you need to erase the entire name and address from the buffer, you can say:

```
FOR X=1 TO 3 : LSET AD$(X)="" :NEXT
```

Notice that the LSET AD$(X)=“” has the effect of putting 24 blanks into the first two fields, and 28 blanks into the third.

The use of array variables could provide other benefits. With a little thought, you can design a more efficient and sophisticated add and change routine for RANDOM7/DEM. Even so, the advantages of using an array are marginal in this sample program. On the other hand, in applications where you have many fields, array fields can be quite helpful.

Overlapping Fields

Suppose you have the following two field statements, one after another in a program:

```
1ØØ4Ø FIELD1,24 AS NA$, 24 AS A1$, 28 AS A2$, 2 AS NP$, 4 AS DV$
1ØØ41 FIELD1,76 AS AD$, 6 AS NV$
```

These two statements give overlapping fields, yet, they are perfectly legal. AD$ overlaps NA$, A1$, and A2$. Now, to print the entire address from the buffer in one statement you can say:

```
PRINT AD$
```

To clear the name field, address field, and city-state-zip field in one statement, you can say:

```
LSET AD$=""
```

NV$ overlays both of the numeric fields, NP$ and DV!. To set both them to 0 in one statement, you can say:

```
LSET NV$=STRING$(6,Ø)
```

As you can see, overlapping fields can be quite handy. We'll run into many applications as we go on.

Dummy Fields

A dummy field is one that your program has no use for, other than to align the subsequent field variables to a specific position in the buffer.

In the last example, AD$ could be considered a dummy field if you are only interested in using the NV$ field. AD$, being 76 bytes long, served to align NV$ so it overlayed the numeric portion of your record.

Dummy fields have a more important application. Suppose you want to field a buffer so it contains one 64-byte field and 24 8-byte fields. There are several ways you can do it, but each requires the use of a dummy field.

The first way is the "knocking your head against the wall" approach. You start your fielding statement as usual, but you find it won't fit into one program line:

```
1ØØØ FIELD1,64ASDA$,8ASMT$(1),8ASMT$(2),8ASMT$(3),8ASMT$(4),8ASM
T$(5),8ASMT$(6),8ASMT$(7),8ASMT$(8),8ASMT$(9),8ASMT$(1Ø),8ASMT$(
11),8ASMT$(12),8ASYT$(1),8ASYT$(2),8ASYT$(3),8ASYT$(4),8ASYT$(5)
8ASYT$(6)
```

Now, to get the final 6 fields you can use a dummy field in line 1001. Give the dummy field, DM$, a length of 208 bytes. That is the combined length of all the fields in line 1000. This has the effect of getting the remaining fields into the correct position in the buffer:

```
1ØØ1 FIELD1,2Ø8ASDM$,8ASYT$(7),8ASYT$(8),8ASYT$(9),8ASYT$(1Ø),8A
SYT$(11),8ASYT$(12)
```

Seeing how this dummy field idea works, another, more elegant way to field the same variables comes to mind:

```
1ØØØ FIELD1,64ASDA$
1ØØ2 FORX=ØTO11:FIELD1,64+(X*8)ASDM$,8ASMT$(X+1):NEXT
1ØØ3 FORX=ØTO11:FIELD1,16Ø+(X*8)ASDM$,8ASYT$(X+1):NEXT
```

In line 1000 we take care of the first field, DA$. Then, in line 1002 we use a FOR NEXT loop to field the MT$ array. Each pass through the loop moves the position of the string variable to be fielded 8 bytes to the right. Finally in line 1003 we take care of the YT$ array.

There's another approach. You don't have to give each field in the array a unique string variable to hold it, and you don't have to pre-field the entire buffer. It might be easier to refer to each of those 8-byte fields as TT$. Now whenever you want to use one of the 24 fields in the program, you can load an integer, such as F%, and call a subroutine. The subroutine could be organized like this:

```
1ØØØ FIELD1, 64 AS DA$, (F%-1)*8 AS DM$, 8 AS TT$ : RETURN
```

Let's say that at some point in your program you want to put the contents of A# into the 4th 8-byte field in the buffer. The command is:

```
F%=4 : GOSUB 1ØØØ : LSET TT$=MKD$(A#)
```

Perhaps you want to load M# with the contents of the 18th numeric field:

```
F%=18 : GOSUB 1ØØØ : M#=CVD(TT$)
```

It's a little fancier, isn't it? In some applications, ideas like this can result in speed improvements or memory savings.

Self-Documenting Fields

I learned this technique from a friend of mine who's a genius at making programs look attractive, readable and understandable. This method puts each field on a separate line so you can put a remark statement next to it to identify it. If you don't document anything else in your program, at least explain the layout of your disk file. Here's how it could have been used in the demo program, RANDOM7/DEM:

```
1ØØ4Ø 'FIELD LAYOUT FOR "TESTFILE"

1ØØ41 FIELD1, Ø AS DM$,  24 AS NA$  'NAME             (STRING)
1ØØ42 FIELD1,24 AS DM$,  24 AS A1$  'ADDRESS          (STRING)
1ØØ43 FIELD1,48 AS DM$,  28 AS A2$  'CITY,STATE,ZIP   (STRING)
1ØØ44 FIELD1,76 AS DM$,   2 AS NP$  'NO. OF PURCHASES  (MKI$)
1ØØ45 FIELD1,78 AS DM$,   4 AS DV$  'DOLLAR VALUE      (MKS$)
```

Notice how the numbers are used to field the dummy variable, DM$. Each length of DM$ is the total number of bytes that precede the corresponding field from the beginning of the buffer. Another way of saying it is, each length of DM$ is an offset for the respective field. The effect of this fielding statement is exactly the same as the one used in line 10040 of RANDOM7/DEM, but what a difference!

Three More Fielding Tricks

Sometimes you may be short on memory, and you just don't want to add another variable name, such as DM$, into your program.

Since BASIC only cares about the last position and length of a string variable, you can use the same variable name twice in a field statement. Consider the following example:

```
FIELD1, 1Ø AS A$, 25 AS A$
```

A$ is used twice in the same FIELD statement! First it is used as a dummy to position over to the 11th byte of the buffer, then for the field you actually want: 25 bytes starting at position 11. The effect is the same as:

```
FIELD1, 1Ø AS DM$, 25 AS A$
```

. . . but you don't need to use DM$. I use this idea quite often. You'll see examples of it elsewhere in this book.

Another idea that is sometimes useful is to use POKE to change the length of a string variable that has already been fielded. Here's how it might be done:

```
FIELD1, 1 AS G$, Ø AS M$
```

Notice that M$ is fielded with a length of 0 for now. Later you can change the length of M$ to the number contained in L% with a POKE:

```
POKE VARPTR(M$),L%
```

When the length of a field is conditional, this technique may be more convenient and faster than refielding the entire buffer.

I first used the POKE technique a few years back, when the old version of TRSDOS limited the buffer length to 255 bytes in BASIC. I found that I could access all 256 bytes with a POKE:

```
FIELD 1, 128 AS A1$, 127 AS A2$     'WE'VE FIELDED 255 BYTES
POKE VARPTR(A2$),128                'NOW WE CAN ACCESS ALL 256
```

The final technique we'll look at for now is using the same field statement for more than one buffer. This is useful when the fields for one buffer are the same as those for another. An example is the case of a disk file that's so large you have to store it on two different drives. Suppose, for example, that you need two drives to store all the data for RANDOM7/DEM. You might want to open the file "TESTFILE" on drive 0 and on drive 1. To do it, you would need to open 2 files, "TESTFILE:0" and "TESTFILE:1" using two different file buffers, one for each. You would have to re-write your field statement as a subroutine and make it so the buffer number is a variable:

```
15ØØ FIELD BF%,24ASNA$,24ASA1$,28ASA2$,2ASNP$,2ASDV$ : RETURN
```

Now to get any record in the "file" which spans 2 drives, you would make a computation to determine what drive the record number is on. This would tell you what buffer to use. Load BF% with the desired buffer number, call subroutine 1500 to field the buffer, and get the record. In this case you have got to refield each time you switch from one buffer to the other. It beats using different variable names or array variables because you don't have to complicate the rest of the program.

Records — Physical and Otherwise

The term "physical record" refers to a 256-byte block of data within a random disk file. Each record in the random file demonstration program, RANDOM7/DEM is one physical record in the disk file "TESTFILE". You may remember that we used the term "sector" in the first chapter. As far as we are concerned, "physical record" and "sector" mean the same thing. The important thing to keep in mind is that the underlying operating system does all disk access one physical record at a time.

The term "logical record" as applied to random disk files means something different. It relates to the application at hand, rather than the particular constraints of the hardware or the disk operating system. In application programs, you often need to work with data for a large number of customer accounts, inventory items or individual transactions. In an accounts receivable program, for example, you might want to have a disk file record for each customer's name and address, telephone number and current balance. Quite often, the data for each record will require less than 256 bytes, so you divide the file into "logical records." Suppose, for example, that the information for each customer can be stored in 128 bytes.

Then, if you have 500 customers, you can store the data in 500 logical records, which occupy 250 physical records on a diskette. The first and second logical records will be on physical record 1, the third and fourth will be on physical record 2, and so forth.

The term "subrecord" is used to refer to the logical records in a specific physical record. If you are storing two logical records in each physical record, then each physical record can be said to have two subrecords: subrecord 1 and subrecord 2. The third logical record of the file is subrecord 1 of physical record 2. The fourth logical record of the file is subrecord 2 of physical record 2.

The term "blocking" refers to the number of subrecords in a 256-byte physical record. If you have two 128-byte subrecords per physical record, the blocking factor is 2. If you are storing six 42-byte subrecords in each physical record, the blocking factor would be 6.

"Sector spanning" refers to the concept of a logical record that might start in one physical record and end in the next. If the logical records are 42 bytes long, you can store 6 of them in each physical record, using only the first 252 bytes.

Without sector spanning, 4 bytes in each physical record are unused. With sector spanning, you can start logical record 7 at byte position 253 of physical record 1. On the TRS-80, when we use logical record lengths from 1 to 255 bytes with automatic sector spanning, we call them "variable length records." This term may be misleading. "Variable length" means a length other than 256, but within a random file, all records must be the same length.

How to Compute Physical Records

For now we will talk about the computations needed when you don't have a variable length or sector spanning capability, or when you choose not to use it. These computations can be valuable because they help you to store more data in the same disk space.

If the logical record length is the same as the physical record length, or if the blocking factor is 1, the physical record number is equal to the logical record number.

You can do a GET or PUT without any special computation. Even though the logical record length in the RANDOM7/DEM program was 82 bytes, you only stored one logical record per 256-byte physical record. Thus the blocking factor was 1, and you kept the program simple by avoiding computations.

If you're blocking more than one logical record into each physical record, the program needs to do some math so it will know which physical record to GET or PUT. First, let's see how the blocking factor can be figured. The blocking factor is computed by dividing the physical record length by the logical record length:

```
BLOCKING FACTOR = INT(256/LOGICAL RECORD LENGTH)
```

If you are storing more than one logical record in each physical record, the program needs to know what physical record to GET or PUT to retrieve or store the desired logical record. The required physical record number is computed with the following formula, where "PR" stands for "physical record", "LR" means "logical record", and "BF" is the blocking factor:

```
PR = INT(( LR - 1 )/BF ) + 1
```

Once you've computed the physical record number, it is necessary to know what subrecord contains the desired logical record. The subrecord number, "SR", can be computed as follows:

```
SR = LR - ((PR-1) * BF )
```

The subrecord computation becomes important when you want to FIELD the buffer so the desired logical record is available to your program. Let's suppose, for example, that you have a file that stores product descriptions and prices. Each product description requires 81 bytes, and each price requires 4 bytes. The logical record length is 81 + 4, or 85 bytes. The blocking factor is INT(256/85), or 3 logical records per physical record.

If you can afford to be inefficient, storing only one logical record per physical record, the fielding command would be:

```
FIELD 1, 81 AS D$, 4 AS P$
```

This compares to the "inefficient" way we handled the data in the RANDOM7/DEM program. On the other hand, if you prefer to be efficient and want to store 3 logical records per physical record, you can use the following FIELD command:

```
FIELD 1, 81 AS D$(1), 4 AS P$(1), 81 AS D$(2), 4 AS P$(2), 81 AS D$(3),4 AS P$(3)
```

As you can see, the field variables are specified as arrays. Each field variable is an array. Each element of each array corresponds to a subrecord within the buffer.

Now, by computing the physical record number from the logical record number and blocking factor, you GET the proper physical record. By computing the subrecord number, you know whether to use D$(1), D$(2) or D$(3) to access the description, and P$(1), P$(2) or P$(3) to access the price.

This approach works fine if you have only a few fields per logical record and not too many subrecords per physical record. A better approach, and one that works well in all situations, is to use a dummy field. Compute the physical record using the same formula. Then, compute the subrecord number and subtract 1. This gives you the number of subrecords that precede the desired subrecord. Multiplying the number of preceding subrecords by the logical record length gives you the length for your dummy field. The field command in this case is:

```
FIELD 1,(SR-1)*85 AS DM$, 81 AS D$, 4 AS P$
```

The dummy field, DM$, "offsets" the location of D$ and P$ within the buffer, depending upon which subrecord you want to access. If, for example, the desired subrecord is computed as 1, then (SR-1)*85 is 0, so the length of DM$ is 0, and D$ starts at the first byte of the buffer. If the desired subrecord is computed as 3, then (SR-1)*85 gives 170, and D$ will be pointed to the 171st byte of the disk buffer.

Three Times Better

With the concepts discussed in the last few sections, you can make the demonstration program three times better. Instead of storing one 82-byte record in each physical record of the file, as RANDOM7/DEM did, store three 85-byte logical records. Why 85-byte records instead of 82? A record length of 85 bytes is very efficient. If you want 86-byte records you can only store 2 per physical record. You can store 3 with 82 or 85 byte records, so you might as well use 85.

To the operator the RANDOM8/DEM looks the same as RANDOM7/DEM. Internally there are some major differences:

- The field variable names are the same, but some additional computations are made and a dummy is used to position the fields over the proper subrecord.
- Instead of doing it just once at the beginning of the program, fielding will be done every time you go from one logical record to another.
- Whenever you want to PUT a different physical record, you must have done a GET first. This ensures that the contents of the subrecords you are not working with will remain unaltered.

Subroutine 14000 provides most of the logic to make these changes. Upon entry to 14000, R contains the number of the logical record to get. Based on the desired record number, R, the blocking factor, 3, and the record length, 85, the subroutine computes the required physical record, PR%, and the subrecord, SR%. If physical record PR% is not already in the buffer, a GET is executed. Finally, the buffer is fielded.

One of the problems with storing more than one logical record per physical record is that, although you know the LOF of the file, upon starting up the program you don't know how many subrecords are active in the last physical record. To illustrate, if you have 3 subrecords per physical record, and upon starting up the LOF is 10, you know that as many as 30 records are active, and no fewer than 28 active. That last physical record may contain 1, 2 or 3 active subrecords.

RANDOM8/DEM figures out the number of records active by first taking the LOF, subtracting 1, and multiplying by the blocking factor (3). It holds this number as N. Then it takes advantage of the fact that anytime Disk BASIC gets a record beyond the end of file, the buffer is filled with hexadecimal zeros. Since this is the case, it knows that the first byte of the first available subrecord, if any, in the last physical record, will have an ASCII value of zero. It counts how many subrecords in the last physical record are non-zero and adds the count to N. N is the number of active records in the file. All this takes place in lines 10030 through 10070, which are executed just after the open.

The complete listing for RANDOM8/DEM is shown on the next few pages. The lines that have been changed or added to RANDOM7/DEM are 0, 1020, 1035, 1080, 2011, 2050, 2060, 2090, 3010, 3030, 3050, 10000, 10015, 10030 through 10070, and 14000 through 14060.

Figure 5.2 — *RANDOM8/DEM**

```
Ø 'RANDOM8/DEM
1 CLEAR 5ØØ                      'CLEAR MEMORY FOR STRINGS
2 CLS: GOSUB 1ØØØØ               'CLEAR SCREEN, GO OPEN THE FILE

1ØØ 'MAIN MENU ROUTINE
1Ø1 CLS : PRINT"RANDOM FILES DEMONSTRATION" : PRINT
11Ø PRINT"<1> ADD NEW RECORDS TO THE FILE"
12Ø PRINT"<2> INQUIRIES AND CHANGES BY RECORD NUMBER"
13Ø PRINT"<3> LIST FILE CONTENTS ON VIDEO DISPLAY"
14Ø PRINT"<4> CLOSE THE FILE AND END"
15Ø PRINT
```

*Note: Not for use with Model II TRSDOS. See discussion of variable length record files beginning on page 157.

```
19Ø LINEINPUT"ENTER THE NUMBER OF YOUR SELECTION: ";A$
191 A%=VAL(A$):IFA%<1ORA%>4THEN1ØØ
192 ON A% GOTO 1ØØØ,2ØØØ,3ØØØ,2ØØ

2ØØ CLS : PRINT"CLOSING FILE"   'TELL OPERATOR
21Ø CLOSE 1                     'CLOSE IT
22Ø END                         'END OF PROGRAM

1ØØØ 'ADD NEW RECORDS AT END OF FILE
1Ø1Ø CLS : OP$="A"                'CLEAR SCREEN, OPTION=ADD
1Ø2Ø R=N+1                        'REC NUMBER IS NEXT PAST EOF
1Ø3Ø PRINT"RECORD";R              'SHOW WHAT RECORD WE'LL ADD
1Ø35 GOSUB 14ØØØ                  'GO GET RECORD AND FIELD
1Ø4Ø GOSUB 11ØØØ                  'GO GET DATA FOR BUFFER
1Ø5Ø GOSUB 13ØØØ                  'GO ALLOW CHANGES
1Ø6Ø PRINT"RECORD IT? (Y=YES)   :"; 'ALLOW FINAL ABORT OPTION
1Ø7Ø LINEINPUTA$                  'GET RESPONSE
1Ø8Ø IFA$="Y" THEN PUT 1,PR% :N=R  'WRITE BUFFER, UPDATE COUNT
1Ø9Ø PRINT"ADD MORE? (Y=YES)    :"; 'LET OPERATOR QUIT
11ØØ LINEINPUT A$                 'GET DECISION
111Ø IFA$<>"Y"THEN1ØØELSE1ØØØ     'BACK TO MENU OR NEXT ADD

2ØØØ 'INQUIRY BY RECORD NUMBER WITH OPTIONAL CHANGES
2Ø1Ø IF LOF(1)=Ø THEN 1ØØ         'CAN'T INQUIRE IF Ø RECORDS
2Ø11 CLS : PRINT"THERE ARE";N;"RECORDS IN THE FILE"
2Ø2Ø PRINT"ENTER NUMBER YOU WANT, OR Ø TO RETURN TO THE MENU:"
2Ø3Ø INPUT"RECORD NUMBER: ";R   'GET DESIRED RECORD NUMBER
2Ø4Ø IF R=ØTHEN1ØØ               'RETURN TO MENU IF Ø ENTERED
2Ø5Ø IF R>NTHEN2ØØØ             'ENFORCE VALID ENTRY
2Ø6Ø GOSUB14ØØØ                  'GO GET AND FIELD IT
2Ø7Ø CF$=""                     'CLEAR CHANGE FLAG
2Ø8Ø GOSUB 13ØØØ                 'GO DISPLAY, ALLOW CHANGES
2Ø9Ø IF CF$="Y"THEN PUT 1,PR%    'REWRITE IF CHANGES MADE
21ØØ GOTO 2ØØØ                   'GO ALLOW ANOTHER INQUIRY

3ØØØ 'LIST FILE CONTENTS ON THE SCREEN
3Ø1Ø IF N=ØTHEN 1ØØ              'CAN'T LIST IF Ø RECORDS
3Ø2Ø CLS                         'CLEAR THE SCREEN
3Ø3Ø FOR R=1 TO N                'FOR EACH RECORD IN THE FILE
3Ø4Ø PRINT"RECORD";R             'PRINT RECORD NUMBER
3Ø5Ø GOSUB 14ØØØ                 'GET RECORD, FIELD BUFFER
3Ø6Ø GOSUB 12ØØØ                 'DISPLAY BUFFER CONTENTS
3Ø7Ø NEXT                        'REPEAT FOR NEXT RECORD
3Ø8Ø LINEINPUT"PRESS <ENTER>";A$ 'STOP WHEN DONE
3Ø9Ø GOTO1ØØ                     'GO BACK TO MENU

1ØØØØ 'OPEN AND COMPUTE NUMBER OF RECORDS ACTIVE
1ØØ1Ø FS$="TESTFILE:Ø"           'STORE FILE NAME IN FS$
1ØØ15 BL%=3:RL%=85               'BLOCKING=3, REC LENGTH=85
1ØØ2Ø PRINT"OPENING FILE: ";FS$  'INFORM OPERATOR
1ØØ3Ø OPEN"R",1,FS$              'OPEN IT
1ØØ4Ø IF LOF(1)=Ø THEN N=Ø:RETURN 'RETURN IF FILE LENGTH = Ø
```

```
1ØØ6Ø N=(LOF(1)-1)*BL%             'COMPUTE NUMBER OF RECORDS
1ØØ61 GET1,LOF(1)                  'GET LAST PHYSICAL RECORD
1ØØ62 FORA%=ØTOBL%-1               'FOR EACH SUBRECORD
1ØØ63 FIELD1,RL%*A%ASA$,1ASA$      'A$=FIRST BYTE OF SUBRECORD
1ØØ64 IFASC(A$)>Ø THEN N=N+1       'ADD 1 TO N IF SUBREC ACTIVE
1ØØ65 NEXT                         'REPEAT FOR NEXT SUBRECORD
1ØØ7Ø RETURN                       'RETURN, N=NUMBER OF RECORDS

11ØØØ 'DATA ENTRY SUBROUTINE FOR ADDITIONS & CHANGES
11Ø1Ø LINEINPUT"NAME:         ";A$ 'ENTER THE NAME
11Ø11 LSET NA$=A$                  'STORE IT IN BUFFER
11Ø12 IFOP$="C"THENRETURN          'RETURN IF CHANGE MODE
11Ø2Ø LINEINPUT"ADDRESS:      ";A$ 'ENTER THE ADDRESS
11Ø21 LSET A1$=A$                  'STORE IT IN BUFFER
11Ø22 IFOP$="C"THENRETURN          'RETURN IF CHANGE MODE
11Ø3Ø LINEINPUT"CITY,ST,ZIP:  ";A$ 'ENTER CITY, STATE, ZIP
11Ø31 LSET A2$=A$                  'STORE IT IN BUFFER
11Ø32 IFOP$="C"THENRETURN          'RETURN IF CHANGE MODE
11Ø4Ø LINEINPUT"# PURCHASES:  ";A$ 'GET NUMBER OF PURCHASES
11Ø41 IFABS(VAL(A$))>32767THEN11Ø4Ø 'ENFORCE INTEGER NUMBER
11Ø42 LSET NP$=MKI$(VAL(A$))       'STORE IT IN BUFFER
11Ø43 IFOP$="C"THENRETURN          'RETURN IF CHANGE MODE
11Ø5Ø LINEINPUT"DOLLAR VALUE: ";A$ 'GET TOTAL DOLLAR VALUE
11Ø51 LSET DV$=MKS$(VAL(A$))       'STORE IT IN BUFFER
11Ø52 IFOP$="C"THENRETURN          'RETURN IF CHANGE MODE
11Ø9Ø RETURN                       'RETURN TO CALLING ROUTINE

12ØØØ 'DISPLAY BUFFER SUBROUTINE
12Ø1Ø PRINT"NAME:       ";NA$     'PRINT THE NAME FIELD
12Ø2Ø PRINT"ADDRESS:    ";A1$     'PRINT THE ADDRESS FIELD
12Ø3Ø PRINT"CITY,ST,ZIP: ";A2$     'PRINT THE CITY,STATE FIELD
12Ø4Ø PRINT"PURCHASES:  ";CVI(NP$)'PRINT NUMBER OF PURCHASES
12Ø5Ø PRINT"VALUE:      ";CVS(DV$)'PRINT DOLLAR VALUE
12Ø6Ø PRINT                       'PRINT A BLANK LINE
12Ø7Ø RETURN                      'RETURN TO CALLING ROUTINE

13ØØØ 'CHANGE BUFFER SUBROUTINE
13Ø1Ø CLS : OP$="C"               'CLEAR SCREEN, OPTION=CHANGE
13Ø2Ø PRINT"RECORD";R             'SHOW WHAT RECORD NUMBER
13Ø3Ø GOSUB12ØØØ                  'SHOW CURRENT BUFFER
13Ø4Ø PRINT"ANY CHANGES? (Y=YES) :"; 'ANY CHANGES TO BE MADE?
13Ø41 LINEINPUTA$                 'GET RESPONSE
13Ø42 IFA$<>"Y"THENRETURN         'RETURN IF NO MORE CHANGES
13Ø5Ø PRINT"CHANGE WHAT FIELD?"   'PRINT CHANGE MESSAGE
13Ø51 PRINT"1=NAME   2=ADDR   3=CITY,ST   4=PURCHASES   5=VALUE"
13Ø52 LINEINPUTA$                 'GET DECISION
13Ø53 A%=VAL(A$):IFA%<1ORA%>5THEN13Ø51
13Ø6Ø ONA%GOSUB11Ø1Ø,11Ø2Ø,11Ø3Ø,11Ø4Ø,11Ø5Ø
13Ø7Ø CF$="Y"                     'CHANGE FLAG = YES
13Ø8Ø GOTO13ØØØ                   'GO RE-DISPLAY CHANGED DATA
```

```
14000 'FIELD AND GET FOR RECORD R
14010 A%=PR%                    'HOLD OLD PHYSICAL RECORD
14020 PR%=INT((R-1)/BL%)+1      'COMPUTE NEW PHYSICAL RECORD
14030 SR%=R-((PR%-1)*BL%)       'COMPUTE SUBRECORD NUMBER
14040 IF PR%<>A% THEN GET1,PR%  'GET IF NEW PHYSICAL RECORD
14050 FIELD1,(SR%-1)*RL%ASDM$,24ASNA$,24ASA1$,28ASA2$,2ASNP$,4AS
DV$
14060 RETURN                    'RETURN TO CALLING ROUTINE
```

Logical Record Planning

When you sit down to design a computer-based system, there are several questions that must go through your mind:

- What is the purpose of the system?
- How will it be used?
- What information needs to be maintained in the computer?
- What information items group together logically?
- What are the relationships among these logical groups?
- How will data be gathered and input?
- How can each group be identified and accessed?
- What redundancies and cross checks need be built in?
- What other systems must be interfaced with?
- What capacities are required? Do we have enough?
- How soon do we need it?
- What's most important? What can be compromised?

This is not a book on system design. Rather, it's a book of tools and techniques, alternatives, and ideas that you can put to use when you design a system. At any rate, at some point you will inevitably come up with one or more lists of data items you want your computer to keep track of for you.

Those lists will probably become files. The data items will become fields in the records that the files contain. Your decisions about how the data will be stored in each field will depend on your disk capacity and the programming techniques you have at your disposal. We'll look at some of the ways you can store data in fields, and when you might use them. In other chapters of this book, you'll see why multiple files are often required, and how they can be programmed to relate to one another.

Random File Data Formats

The most common storage methods, and the field lengths required for them are listed below. For each type of data storage, the method for transferring to and from the disk buffer is shown. (A$, A%, A! and A# represent BASIC variables of any name. F$ represents any string variable that you've fielded into a disk buffer.)

```
Type:         Alphanumeric
Description:  Letters, numbers, and special characters.
```

```
Length:        1 to 255 bytes.
Storage:       LSET F$ = A$
Retrieval:     A$ = F$
Typical Uses:  Names, addresses, descriptions, inventory codes,
               status codes, just about anything.

Type:          Integer
Description:   Whole numbers ranging from -32768 to 32767
Length:        2 bytes.
Storage:       LSET F$ = MKI$(A%)
Retrieval:     A% = CVI(F$)
Typical Uses:  Quantities, (as for counts of inventory and
               sales.)  Pointers, (numbers that reference disk
               file records, allowing your program to follow a
               path from one record to another, in the same
               file or a different file.)

Type:          Single Precision
Description:   Decimal numbers up to 6 significant digits.
Length:        4 bytes.
Storage:       LSET F$ = MKS$(A!)
Retrieval:     A! = CVS(F$)
Typical Uses:  Quantities that require a decimal, or that may
               may range up to 999999. Prices, costs, and
               dollar amounts that may range up to $9999.99.

Type:          Double Precision
Description:   Decimal numbers up to 16 significant digits.
Length:        8 bytes.
Storage:       LSET F$ = MKD$(A#)
Retrieval:     A# = CVD(F$)
Typical Uses:  Dollar amounts and balances that may be quite
               large. Quantities that must be very large or
               small, and yet very precise. Phone numbers.
```

In addition to the common storage methods, my book *BASIC Faster and Better and Other Mysteries* tells other ways you can create strings that can be LSET into data fields:

Here are some of them:

```
Type:          Small integer
Description:   Unsigned integer from Ø to 255.
Length:        1 byte.
Storage:       LSET F$ = CHR$(A%)
Retrieval:     A% = ASC (F$)
Typical Uses:  Small quantities, such as the number of
               employee deductions in a payroll record.
               Transaction type or record type codes. Status
               codes.

Type:          Unsigned integer
Description:   Unsigned integer from Ø to 65535.
Length:        2 bytes.
```

```
Storage:       LSET F$ = MKI$ (FNSI%(A!))
Retrieval:     A! = FNIS!(CVI(F$))
Typical Uses:  Inventory and sales quantities that need not be
               negative. Record pointers for very large files.

Type:          2-byte date
Description    An 8-byte date "MM/DD/YY" stored in 2 bytes
Length:        2 bytes.
Storage:       LSET F$ = FNC2$(FNCD$(A$))
Retrieval:     A$ = FNUD$(FNU2$(F$))
Typical Uses:  Any situation where you want to store a date:
               Transaction dates, expiration dates, birth dates,
               and, to know how current a record is, "date last
               updated".

Type:          Compressed alphanumeric
Description:   A string compressed by 33% with the COMUNCOM USR
               routine, and the FNKM$ function.
Length:        2 bytes for every 3 bytes of the uncompressed string.
Storage:       LSET F$ = FNKM$(A$,1)
Retrieval:     A$ = FNKM$(F$,2)
Typical Uses:  Same as regular alphanumeric, but when storage
               space on disk is limited. Note, this format
               allows only the uppercase letters A-Z, digits
               Ø-9, and the special characters comma, period,
               dash, and blank.

Type:          Compressed numeric
Description:   Signed or unsigned numbers compressed
               into strings with various methods.
Length:        3 or 4 bytes
Storage:       LSET F$ = FNU3$(A#)
Retrieval:     A# = FNU3#(F$)
Typical Uses:  Large numbers when disk space is at a premium.
               When a decimal is required, the number may be
               stored as a whole number and the decimal place
               assumed.
```

Deciding the Record Layouts

Once you've determined what data fields you want to store in each logical record, what methods you'll use to store them, and how many bytes each will require, you need to decide the sequence in which the fields will be stored within each record. You also will need to decide on the string variable names to use when fielding each record.

Once you've designed your logical record, you may wish to make a few adjustments to it to get the most efficient disk storage.

Let's suppose, for example, that your logical record length comes to 100 bytes after you add the lengths of each field. You can block two 100-byte logical records into a 256-byte physical record, but 56 bytes are wasted. In this situation, you are wise to make your logical record length 128 by adding a few fields, or adding a dummy field. The extra 28 bytes for each logical record cost you nothing!

You may wish to divide the 100-byte logical record, storing part of it in another file to get more efficient blocking. Disk storage efficiency would be improved by using 2 files, one with a 15-byte logical record and the other with an 85-byte logical record, or perhaps one file with a 64-byte record and the other with a 36-byte logical record. Your trade-off for disk storage efficiency is access speed. You will, if you divide the record, need two GETs instead of one, and 2 PUTs instead of one.

When designing logical record lengths, shoot for even factors of 256. When you divide the logical record length into 256, the remainder is the number of bytes wasted per physical record. Appendix 5 in *BASIC Faster and Better and Other Mysteries* gives you a list of the divisors of 256.

Those logical record lengths it highlights with "**" are the best. They include 1, 2, 4, 8, 16, 64 and 128. Those with "*" are fairly good. They include most lengths less than 19, as well as 21, 23, 25, 28, 36, 42, 51 and 85. The lower the remainder is, the more efficient the logical record length.

Variable Length Records

A variable length record capability allows you to set up a random file in which the records are each a uniform length of anywhere from 1 to 256 bytes. The length you select is called your logical record length. As far as you are concerned, the capacity of your buffer is one logical record. Your field statements are limited to your logical record length, and your GETs and PUTs handle one logical record at a time. You have no concerns about subrecords, physical records or blocking.

Those problems are all handled by the system. Though the operating system still does its disk reads and writes 256 bytes at a time, you need not be aware of it. Sometimes one logical record will span two physical records. To get it, the DOS has to do two GETs. As your program in disk BASIC sees it, it is all automatic.

The variable length capability for random files is available with TRSDOS 1.3 on the Model III. On TRSDOS 2.3 for the Model I, a variable length capability appears to have been built in, but it is not fully implemented. Attempts to use it will result in errors. Most of the other implementations of Disk BASIC for TRS-80 computers, available with disk operating systems from other vendors, have the variable length capability.

To provide for variable length records, the system requires a larger work space for each file buffer. Internally it has to hold the contents of 2 physical records at a time so it can work with logical records that happen to cross from one physical record to the next. This space is in addition to the memory area that is used for the buffer which you can field from BASIC. The result is that each file you specify upon entering BASIC takes 512 additional bytes. Instead of requiring 360 bytes per file, 872 bytes are required. As you can see, the variable length record capability is costly in memory storage. This can be a factor when you have a large program that requires many files open simultaneously.

Because the memory required per file is different, this affects the addresses of everything beyond the file buffer areas. The program storage area, for example, must start at a higher address. Consequently, it's necessary to specify whether or not you will be using variable length files when you enter BASIC. When you enter BASIC from DOS Ready, and you intend to use the variable length file capability for at least one of the files you will be opening, you put a "V" after your response to the "How Many Files" question. Here's how it looks:

```
How Many Files? 3V
Memory Size?
```

```
TRS-8Ø Model III Disk BASIC Rev 1.3
(c)(p) 198Ø by Tandy Corp. All Rights Reserved.
Created 5-Jul-8Ø
37,178 Free Bytes   3 Variable Files
Ready
>
```

In this case, we said 3V. That response indicates that, at most, we intend to have 3 files open simultaneously (in buffers 1, 2 and 3), and one or more of them can be a variable length record random file. Notice that it doesn't give you the chance to say, "I want 1 variable record length file and 2 regular files." To keep things simple, it reserves enough space so that any or all of your files can be variable length.

The same entry to BASIC from DOS Ready could have been done in one statement. We enter the short-cut command as usual, but put a "V" after the how many files specification:

```
TRSDOS Ready
BASIC -F:3V
```

If you are using Model II TRSDOS it is important to know that your version of BASIC always acts as if the "V" option is active. In effect, all files use variable length records.

Once you are in BASIC, there is really only one major change in your programming. Your OPEN"R" statement specifies the logical record length as the last parameter:

```
OPEN"R",1,"DATAFILE/TXT",25
```

. . . opens a file named "DATAFILE/TXT" and assigns a logical record length of 25 bytes per record.

```
OPEN"R",BF%,FS$,RL%
```

. . . is the general command format that opens a variable length record file. BF% is the file buffer number. FS$ is the file name and drive, and RL% is the desired record length. RL% may be from 0 to 255. If it is 0, or if RL% is omitted, a logical record length of 256 bytes is assumed, making it like a non-variable length record random file.

There are only a few other differences you need to keep in mind:

- The total length of the items in a field statement must not exceed the record length you specified for the file. A "Field Overflow" error, code 51, results.
- A GET statement results in an "Internal error", error code 52, if you attempt to GET a record number that is higher than the LOF of the file. (With normal random files, a GET past the LOF does not result in an error, and it serves to fill the buffer with zeros. This is not the case with variable files.)
- The LOF function returns the number of the last logical record, not the last 256-byte physical record.

Variable Length Demonstration

In the absence of a variable length record capability, we had to add a lot of logic and fancy fielding to RANDOM7/DEM to make it store 3 logical records per physical record. The result was the RANDOM8/DEM program. If you enter BASIC with the "V" option, you need to make only one change to the RANDOM7/DEM program to make it store 3 logical records per physical record.

You'll recall that in RANDOM7/DEM you were fielding 82 bytes of data. Thus, 82 bytes is the desired logical record length. Dividing 256 by 82 tells you that 3.12195 logical records can be stored in each physical record. The first physical record will store the first 3 logical records plus the first 10 bytes of the 4th logical record.

The only change to RANDOM7/DEM is:

```
10030 OPEN"R",1,FS$,82          'OPEN IT, 82 BYTES PER RECORD
```

Of course, now that you've changed the logical record length of TESTFILE, the data already created in it will be misaligned. You can either kill the old "TESTFILE" and re-enter the data, or write a short temporary conversion program:

```
10 CLEAR 500
20 OPEN"R",1,"TESTFILE:0"        'OPEN OLD FILE, 256 BYTE RECS
30 OPEN"R",2,"TESTFILE/NEW:0",82 'OPEN NEW FILE, 82 BYTE RECS
40 FIELD 1,82 AS A1$             'FIELD OLD FILE
50 FIELD 2,82 AS A2$             'FIELD NEW FILE
60 FORX=1TOLOF(1)                'FOR EACH RECORD IN OLD FILE
70 GET1,X                        'GET OLD RECORD
80 LSET A2$=A1$                  'COPY DATA FROM OLD TO NEW
90 PUT2,X                        'PUT NEW RECORD
100 NEXT                         'REPEAT FOR ALL RECORDS
110 CLOSE                        'CLOSE BOTH FILES
```

After this conversion is done, you can go to TRSDOS Ready and do the following:

```
KILL TESTFILE:0
RENAME TESTFILE/NEW:0 TESTFILE
```

There are pros and cons for the use of variable length records:

On the plus side:

- Programming in BASIC is simplified.
- Disk space may be used more efficiently.

On the minus side:

- More memory overhead is required. You have less space for programs and variables.
- Your programs may be incompatible from one version of BASIC to another.
- Reliability and error handling are more critical.

The Random Disk File Handler

The random disk file handler is a set of subroutines that you can merge into any BASIC program in which you wish to use one or more random access files. As with any handler, it is designed to be modified according to the requirements of your application. I think you'll find that the standard procedures will save quite a bit of time in program development. The random disk file handler takes care of the following functions:

1. The initialization of several arrays to store status information and parameters for up to 15 files.

2. The specification of logical record lengths for up to 15 files. Automatic blocking of multiple logical records into 256-byte physical records when the logical record length is less than 129 bytes.

3. Standard procedures for getting and fielding logical records by logical record number, with disk error handling.

4. Standard procedures for writing records to disk, with error handling.

5. The optional handling of "soft gets" and "soft puts" to eliminate unnecessary and time-consuming disk accesses. (In other words, the handler keeps track of the last physical record that was accessed for each file. If the next logical record requested is in the same physical record, the handler skips the GET command. The handler, if desired, will only issue a PUT command when a different physical record is accessed.)

6. Standard procedures for opening and closing files with error handling.

7. Disk error handling procedures that let the operator correct the error condition and retry as often as required, without the destruction of program data or video display layouts. Diagnostic error messages are printed on the bottom line of the screen, but upon successful resumption, the screen contents and cursor position are restored.

Over the past few years I have used the subroutines contained in the random file handler in dozens of programs on the TRS-80 Model I and III. The standardization has been a real time saver and has freed me to concentrate on the applications, rather than the details of disk programming.

Since the random file handler works with fixed length records rather than variable length records, it provides complete compatibility among the various operating systems available for the Model I and III. A few changes are required for the Model II. They are listed at the end of this chapter.

RFH Organization

The random disk file handler subroutines occupy program lines in the 58000 series. No other supporting subroutines are required. The only requirement is that you define variables starting with "L" and "P" as integers. You can do this with a DEFINT command, or you can change the handler by inserting "%" after each variable beginning with "L" or "P".

They are all explicitly defined in the listings shown. If you wish to save memory, you can remove the "$" symbol following each variable starting with "F", and DEFSTR "F" elsewhere in your program.

RFH Subroutines

Initialization:

```
58ØØØ A%=3:DIMPR(A%),PP(A%),LR(A%),LL(A%),FD$(A%),FR$(A%)
58ØØ1 LL(1)=36:LL(2)=6:LL(3)=42
58ØØ2 RETURN
```

Fielding:

```
58Ø1Ø FIELDPF,LS*LL(PF)ASFD$(PF),LL(PF)ASFR$(PF)
58Ø11 RETURN
58Ø2Ø FIELDPF,LS*LL(PF)ASFD$(PF),LL(PF)ASFR$(PF)
58Ø21 RETURN
58Ø3Ø FIELDPF,LS*LL(PF)ASFD$(PF),LL(PF)ASFR$(PF)
58Ø31 RETURN
```

Field and Get Logical Record:

```
582ØØ LB=INT(256/LL(PF)):LS=LR(PF)-LB*INT((LR(PF)-1)/LB)-1:PR(PF
)=INT((LR(PF)-1)/LB)+1
582Ø5 ONPFGOSUB58Ø1Ø,58Ø2Ø,58Ø3Ø
```

Physical Record Soft-Get:

```
5821Ø IFPR(PF)=PP(PF)THENRETURN
58215 IFPSAND(2↑(PF-1))THENGOSUB583Ø5
```

Physical Record Hard-Get:

```
5822Ø PP(PF)=PR(PF):ONERRORGOTO589ØØ:GETPF,PR(PF):ONERRORGOTOØ:R
ETURN
```

File Open:

```
5825Ø GOSUB5829Ø:ONERRORGOTO5891Ø:OPEN"R",PF,FS$:ONERRORGOTOØ:PP
(PF)=Ø:RETURN
```

File Close:

```
5829Ø ONERRORGOTO5893Ø:CLOSEPF:ONERRORGOTOØ:RETURN
```

Hard-Put:

```
583ØØ ONERRORGOTO5892Ø:PUTPF,PR(PF):ONERRORGOTOØ:RETURN
```

Soft-Put:

```
583Ø5 IFPP(PF)=ØTHENRETURNELSEONERRORGOTO5892Ø:PUTPF,PP(PF):ONER
RORGOTOØ:RETURN
```

Error Handling:

```
589ØØ A$="DISK READ ERROR":GOTO5899Ø
5891Ø A$="CAN'T OPEN DISK FILE":GOTO5899Ø
5892Ø A$="DISK WRITE ERROR":GOTO5899Ø
5893Ø A$="CAN'T CLOSE DISK FILE":GOTO5899Ø
5899Ø A1$="":A%=VARPTR(A1$):POKEA%,64:POKEA%+1,192:POKEA%+2,63:A
2$=A1$:A%=PEEK(16416):A1%=PEEK(16417)
58991 PRINT@96Ø,CHR$(143);A$;TAB(22)"(E=";MID$(STR$(ERR/2),2);"
F=";MID$(STR$(PF),2);" R=";MID$(STR$(PR(PF)),2);")";TAB(41);"PRE
SS ENTER TO RETRY!";CHR$(143);
58992 A$=INKEY$:IFA$=""THEN58992
58993 PRINT@96Ø,CHR$(31);
58994 LSETA1$=A2$:POKE16416,A%:POKE16417,A1%
58995 'TEST ON AN ABORT COMMAND HERE
58996 RESUME
```

Simple Variables

A$,A1$,A2$,A%,A1% Temporary working variables during disk error handling. (Not used if no error is encountered.)

FS$ Provided by mainline program to specify file name and drive number for an OPEN.

LB% Blocking factor (number of subrecords per physical record for the current file). Computed by the handler to determine physical number to GET.

LS% Number of subrecords within the current physical record that precede the desired logical record. Computed by the handler before fielding.

PF% Provided by the mainline program to specify the file number (1-15) to be used in subsequent calls to the handler.

PS% Loaded by the mainline program to specify the files for which the soft-put option is active. If PS%=0 the soft-put option is deactivated for all files. If PS%=255 the soft-put option is active for all files. Otherwise, bit 0 is set to activate the option for file 1, bit 1 is set to activate the option for file 2, etc.

Arrays

LL%( ) Specifies the logical record length for each file. The subscript refers to the file number.

LR%( ) Specifies the current (or requested) logical record number. The subscript refers to the file number provided by your mainline program before a call to subroutine 58200.

PR%( ) Specifies the current physical record number. The subscript refers to the file number, automatically computed when subroutine 58220 is called, but your mainline program may load it when calling 58210, 58220 or 58300.

PP%( ) Specifies the previous physical record number. The subscript refers to the file number. If the newly requested physical number, PR%(PF%), is equal to PP%(PF%) then no GET is performed by subroutine 58210.

FD$( ) The dummy field containing all subrecords that precede the current logical record. The subscript refers to the file number. Fielded automatically by the handler.

FR$( ) The default field for each file shown in the handler listing. The subscript refers to file number. (In most applications, you will want to replace the FR$(PF%) field with your own fielding.) The length of the FR$(PF%) field is equal to the logical record length for the file, PF%.

Line Comments

58000 (Perform Initialization of Handler Arrays.)

Load A% with the number of files to dimension for.
Dimension PR% array, PP% array, LR% array, LL% array, FD$ array. (Optionally, dimension FR$ array if default fielding is to be used.)

58001 Load the logical record lengths for each file into the LL% array.

58002 Return to mainline program.

58010

Handle logical record fielding for file 1.
The first field is a dummy field, FD$(1), whose length equals the number of preceding subrecords, LS%, multiplied by the logical record length for file 1, LL%(1).
Handle the fielding for subsequent fields in the logical record. Unless modified for the specific program, default is to field the whole logical record as FR$(1).

58011 Return. (Normally to line 58205.)

58020 Handle logical record fielding for file 2.

58021 Return.

58030 Handle logical record fielding for file 3.

58031 Return.

58040 – 58150 (Handle fielding for files 4 through 15, if necessary, according to the same pattern used for 58010 through 58031.)

58200

(Field and get logical record specified as LR%(PF%)).
Compute blocking factor for current file as LB%.
Compute number of preceding subrecords as LS%.
Compute physical record number as PR%(PF%).

58205

Based on the file number, PF%, call a subroutine to field the logical record. (Subroutine 58010 fields for file 1, 58020 fields for file 2, etc.)

58210 (Physical record soft-get.)

If the desired physical record for the desired file, PR%(PF%), is equal to the previous physical record gotten for the same file, PP%(PF%) then return to mainline program — no GET is required.

58215

Before doing the GET, check to see if the soft-put option is active for the current file. If the bit within PS%, corresponding to the current file number, PF%, is set then the soft-put option is active. Call subroutine 58305 to PUT the previous physical record.

58220 (Physical record hard-get)

Load the previous physical record indicator, PP%(PF%), with the number of the current physical record, PR%(PF%).
Load the "ON ERROR GOTO" line number, 58900.
Get the desired physical record, PR%(PF%), from the desired file, PF%.
Deactivate the "ON ERROR GOTO" line number.
Return to mainline program.

58250 (OPEN the requested random file.)

:First, call subroutine 58290 to insure that the file is closed.
:Load the "ON ERROR GOTO" line number, 58910.
:Open in random mode, the file number specified by PF%, using the file name specified by FS$.
:Deactivate the "ON ERROR GOTO" line number.
:Set previous physical record number for the file to 0, to insure that the first GET is a hard-get.
:Return to the mainline program.

58290 (Close the requested file.)

:Load the "ON ERROR GOTO" line number, 58930.
:Close the requested file, PF%.
:Deactivate the "ON ERROR GOTO" line number.
:Return to the mainline program.

58300 (Put the requested physical record number.)

:Load the "ON ERROR GOTO" line number, 58290.
:Write the desired physical record, PR%(PF%) to the desired file, PF%.
:Deactivate the "ON ERROR GOTO" line number.
:Return to the mainline program.

58305 (Put previous physical record — soft put.)

:If previous physical record is zero, then return.
:Otherwise, load the "ON ERROR GOTO" line number, 58920.
:Write the previous physical record, PP%(PF%) to the desired file, PF%.
:Deactivate the "ON ERROR GOTO" line number.
:Return (normally to 58215).

Error Handling — Lines 58900 - 58996

58900 (Handle errors for disk file GET commands.)

Load message, "DISK READ ERROR".
:Go to 58990 to display it.

58910 (Handle errors for disk file OPEN commands.)

Load message, "CAN'T OPEN DISK FILE".
:Go to 58990 to display it.

58920 (Handle errors for disk file PUT commands.)

:Load message, "DISK WRITE ERROR".
:Go to 58990 to display it.

58930 (Handle errors for disk file CLOSE commands.)

Load message, "CAN'T CLOSE DISK FILE".
:Go to 58990 to display it.

58990 (Handle display of error messages and restore screen on RESUME.)

:Make sure that A1$ is initialized.
:Put VARPTR address for A1$ in A%.
:Poke the address so A1$ has a length of 64.
:Point A1$ to bottom line of video display. (Poke LSB of 960)
:Continue. (Poke MSB of 960)
:Save bottom line of display in temporary string, A2$.
:Save LSB of current cursor address in A%,
:Save MSB of current cursor address in A1%.

58991 Print error message on bottom line of display.

(Bracket the message in graphics characters.)

58992

Wait for the operator to press a key.
Store the result of the key depression in A$.

58993

Now that the operator has pressed a key, clear the bottom line of the display.

58994

Restore previous contents to bottom display line.
:Restore cursor position. (Re-poke LSB.)
:Continue. (Re-poke MSB of cursor position.)

58995

Logic that RESUMEs to a "safe" line number upon an unrecoverable error, (if operator presses a key other than ENTER), should be inserted here. For example:

```
"IF A$<>CHR$(13) THEN RESUME 1ØØ"
```

58996 Retry the line containing the error with a RESUME.

Implementing the RFH

When developing a program that utilizes the random file handler, you should first type in or merge the standard handler subroutines that occupy lines 58000 through 58996. You will then need to make a few modifications to certain subroutines to customize them for your file layouts. Finally, you may program the remainder of your application, using GOSUB commands where you wish to execute functions performed by the handler. Remember that variables beginning with P and L must be defined as integers.

Number of Files Modification

The random files handler, as it is shown, is configured to allow 3 random files in a program. If your application has fewer than 3 files or more than 3 files, you'll need to make the following modifications:

1. Change the "3" in line 58000 to the number of files you require.

2. If you need more than 3 files, insert lines between 58040 and 58151, following the same pattern shown between 58010 and 58031. If, for example, you require 4 files, you will need a line at 58040, (identical to the line at 58030), and a line at 58041, (identical to the line at 58031). If you need fewer than 3 files, you can delete 58030 and 58031. If you only need 1 file, you can delete 58020 and 58021.

3. If you need more than 3 files, extend the ON GOSUB command in line 58205, by adding ",58040,58050", and so forth, according to the number of files you need. If you need fewer than 3 files, delete the "58030", and if you wish, the "58020" from line 58205.

Customizing the Fielding Commands

Lines 58010 through 58150 perform the logical record fielding for each of up to 15 files.

Line 58010 fields the desired logical record for file 1, line 58020 fields the desired logical record for file 2, and so forth. To put in your own fielding, delete the last 15 bytes which read:

```
LL(PF)ASFR$(PF)
```

Then, replace your fielding for the logical record. Suppose, for example, you want each logical record for file 3 to be 28 bytes long, containing 2 fields, D$, and P$. You want D$ to store a 24-byte description, and P$ to store a 4-byte price. To do this you simply add the following to line 58030:

```
24 AS D$, 4 AS P$
```

Line 58030 will then read:

```
58Ø3Ø FIELDPF,LS*LL(PF)ASFD$(PF),24 AS D$, 4 AS P$
```

You should do the same for every other file you'll be using.

If you wish, you may, instead of customizing the fielding, leave the fielding lines as they are. This causes the handler to load the entire contents of a given logical record into the FR$ array element that corresponds to the file number. The logical record will just have one field.

Specifying Your Logical Record Lengths

The only other customizing step required is to specify the logical record length for each file. This is done in line 58001. In the listing shown, the logical record length for file 1 is 36 bytes, for file 2 it is 6 bytes, and for file 3 it is 42 bytes. Modify line 58001 so that, for each file you'll be using, the corresponding element of the LL% array is loaded with the logical record length.

If you are using NEWDOS 2.1, you may wish to change the "256" in line 58200 to a "255". You can also use the POKE VARPTR method we discussed earlier to allow fielding of 256 bytes. This could be done in lines 58011, 58021 and 58031 for a 3-file application.

Calling the RFH Subroutines

Once you've modified the random disk file handler for your application program, it is a simple matter to use it. You simply load a few required variables, and then GOSUB to the proper line number. You should use the handler subroutines even for very simple functions, such as opens and closes. This insures that you'll have proper error handling in the event the diskette is not present, or in the event there is a disk input or output error.

Initializing the Handler Arrays

The first action that your mainline program should take when you want to use the random disk file handler is to GOSUB 58000. Subroutine 58000 dimensions several arrays, according to the number of files you'll be using. It also loads the logical record lengths for each file into the LL% array.

You should execute the GOSUB 58000 command only once within a program, otherwise you'll get a "redimensioned array" error.

Opening Files

Before you can use a disk file you must OPEN it. To open a disk file, your program should load the desired file name, including disk drive number, into the variable FS$. PF% should be loaded with the desired file number. Then, you simply issue a GOSUB 58250 command. To open a random file named "ACCOUNTS" on drive one, as file 2, your command is:

```
FS$ = "ACCOUNTS:1" : PF=2 : GOSUB5825Ø
```

I often open more than one file with the same program line. Here is a program line that opens 3 files, "INDEX:1", "MASTER:1" and "TRANS:2":

```
FORPF=1TO3:
FS$=MID$("INDEX:1 MASTER:1 TRANS:2",(PF-1)*8+1,8):
GOSUB5825Ø:
NEXT
```

If for some reason the computer cannot open the disk file, the following error message will be displayed on the bottom line of your video display:

```
CAN'T OPEN DISK FILE  (E=## F=## R=Ø)     PRESS ENTER TO RETRY!
```

The number following the "E" will give the error code from your Disk BASIC manual. The number following the "F" will be the file number. The most likely error codes are:

57 Disk I/O Error — (The diskette is not properly centered on the disk drive spindle; there is a bad spot on the diskette; the directory is invalid, or the diskette has not been formatted.)

64 Bad File Name — (You've made a programming error. Check the contents of FS$.)

67 Too Many Files — (The disk drive is not turned on or connected, the door is open, the diskette is not inserted properly, or you have more than 48 files on the diskette.)

69 File Access Denied — (An error with the password.)

52 Bad File Number — (The How Many Files question was answered improperly, or you've made a programming error — check the contents of PF%.)

Closing Files

To close a file, you can load PF% with the file number and GOSUB 58290. To close files 1 through 5 you command can be:

```
FOR PF = 1 TO 5 : GOSUB5829Ø :NEXT
```

To simply close file 3, your command could be:

```
PF=3:GOSUB5829Ø
```

If for some reason the computer cannot close a disk file, you'll get an error message on the bottom line of your screen:

```
CAN'T CLOSE DISK FILE (E=## F=## R=Ø)     PRESS ENTER TO RETRY!
```

The number following "E" will be the error code. The file number will follow the "F". The most common cause of an error when closing files is a premature removal of the diskette by the operator.

GETting and Fielding a Logical Record

Here's where the real programming convenience of the handler comes into play. To GET and field a logical record, you simply load PF% with the file number and LR%(PF%) with the desired logical record number. Then you GOSUB 58200. For example, to get logical record 10 from file 3 your command is:

```
PF=3 : LR(PF)=1Ø : GOSUB582ØØ
```

To get a series of logical records (for searching or printing), you can use a FOR-NEXT loop. The following logic reads through the first 50 logical records of file 3.

```
PF=3
FOR X = 1 TO 5Ø
LR(3)=X : GOSUB582ØØ
'PERFORM DESIRED FUNCTION HERE
NEXT
```

Subroutine 58200 does a "soft-get". It computes the physical record number in the desired file based on the requested logical record number. The subroutine only does a GET if the computed physical record number is different from the last physical record accessed. Each call to subroutine 58200 refields the disk buffer. In most cases, the soft-get capability improves the speed of your programs.

If you wish to force a "hard-get" (making the computer re-read the diskette physical record into the buffer), you can do so by loading PP%(PF%) with 0 before your GOSUB58200. The PP% array contains the previous physical record number for each file. A "hard-get" is performed only when PP%(PF%) is not equal to PR%(PF%). To GET and field logical record number 20 of file 1, forcing a physical disk access, your command could be:

```
PF=1:PP(PF)=Ø:LR(PF)=2Ø:GOSUB582ØØ
```

If for some reason an error is encountered when the computer tries to read the diskette, you'll get an error message on the bottom line of your screen:

```
DISK READ ERROR     (E=## F=## R=#####)    PRESS ENTER TO RETRY!
```

The number following the "E" indicates the error code. The number following "F" indicates the file number. The number following "R" gives the physical record number that the computer was trying to access. The most common error code is 57. Usually, pressing <ENTER> to let the computer try again will result in a successful read, especially if the operator verifies that the diskette is properly centered on the drive spindle. Here are the some of the error codes you could get:

57 DISK I/O ERROR — (The diskette is not properly centered on the disk drive spindle, there is a bad spot on the diskette, the directory is invalid, or the diskette has not been formatted.)

63 Bad Record Number — (Most likely, your program tried to access a negative or zero logical record number.)

50 Field Overflow — (You could get this error during program development if your operating system only allows fielding of 255 bytes and you try to field 256 bytes.)

GETting a Physical Record

In most application programming, you'll be working with logical records, but (if you wish) you can bypass the fielding and physical record computation functions. To do a physical record "soft-get", you simply load PF% with the file number, PR%(PF%) with the desired physical record number, and GOSUB 58210. For example, to get physical record 25 from file 4, your command is:

```
PF=4:PR(4)=25:GOSUB58210
```

The call to 58210 does a "soft-get". That is, if the requested physical record number is already in the buffer, the disk is not accessed. The subroutine determines this by comparing PR%(PF%) to PP%(PF%). If they are the same, no GET is necessary.

To do a "hard-get" with full error handling, you can do a GOSUB 58220 after loading PF% with the desired file number, and PR%(PF%) with the desired physical record number.

With any call to 58200, 58210 or 58220, the previous record number indicator, PP%(PF%), is updated so that the handler knows which record number is currently in the buffer.

The errors that you are likely to get when accessing physical records are the same as those explained for logical record GETs.

PUTting a Physical Record

To PUT a physical record, you simply load PF% with the desired file number, and PR%(PF%) with the desired physical record number. Then you GOSUB58300. For example, to record the contents of disk buffer 1 to physical record 30, your command is:

```
PF=1:PR(PF)=30:GOSUB58300
```

In most cases, PR%(PF%) will already contain the physical record number you want to PUT because you will have executed a GET prior to the PUT. The file number, PF%, may also already have the proper contents, but in multi-file applications it's always good practice to load PF% before a call to the handler.

Let's suppose you want to update the contents of logical record 132 in file 2. The sequence of commands is:

```
PF=2:LR(PF)=132:GOSUB58200   'GET THE LOGICAL RECORD
LSET ... = ...               'LSET NEW VALUES INTO THE DESIRED FIELDS
LSET ... = ...               'ETC.
PF=2:GOSUB58300              'PUT THE PHYSICAL RECORD
```

If the computer encounters a disk error when trying to execute the PUT command, the following error message will be displayed on the bottom line of your screen:

```
DISK WRITE ERROR    (E=## F=## R=#####)    PRESS ENTER TO RETRY!
```

The number following "E" will indicate the error code. "F" will indicate the file number, and "R" will indicate the physical record number the computer was trying to PUT. Here are the most likely errors you may encounter:

57 Disk I/O Error — (The diskette is not properly centered on the disk drive spindle; there is a bad spot on the diskette; the directory is invalid, or the diskette has not been formatted.)

61 Disk Full — (There is no space left on the diskette.)

63 Diskette Write-Protected — (The notch on the diskette is covered, or there is some other hardware or diskette fault. The operator can correct the problem and press <ENTER> to resume.)

69 Bad Record Number — (Your program is trying to write to a zero or negative record number, or it is trying to write to a physical record number beyond the range of your operating system.)

Using the "Soft-put" Capability

The random disk file handler lets you use "soft-puts" for random files. With the soft-put option, the computer doesn't issue a PUT command until it needs to. The effect is that your program will run faster because the disk drive is turned on less frequently. The PUT command is delayed until you access a new physical record. Then, just before the GET of the new physical record, the previous physical record is PUT.

The soft-put option can be dangerous in operator-oriented applications. If the operator removes the diskette or there is a power failure before the updated record is PUT, the new data won't get recorded. I prefer to use "hard-puts" (with subroutine 58300) in operator-oriented applications. The soft-put option is better suited to applications where the computer is updating disk files continuously in a batch mode.

The handler lets you set a flag for each file for which you want the soft-put option activated. You do this by setting the bit in integer PS% that corresponds to the file number.

To activate the soft-put option for file 3, your command is:

```
PS=PS OR 2↑(3-1)
```

To activate the option for file 2 your command is:

```
PS=PS OR 2↑(2-1)
```

To activate it for all files, you can use the command:

```
PS=255
```

To deactivate it for all files, you can use the command:

```
PS=Ø
```

To deactivate the soft-put option for file 5, your command is:

```
PS = PS ANDNOT 2↑(5-1)
```

In applications where you are just reading from a file for display or printout purposes, you should deactivate the soft-put option for the file. This prevents the computer from PUTting the previous physical record each time a new logical record is accessed.

For any file for which you are using the soft-put capability, it is very important for your program to do the final PUT before the file is closed (or before the application is otherwise terminated). If for example, you are operating file 3 with the soft-put option, you should execute the following command before closing file 3:

```
PF=3 : GOSUB583ØØ
```

(You don't need to load PR(3) with the physical record number — it already has it.)

Other than the special considerations we just discussed, no other commands are required for the soft-put option.

Here's an example of how the soft-put option is used. Let's say file 3 has a logical record length of 16 bytes. Each logical record has two 8-byte fields, FH(1) and FH(2). You wish to add the double-precision number stored in FH(1) to the double-precision number stored in FH(2) for each of the first 500 logical records, storing the result in FH(2). Your program commands are:

```
PS=PS OR 2↑(3-1)                'ACTIVATE SOFT PUT OPTION FOR FILE 3
PF=3                            'SPECIFY FILE NUMBER 3
FOR X=1TO500                    '
LR(3)=X :GOSUB58200             'GET AND FIELD LOGICAL RECORD
LSET FH(2) =
MKD$(CVD(FH(1))+CVD(FH(2)))     'UPDATE THE DATA IN CURRENT RECORD
NEXT                            'REPEAT FOR NEXT LOGICAL RECORD
GOSUB58300                      'PUT THE FINAL RECORD
```

This method is much faster than issuing a GOSUB 58300 after each record is updated. A PUT is only executed after every 16th logical record is processed.

Your error messages under the soft-put option will fall into the same categories as those for the other get and put subroutine calls.

Random File Handler Modifications

In addition to the modifications you normally must make to customize the random disk file handler for your file layouts and the number of files you wish to use, there are many other modifications you can make. Some of them are explained below.

Deletion of Unneeded Capabilities

In some applications, you may not need the logical record fielding capabilities of the handler. For these applications you may delete lines 58001, 58010 – 58051, 58200 and 58205. In addition, you may delete the references to LL(A%) and LR(A%) in line 58000.

In programs that just read from random files (for the purpose of display or printouts), you may delete all instructions that handle disk file PUTs. You may delete lines 58215, 58300, 58305 and 58920.

If your application doesn't require the "soft-put" capability, you may delete lines 58215 and 58305. I usually make this modification. It makes things a little simpler.

The disk error handling routines save the cursor position and the contents of the bottom line of the video display. If you've designed your application with PRINT@ commands and your application reserves the bottom line of the display for prompting and error messages, you can eliminate some logic from the handler. You may delete line 58994 and change line 58990 to "58990 REM". With this change, the variables A1$, A2$, A%, and A1$, are not used by the handler.

Other Modifications

You may be using some of your files as sequential files, and, therefore, they will not be processed by the handler. If this is the case, you can delete the fielding lines (58010–58011, 58020–58021, etc.) that correspond to the file number.

Some files may have alternative logical record fieldings. Let's suppose, for example, that in one situation file 1 is to be fielded differently than in another situation. To handle this, you can load a "flag" variable just before your call to 58200. Then you can test on the flag in line 58010, directing the program logic to the proper fielding command.

Some files may have logical records with more fields than will fit in a single FIELD command. Let's suppose, for example, that file 2 has 20 fields, but you don't have enough space in line 58020. To handle this, you can field the first 10 fields in line 58020, and the remaining 10 fields in line 58021. If the combined length of the first 10 fields is 37 bytes, your field command in line 58021 starts with:

```
58Ø21 FIELDPF,LS*LL(PF)+37 AS FD$(PF),
```

and the fielding for the last 10 fields follows the comma. The return command that is normally in 58021 can be moved up to 58022.

Line 58992 has its own single-key routine. If you have subroutine 40500 in your program (as presented in *BASIC Faster and Better and Other Mysteries*), you can replace the logic in line 58992 with a "GOSUB40500". If you don't want A$ to be destroyed by a possible disk error, you can replace "A$" in line 58992 with a different variable name.

You should insert logic in line 58995 to allow escape from an unrecoverable disk error. Since the message displayed is "PRESS ENTER TO RETRY!", you can, in line 58995, RESUME to a special line number if any key other than <ENTER> is pressed.

The error handling messages are designed to be short, so they will fit on one line of the display. You can, of course, expand on the error handling logic. If you want to conserve memory, you can have a single message, such as "DISK ERROR". To make this modification you can change all "ON ERROR GOTO" statements in the handler so they will go to line 58900. You can then delete lines 58910 through 58930, changing line 58900 to read:

```
589ØØ A$="DISK ERROR"
```

Modifications for the Model II

BASIC on the TRS-80 Model II returns an error if you attempt to do a GET beyond the end of a file. You'll also find that the error code numbers are different, and the logic for restoring the last line of the video display is not applicable. The following replacements and additions change the handler so it can be used on the Model II:

```
5822Ø PP(PF)=PR(PF):IFPR(PF)>LOF(PF)THENGOSUB583ØØ
58221 ONERRORGOTO589ØØ:GETPF,PR(PF):ONERRORGOTOØ:RETURN
5899Ø '
58991 PRINT@184Ø,CHR$(143);A$;TAB(22)"(E=";MID$(STR$(ERR),2);" F
=";MID$(STR$(PF),2);" R=";MID$(STR$(PR(PF)),2);")";TAB(41);"PRES
S ENTER TO RETRY!";CHR$(143);
58993 PRINT@184Ø,CHR$(24);
58994 '
```

The Labeled Files Handler

The Labeled Files Handler provides a standard method for handling statistics about a random file. It makes it easy to implement features into your disk programs that make data storage faster, more efficient, and more secure.

Random disk files present a special problem, especially when you are using a logical record length that is different from the physical record length. Your program needs a way to know how many logical records are in the file. Your BASIC program can get the number of physical records in the file with the built-in LOF function. For example, LOF(2) gives you the length, in physical records, of file 2. But, suppose you've blocked 6 logical records per physical record. You have no automatic way of knowing which logical record is the last one in the event you want to add new records at the end of the file.

One way to handle this problem was shown in the RANDOM8/DEM program. You got the last record in the file and looked for the first subrecord that contained an ASCII 0 as the first byte. Many times, however, it's desirable to build a file of the length you anticipate you'll need before you start putting data in it. This method of "pre-allocating files" helps to avoid unexpected disk full errors, and it prevents lost data if you fail to close the file. The technique used in RANDOM8/DEM won't work with pre-allocated files.

Deletions present another problem. Suppose you have a file of 500 customer names and addresses, and you want to delete the 245th customer. It's an easy matter to just put a "deleted" code in that customer's logical record, but suppose you want to reuse that space when you add a new customer. Disk BASIC provides no automatic way to do it.

Suppose you have an application where there are several files, perhaps on multiple diskettes. Disk BASIC gives you no automatic way of knowing the date each file was last processed, so there can be a risk of using files that are not current. In multi-file applications, you may want to insure that each file being used is synchronized with the others.

For example, in an accounts receivable application, you want to verify that the customer master file is not being used with a transaction file having a different date.

I've found that the best solution to these problems is to devise a standard format to programmatically "label" random disk files. Having a label that specifies the current statistics about a file makes it possible to use standard subroutines for adding logical records, deleting logical records, verifying the date the file was last processed, and verifying the validity of the file.

The standard label I use for storing file statistics occupies the first 10 bytes of a random disk file. Since this is the position that would normally be occupied by the first logical record, I simply consider logical record 2 to be my first record. (In the event my logical record length is less than 10, I consider the first logical record beyond the 10th byte to be the first.) The 10-byte label contains 5 integers in MKI$ format. They are:

Bytes 1 and 2 Next logical record number at the end of the file. This is the location where the next record will be added if there have been no deletions.

Bytes 3 and 4 Last logical record number deleted. If there are currently no deletions within the file, this number will be zero.

Bytes 5 and 6 Number of active logical records currently in the file. For an empty file, this number is zero. When a logical record is added, this number is incremented. For a deletion, this number is decremented.

Bytes 7 and 8 File validity code. If this number is 0, the file is valid. If this number is 1, the file is invalid. An invalid condition indicates that the file was not closed properly after additions or deletions to the file were made.

Bytes 9 and 10 File synchronization code. Here, I normally store a number that specifies the date the file was last processed.

After opening a labeled file, I call a subroutine that loads the statistics into a 2-dimensional array, L0%(X,Y). The "X" dimension, specifies the file number. The "Y" dimension specifies the file statistic, 1 through 5. Where PF% is the file number, the L0% array has the following contents:

LO%(PF%,1)	Next logical record, end of file.
LO%(PF%,2)	Logical record number of most recent deletion.
LO%(PF%,3)	Number of active logical records in the file.
LO%(PF%,4)	File validity code.
LO%(PF%,5)	File synchronization code or date.

As you'll see later, the zero array element for each file, L0%(PF%,0), is used for temporary storage when we're adding a new record. F0$(1) through F0$(5) are used for fielding the five 2-byte elements on disk.

Organization of the LFH

The subroutines for labeled file handling occupy lines 58400 Through 58441, and they work in conjunction with the standard random disk file handler subroutines. Only one change is required. To line 58000 of the random disk file handler, you should insure that the L0% and F0$ arrays are dimensioned by adding ",L0%(A%,5),F0$(5)" to the list of arrays.

LFH Subroutines

Write File Statistics to Disk:

```
584ØØ PR(PF)=1:LØ(PF,4)=KV%:LØ(PF,5)=KS%:GOSUB5841Ø:GOSUB5821Ø:F
ORA%=1TO5:LSETFØ$(A%)=MKI$(LØ(PF,A%)):NEXT:GOSUB583ØØ:RETURN
```

Read File Statistics from Disk:

```
584Ø5 PR(PF)=1:GOSUB5841Ø:GOSUB5821Ø:FORA%=1TO5:LØ(PF,A%)=CVI(FØ
$(A%)):NEXT:RETURN
```

Initialize Statistics for a File:

```
584Ø8 LØ(PF,1)=9/LL(PF)+2:LØ(PF,5)=KS%:FORA%=2TO4:LØ(PF,A%)=Ø:NE
XT:GOTO584ØØ
```

Fielding for File Statistics:

```
5841Ø FIELDPF,2ASFØ$(1),2ASFØ$(2),2ASFØ$(3),2ASFØ$(4),2ASFØ$(5):
RETURN
```

Get Logical Record to Add:

```
5842Ø IFLØ(PF,2)=ØTHEN58421ELSELR(PF)=LØ(PF,2):GOSUB582ØØ:FIELDP
F,LS*LL(PF)+2ASFD$(PF):LØ(PF,Ø)=CVI(RIGHT$(FD$(PF),2)):RETURN
58421 LR(PF)=LØ(PF,1):LØ(PF,Ø)=-1:GOSUB582ØØ:RETURN
```

Add Logical Record:

```
5843Ø GOSUB583ØØ:LØ(PF,3)=LØ(PF,3)+1:IFLØ(PF,Ø)>-1THENLØ(PF,2)=L
Ø(PF,Ø)ELSELØ(PF,1)=LØ(PF,1)+1
58431 GOSUB584ØØ:RETURN
```

Delete Logical Record:

```
5844Ø FIELDPF,LS*LL(PF)ASFD$(PF),LL(PF)+(LL(PF)=256)ASFD$(PF):LS
ETFD$(PF)=MKI$(LØ(PF,2))+STRING$(LL(PF)-2+(LL(PF)=256),255)
58441 GOSUB583ØØ:LØ(PF,2)=LR(PF):LØ(PF,3)=LØ(PF,3)-1:GOSUB584ØØ:
RETURN
```

Line Comments

58400 (Write file statistics to disk.)

Desired physical record, PR%(PF%), is 1.
:Load validity code, KV% into statistic element 4.
:Load synchronization code, KS%, into statistic element 5.
:Call subroutine 58410 to field the buffer for storing statistics.
:Call subroutine 58210 to get physical record 1 of the file.
:For each statistic, 1 through 5,
:LSET it into the corresponding field.
:Repeat for the next statistic.
:Call subroutine 58300 to PUT physical record 1.
:Return.

58405 (Read file statistics from disk.)

Desired physical record, PR%(PF%), is 1.
:Call subroutine 58410 to field the buffer for file statistics.
:Call subroutine 58210 to get physical record 1 of the file.

:For each statistic, 1 through 5,
:Load it into the corresponding element of the L0% array.
:Repeat for the next statistic.
:Return.

58408 (Initialize statistics for a file.)

Load "next record, EOF" statistic, L0%(PF%,1) as the first logical record number of the file that is beyond the first 10 bytes.
:Load the file synchronization or date statistic, L0%(PF%,5), with the current contents of KS%.
:For every other statistic,
:Initialize to zero.
:Repeat for next statistic.
:Go to 58400 to write the statistics to disk physical record 1.

58410 (Fielding for file statistics.)

Field the first 10 bytes of the specified file, PF%, to allow reading and writing of file statistics 1 through 5.
:Return

58420 (Get logical record to add.)

If there are no internal deletions, then go to line 58421 to add at the end of the file, Otherwise, add at the location of the last deleted logical record. Load desired logical record with the contents of statistic 2 for the current file.
:Get the logical record by calling subroutine 58200.
:Now that you have gotten a previously deleted record, you know that the first 2 bytes of it contain the pointer to the deletion prior to this one in the chain. Field the buffer so that the last 2 bytes of the dummy field, FD$(PF), contain the pointer.
:Store the pointer temporarily in statistics element 0.
:Return to the mainline program.

58421

There are no internal deletions, so the logical record to be gotten is the next record at the end of file.
:Load statistics element 0 for this file with "-1" to indicate that you will be adding at the end of the file.
:Get the logical record.
:Return to the mainline program.

58430 (Add the current logical record.)

Call subroutine 58300 to PUT the current physical record.
:Add 1 to the count of active records for the current file.
:If you are replacing a previously deleted record, then the new "last deletion" pointer for the file is the pointer that was stored in the first 2 bytes of this record.
:Otherwise, add 1 to the "next record at end of file" statistic.

58431

Call subroutine 58400 to record the new statistics for this file.
:Return to the mainline program.

58440 (Delete the current logical record.)

Field the buffer so the field, FD$(PF%), contains
the current logical record. Note the "creative fielding." The first fielding of FD$(PF%) is replaced by the second, avoiding the need to use more than one variable.
:LSET the dummy field so that the preceding subrecords are unchanged, the first 2 bytes of the current logical record contain the pointer to the previous deletion, and the remaining bytes of the current logical record contain CHR$(255) codes.

58441

Call subroutine 58300 to PUT the current physical record.
:Load the "last deletion" statistic of this file with the current logical record number.
:Subtract 1 from the count of active records for this file.
:Call subroutine 58400 to record the new statistics for this file.
:Return to the main program.

Using the LFH

The labeled files handler relies heavily upon the random disk file handler subroutines. In effect, they add a higher level of "automation" to the RFH, but you still, of course, can use any of the random disk file handler subroutines directly.

Furthermore, you should be aware that not all random files within an application need to be labeled. For example, I have, in some programs, treated an inventory master file as two separate files. One file, "ITEMDESC", contains the product numbers, prices, and descriptions. A separate file, "ITEMQUAN", perhaps on a different disk drive, contains the on-hand and on-order quantities for each product. The "ITEMQUAN" file is, in effect "parallel" to the "ITEMDESC" file. Though the logical record lengths for the 2 files are different, the "nth" item in the "ITEMQUAN" file contains the quantities that go with the "nth" item in the "ITEMDESC" file. Because of this parallel layout, the "ITEMDESC" file is a labeled file and it contains the file statistics, but the "ITEMQUAN" file is not.

The KS% and KV% Variables

The KS% integer variable specifies the date, or synchronization code, that is to be recorded into a file's label. Normally, upon the daily startup of an application, you will want to load KS% with a integer date that will be recorded into the file labels processed during the run.

If your application is designed so the current date has been loaded into TIME$, you can load an integer month and day into KS% with the following expression:

```
KS% = VAL(TIME$)*1ØØ + VAL(MID$(TIME$,4))
```

With this method, the date "12/05/82", is stored in KS% as 1205. If you've inserted the 2-byte date function calls explained in *Basic Faster and Better*, you can store month, day and year in integer KS%:

```
KS% = CVI(FNC2$(FNCD$(TIME$)))
```

There are many other methods you can use. I've seen systems where a synchronization number is simply incremented by 1, each time a file is opened.

Integer KV% controls the validity codes that are recorded. The KV% variable should be loaded with 1 when you start a run. Just before closing your files, it should be loaded with 0.

Here's how the validation variable works. After adding or deleting a record during a run, the file statistics are re-recorded in the first 10 bytes of the file. The current value of KV%, (1), is recorded. When you close the file later, you load KV% with 0 and call the subroutine to record the file statistics. That way, you know a file wasn't properly closed if its validity code is 1 the next time you open it. If no additions or deletions are made to a file during a run, the 1 never gets recorded. It doesn't matter whether or not you've closed a file properly if you've simply been reading from it.

Let's emphasize that the file synchronization and file validity functions of the labeled file handler are purely optional. If you prefer, you can ignore them, modify them or delete them entirely.

Initializing a Labeled File

When you initialize a labeled file, you are simply setting the count of records to zero, the last deletion pointer to zero and the next record pointer to 2. (If the logical record length is less than 10, the next record pointer is loaded with the number of the first logical record beyond the 10th byte.) Subroutine 58408 initializes the file statistics for any open file, where the file number is specified by PF%. For example, your call to initialize the statistics for file 4 is:

```
PF=4:GOSUB584Ø8
```

In many applications, you will only need to initialize a file once, just before your original setup. An accounts receivable customer master file or an inventory master file are examples. In other applications or for other files within an application, you may initialize daily or monthly.

In an accounts receivable application, for example, you might have a file of customer charges and payment transactions that is initialized back to 0 logical records at the beginning of each month.

After calling subroutine 58408, the file statistic, L0%(PF%,1), points to the logical record number where the first record will be added. It is computed as the first logical record beyond the 10-byte label, using the following formula:

```
First Logical Record Number in a Labeled File =  9/LL(PF)+2
```

. . . where PF% is the file number, and LL%(PF%) is the logical record length for the file. Here are the possible initial values of L0%(PF%,1), based on logical record length in bytes:

RECORD LENGTH	FIRST LOGICAL RECORD
1	11
2	6
3	5
4	4
5	3
6	3

7	3
8	3
9	3
1Ø bytes and more	2

Loading the File Statistics

Just after opening a labeled file, you will need to load the file statistics into the L0% array. This is done by loading PF% with the file number and then calling subroutine 58405. Here is the logic that might be used to open "ARMASTER" as file 3, load its statistics and check on its validity:

```
12ØØØ FS$="ARMASTER:2" :PF=3: GOSUB5825Ø : GOSUB584Ø5
12Ø1Ø IF LØ(PF,4)=Ø THEN 12Ø3Ø
12Ø2Ø PRINT"A/R MASTER FILE WAS NOT CLOSED PROPERLY LAST RUN..."
12Ø21 PRINT"PLEASE VERIFY THAT ANY ACCOUNTS YOU ADDED RECALL PRO
PERLY."
12Ø3Ø 'CONTINUE WITH PROGRAM LOGIC HERE...
```

Adding a Record

To add a new record to a labeled file, you will need to be familiar with two subroutines in the handler. Subroutine 58420 GETs and fields the next available logical record. Subroutine 58430 adds it to the file and updates the file statistics after you've LSET or RSET the new data into its fields. Before calling either of these subroutines, PF% must contain the desired file number.

After the return from subroutine 58420, LR%(PF%) will contain the logical record number that the handler has selected as the next location for the add. If there have been deletions within the file, the record selected for an add will be the most recently deleted record number.

When adding, if you wish to force the new record to be at the end of the file, you can call 58421 instead of 58420. This bypasses the logic that selects the most recently deleted record number.

Once you've gotten and fielded the record with your call to subroutine 58420 or 58421, your program logic may allow the entry of the data, and each field should be LSET or RSET with the contents of the new record. If you wish, you may allow the operator to decide not to add the record. No file statistics have been permanently altered by subroutines 58420 and 58421. (However, if you've already LSET or RSET data into the fields, you should call subroutine 58220 to restore the contents of the buffer.)

When the data has been entered into the fields, your next step is to call subroutine 58430. This does the PUT for you, recording the logical record onto the diskette. Then it adds 1 to the count of active records. If you are replacing a previously deleted record, the last deletion pointer is updated. If you are adding at the end of the file, the pointer to the next record is incremented. Finally, the new file statistics are recorded into the first 10 bytes of the file.

Here is an example of how your logic might be organized to add a record to file 3:

```
PF=3 : GOSUB5842Ø      'GET AND FIELD NEXT AVAILABLE RECORD
LINEINPUT .......      'LOGIC TO ENTER THE DATA IS HERE
LSET ............      'LOGIC TO LSET DATA INTO FIELDS IS HERE
PF=3 : GOSUB5843Ø      'WRITE THE NEW RECORD AND UPDATE STATS.
```

Changing the Contents of an Existing Record

The file statistics are not affected when you are simply making changes to one or more fields in a logical record. You need only load PF% with the file number, call subroutine 58200 to get the desired record, LR%(PF%), LSET or RSET the changed data, and call subroutine 58300 to record the new contents.

Deleting a Logical Record

Before deleting a logical record, you must bring it into the file buffer. To do this, you load PF% with the file number; load LR%(PF%) with the desired logical record number, and you GOSUB 58200. Then you simply GOSUB 58440.

Subroutine 58440 handles the deletion logic for you by linking the current logical record into a chain of deleted records. The first 2 bytes of the record point to the prior deletion in the chain. The remainder of the current logical record is filled with a string of ASCII 255's. Then the record is PUT, and the file statistics are updated.

As an example, let's suppose you want to delete logical record 302 of file 1. The commands are:

```
PF=1: LR(PF)=3Ø2: GOSUB582ØØ 'GET AND FIELD THE LOGICAL RECORD
GOSUB5844Ø                   'DELETE THE CURRENT RECORD
```

Your program should prevent the possibility of deleting a record that has already been deleted. You can tell that a record has already been deleted by the presence of ASCII 255's in each byte beyond the first two. Thus, if NM$ is an alphanumeric field starting in the 6th byte of your logical record, you know it is a deleted record if ASC(NM$)=255. You will need to test on a portion of your record that won't normally be an ASCII 255. In applications where every byte of your logical record has the possibility of containing the ASC(255) code, you may need to design a special 1-byte field as your deletion indicator.

From what we've discussed so far about deletions you can see that for deletion handling under this system, your logical record length must be at least 3 bytes. I've found that, in practice, this limitation is rarely a problem. Most often, when I am working with logical record lengths under 3 bytes, I don't need the deletion capabilities anyway.

Reading a Labeled File Sequentially

Reading a labeled random disk file sequentially is easy. I most often just use a FOR-NEXT loop. Remember that the first logical record number of a labeled file is normally 2. The last logical record number for any labeled file, PF%, is L0%(PF%,1)–1. We skip the processing or printing of deleted records by checking for ASCII 255's within the record.

Shown below is the logic pattern you could use to read file 3. It assumes that XY$ is an alphanumeric field within each logical record and that the first byte of XY$ would equal ASCII 255 only in the event of a deletion.

```
1851Ø PF=3                        'LOAD THE FILE NUMBER
1852Ø FOR X = 2 TO LØ(PF,1)-1     'FOR X = FIRST RECORD TO LAST
1853Ø LR(PF)=X:GOSUB582ØØ         'GET AND FIELD THE LOGICAL RECORD
1854Ø IF ASC(XY$)=255 THEN 1859Ø  'SKIP IF IT'S A DELETED RECORD

1855Ø 'Printing or processing logic goes here. You may use a GOSUB
or "ON GOSUB" to do the printing or processing elsewhere.

1859Ø NEXT                        'REPEAT FOR NEXT LOGICAL RECORD
```

The logic shown above assumes that the record length is at least 10 bytes long, because you are starting the sequential read at logical record 2. To make the routine workable for a file having any logical record length, you can replace line 18520 as follows:

```
1852Ø FOR X = 9/LL(PF)+2 TO LØ(PF,1)-1
```

Before starting your FOR-NEXT loop to read a file, you may want to test L0%(PF%,3) for 0. If it is 0, you know that there are no active records in the file, so there is no point in beginning the sequential read.

Notice that the sample file-reading routine can be used for multiple purposes within a program. Let's suppose, for example, that you want to be able to print the contents of file 3 in four different report formats. You can make the file-reading routine into a subroutine by adding a "RETURN" command after the "NEXT" in line 18590. Before calling subroutine 18500, load a variable, RN%, to indicate which of the four reports you want to print. Then in line 18550 use the following:

```
1855Ø ON RN% GOSUB 2Ø1ØØ,2Ø2ØØ,2Ø3ØØ,2Ø4ØØ
```

At line 20100 we would have a subroutine that prints one line of data in the format required for report 1. At line 20200 you would have a subroutine that prints a line of data in the format required for report 2. At 20300 and 20400, you would handle report formats 3 and 4.

Closing a Labeled File

If you want to take advantage of the validity-checking capabilities of the labeled file handler, your procedure for closing a labeled file involves, first, loading KV% with 0, and then, calling subroutine 58400 to record the file statistics. Finally, you call subroutine 58290 to do the actual close. Before doing these calls, load PF% with the file number to be closed.

Suppose files 3, 4 and 5 are labeled files, and files 1 and 2 are not. Your closing logic is:

```
KV%=Ø
FOR PF=3 TO 5 : GOSUB584ØØ : NEXT
FOR PF=1 TO 5 : GOSUB5829Ø : NEXT
```

Because it will modify the first 10 bytes, be sure not to call subroutine 58400 for an unlabeled file!

"LFH/DEM" — Labeled Files Demonstration

"LFH/DEM" is a program that demonstrates the random files handler with a labeled file. You can use it to get a better understanding of the concepts we've discussed and to test any modifications you may wish to make to the handler.

For demonstration purposes, LFH/DEM maintains a list of names and telephone numbers as file 1. Each logical record within the file is 36 bytes long, consisting of a 24-byte field for the name and a 12-byte field for the phone number.

When you run the program, it will first ask you to specify the file name to be used:

```
RANDOM DISK FILE HANDLER -- LABELED FILES DEMONSTRATION

FILE NAME:                ............
```

You can type any valid file name and drive number. For example, if you want to use "TESTFILE" as your file name on disk drive 0, you can enter, "TESTFILE:0" (without quotes).

Then the program will open the file you specified and ask you if you want to initialize the file. If it's a new file, you must answer "yes" by typing <Y>. If it's a file you've already initialized with the demo program, you can answer "no" by typing <N>. If it's an old file that you want to restore to zero active records, answer <Y> to re-initialize it.

Next the computer will show you 3 of the 5 file statistics being maintained:

NEXT EOF The next logical record number at the end of the file.

LAST DEL The most recently deleted logical record number.

COUNT The current count of active logical records in the file.

Then the program will allow you to enter one-letter commands to test out the various capabilities of the random disk file handler. The illustration below shows how your video display will look after you've started the program, opened a file, and initialized it:

```
RANDOM DISK FILE HANDLER -- LABELED FILES DEMONSTRATION


FILE NAME:                  TESTFILE:Ø
OPENING AS FILE 1...
INITIALIZE FILE?  (Y/N)     Y
INITIALIZED.
NEXT EOF = 2 ,  LAST DEL = Ø , COUNT = Ø

DEMONSTRATION COMMANDS ARE:

<A> ADD A RECORD    <R> RECALL A RECORD     <C> CHANGE A RECORD
<D> DELETE A RECORD <L> LIST ACTIVE RECORDS <E> CLOSE AND END
COMMAND? (A,R,C,D,L OR E)
```

Since you'll be starting out with an empty file, the first response you should give to the command request is to type <A> and press <ENTER> to add a new record. The displayed message will then request that you enter a name (up to 24 characters), and next, a phone number (up to 12 characters).

When you press <ENTER> after typing the phone number, the computer will display a message, showing you that it is adding the record, and indicating the logical record number that is being used for the new record. It will then show you the newly updated statistics for the file. Here's how it will look if you add "JOHN JONES" as the first name:

```
COMMAND? (A,R,C,D,L OR E)    A
TYPE THE NAME:               JONES, JOHN
TYPE THE PHONE NUMBER:       333 333-3333
ADDING AS RECORD NUMBER:     2
NEXT EOF = 3 ,  LAST DEL = Ø ,  COUNT = 1
```

Notice that the computer added the first name and phone number entry as logical record 2. (Remember that logical record 1 is not used because that's where our file statistics are being stored.)

Notice also that the "NEXT EOF" pointer has been bumped up to 3. The next record you add will be added as logical record 3. The count is now 1 because you have 1 active record.

To recall and redisplay the record you've added, use the <R> command. After entering <R>, the computer requests the record number to be recalled. To get the first record in the file, enter <2> as the record number; the disk is accessed, and the data is redisplayed:

```
COMMAND? (A,R,C,D,L OR E)   R
RECORD NUMBER?              2
NAME:                       JONES, JOHN
PHONE NUMBER:               333 333-3333
```

Continuing the demonstration, you can add several more records with the <A> command and (when desired) recall them by number with the <R> command. The <L> command lists the current contents of the file. Here's how it might look after you've added 4 records:

```
COMMAND? (A,R,C,D,L OR E)   L
RECORD 2                    JONES, JOHN
                            333 333-3333
RECORD 3                    ROBERTS, BILL
                            333 332-3214
RECORD 4                    JOHNSON, SAM
                            432 331-3322
RECORD 5                    ADAMS, SUE
                            931 987-6543
```

Now, let's suppose that, after listing the file, you want to make a correction to the phone number in record 4. In response to the command request, you simply enter <C> for change. The computer will request that you enter the record number to be changed, and then it will get and display the current contents. Since you don't want to change the name, you can press <ENTER> instead of typing a new name. Then you can type the corrected phone number. Here's how the dialoge will appear on your screen:

```
COMMAND? (A,R,C,D,L OR E)   C
RECORD NUMBER?              4
NAME:                       JOHNSON, SAM
PHONE NUMBER:               432 331-3322
CHANGE NAME TO:
CHANGE PHONE NUMBER TO:     432 888-3333
```

If you now use the <R> or <L> command, you will see that the phone number in record 4 has been changed.

To show how deletions are handled, let's first delete record number 3. To do it, enter <D> in response to the command request, and <3> for the record number to be deleted. The computer displays the contents of the record and a message indicating that it's deleting it. Then the new file statistics are displayed. Assuming that you've entered the 4 records as shown above, your screen will display:

```
COMMAND? (A,R,C,D,L OR E)   D
RECORD NUMBER               3
NAME:                       ROBERTS, BILL
```

```
PHONE NUMBER:                333 332-3214
DELETING...
NEXT EOF = 6 ,  LAST DEL = 3 , COUNT = 3
```

Notice that the "LAST DEL" file statistic reflects the last record number deleted, and the count has been decremented by 1.

To illustrate a point, let's also delete record 4. Our file statistics are now:

```
NEXT EOF = 6 ,  LAST DEL = 4 ,  COUNT = 2
```

If, at this point, you list the file with the <L> command, the display will show:

```
COMMAND? (A,R,C,D,L OR E)    L
RECORD 2                     JONES, JOHN
                             333 333-3333
RECORD 5                     ADAMS, SUE
                             931 987-6543
```

Now, if you add a new record, it will be added as record 4, because record 4 was the last one deleted. The statistics after the "add" will be:

```
NEXT EOF = 6,  LAST DEL = 3 ,  COUNT = 3
```

If you add another record, it will be added as record 3, because record 3 was the previous record in the deletion chain. The statistics are now:

```
NEXT EOF = 6 , LAST DEL = Ø ,  COUNT = 4
```

Notice, now, that the "LAST DEL" statistic is again 0. At this point, you have reached the end of the chain of internal deletions. The next record you add will go at the end of the file, logical record position 6. After adding it, the statistics are:

```
NEXT EOF = 7 , LAST DEL = Ø ,  COUNT = 5
```

When you are finished adding, deleting and changing records, you can use the <E> command. It ends the demonstration by putting a valid-termination indicator in the file statistics, and "locks-in" the data by closing the file.

As you can see, the "LFH/DEM" program can give you a good understanding of how file statistics are handled, and how the random disk file handler subroutines can be called from a BASIC program. In actual applications, you will want to add fancy video display and keyboard routines so your program is more "operator-oriented." Also, you won't normally display all the file statistics in a real application. You might want to put in a few calculations so that the first record, as far as the operator is concerned, is 1, instead of 2.

When using the demo program, be sure to check out the disk error-handling capabilities. You might try, for example, removing a diskette before an add command. You can press <ENTER> to retry, and you'll see that the program successfully resumes when you re-insert the diskette. If you've got 8 or more records in your file, you can simulate a disk error by opening the disk drive door during a list command. When you resume, the error message will be erased from the screen, the cursor position will be restored, and it will look as if nothing happened!

Setting Up LFH/DEM

Here are the steps you need to do to write the LFH/DEM program.

1. Type-in or load the random disk file handler subroutines exactly as they are shown. They will occupy lines 58000 through 58996.

2. Verify that line 58001 specifies a logical record length of 36 bytes for file 1. It should contain the command, "LL(1) = 36".

3. Modify line 58010 so that it handles 2 fields for file 1. The two fields used in the demonstration program are FH$(1) and FH$(2). FH$(1) will hold the 24-byte name, and FH$(2) will hold the 12-byte phone number. After your modification, it will read:

```
58Ø1Ø FIELD PF, LS*LL(PF) AS FD$(PF), 24 AS FH(1), 12 AS FH(2)
```

4. You can, if you wish, add the following to the DIM statement in line 58000 to save a few bytes of memory:

```
,LØ (A%,5) ,FØ$(5)
```

Since you'll only be using 1 file in the demonstration program, you can make a few other optional modifications to save memory. You can, for example, change the "3" in line 58000 to 1, and you can delete the references to LL(2) and LL(3) in 58001. You can delete lines 58020 through 58031.

5. Merge or type-in the labeled random disk file handler subroutines. They occupy lines 58400 through 58441.

6. Type-in, or modify line 58995 so that, if you get an unrecoverable disk error, you can resume to a safe point in the program. As you'll see, line 1000 will be a good line to resume to, so line 58995 should read:

```
58995 IF A$ <> CHR$(13) THEN RESUME 1ØØØ
```

7. Now that you've got the standard subroutines you'll need, enter the body of the LFH/DEM program as it is shown and documented below.

Figure 7.1 — *LFH/DEM*

```
Ø 'LFH/DEM
1 CLEAR1ØØØ
2 DEFINTA-Z:DEFSTRF
5Ø GOSUB58ØØØ
6Ø KS%=VAL(TIME$)*1ØØ+VAL(MID$(TIME$,4)):KV%=1

1ØØØ CLS:PRINT"RANDOM DISK FILE HANDLER -- LABELED FILES DEMONST
RATION":PRINTSTRING$(63,"=")
1Ø1Ø LINEINPUT"FILE NAME:                    ";FS$
1Ø2Ø PRINT"OPENING AS FILE 1...":PF=1:GOSUB5825Ø:GOSUB584Ø5
1Ø3Ø LINEINPUT"INITIALIZE FILE?  (Y/N)      ";A$
1Ø4Ø IFA$="Y"THENPF=1:KV%=Ø:GOSUB584Ø8:PRINT"INITIALIZED.":KV%=1
1Ø5Ø IFLØ(1,4)=1THENPRINT"MAY BE INVALID - CHECK THE CONTENTS...
"
1Ø6Ø GOSUB3ØØØ
1Ø7Ø PRINT
1Ø71 PRINT"DEMONSTRATION COMMANDS ARE:"
1Ø72 PRINT
```

```
1Ø73 PRINT"<A> ADD A RECORD    <R> RECALL A RECORD     <C> CHANG
E A RECORD"
1Ø74 PRINT"<D> DELETE A RECORD <L> LIST ACTIVE RECORDS <E> CLOSE
 AND END"
1Ø75 PRINT
2ØØØ PRINT:LINEINPUT"COMMAND? (A,R,C,D,L, OR E)  ";CD$
2Ø1Ø CD%=INSTR("ARCDLE",CD$):IFLEN(CD$)<>1ORCD%=ØTHEN2ØØØ
2Ø11 ONCD%GOTO22ØØ,23ØØ,23ØØ,23ØØ,24ØØ,25ØØ

22ØØ PF=1:GOSUB5842Ø
221Ø LINEINPUT"TYPE THE NAME:               ";A$
2211 LSETFH(1)=A$
2212 LINEINPUT"TYPE THE PHONE NUMBER:       ";A$
2213 LSETFH(2)=A$
222Ø PRINT"ADDING AS RECORD NUMBER:   ";LR(1)
223Ø GOSUB5843Ø
224Ø GOSUB3ØØØ:GOTO2ØØØ

23ØØ LINEINPUT"RECORD NUMBER?               ";A$
23Ø1 LR(1)=VAL(A$):IFLR(1)<2ORLR(1)>=LØ(1,1)THENPRINT"INVALID
RECORD NUMBER...":GOTO2ØØØ
231Ø PF=1:GOSUB582ØØ
232Ø IFMID$(FH(1),3,1)=CHR$(255)THENPRINT"DELETED...":GOTO2ØØØ
233Ø PRINT"NAME:";TAB(28)FH(1)
2331 PRINT"PHONE NUMBER:";TAB(28);FH(2)
234Ø IFCD$="C"THEN235ØELSEIFCD$="D"THEN236ØELSEGOTO2ØØØ

235Ø LINEINPUT"CHANGE NAME TO:              ";A$
2351 IFA$<>""THENLSETFH(1)=A$
2352 LINEINPUT"CHANGE PHONE NUMBER TO:      ";A$
2353 IFA$<>""THENLSETFH(2)=A$
2355 PF=1:GOSUB583ØØ:GOTO2ØØØ

236Ø PRINT"DELETING...":PF=1:GOSUB5844Ø
2365 GOSUB3ØØØ:GOTO2ØØØ

24ØØ IFLØ(1,3)=ØTHEN2ØØØ
241Ø FORX=2TOLØ(1,1)-1
242Ø PF=1:LR(1)=X:GOSUB582ØØ
243Ø IFMID$(FH(1),3,1)=CHR$(255)THEN245Ø
244Ø PRINT"RECORD";X;TAB(28);FH(1)
2441 PRINTTAB(28);FH(2)
245Ø NEXT:GOTO2ØØØ

25ØØ KV%=Ø:PF=1:GOSUB584ØØ:GOSUB5829Ø:CLS:END

3ØØØ PRINT"NEXT EOF =";LØ(1,1);",  LAST DEL =";LØ(1,2);", COUNT
=";LØ(1,3):RETURN
```

LFH/DEM Line Comments

1 Clear string storage for 1000 bytes.

2 Define all variables as integer.

:Define variables starting with "F" as string.

50 Call subroutine 58000 in the random disk file handler to initialize and dimension the arrays for disk file access.

60 Load KS% with a 2-byte representation of the date. This will be stored in the label of our disk file.

:Load KV% with 1 so that an "invalid file" code will be recorded into the disk label if the program is terminated without a close.

1000 (Begin mainline demonstration.)

:Clear the screen.
:Print title, "Random Disk Demo...".
:Underline it with a graphics bar.

1010 Allow operator to enter the name of the file to be used for the demonstration.

1020 Print message indicating we're opening the file.

:Set file number indicator, PF%, equal to 1.
:Call subroutine 58250 to open the file with error handling.
:Call subroutine 58405 to load the file statistics into L0%(1,1) through L0%(1,5).

1030 Let the operator enter "Y" if the file is to be initialized to zero logical records, "N" if it is to be used as-is.

1040 If the operator entered "Y", then specify that you'll be initializing file 1,

:Load validity indicator, KV%, with 0 so that it will start out as a valid file,
:Call subroutine 58408 to initialize the file,
:Print a message to show that it's initialized,
:Reload validity indicator, KV%, with 1.

1050 If the validity indicator for file 1 is 1, then print a message indicating that the file may not have been closed properly during the last run.

1060 Call subroutine 3000 to show the file statistics you're starting out with.

1070 Display a message showing the operator the valid commands that may be used in this demonstration program.

2000 (Accept, validate, and execute next command to be demonstrated.)

:Let operator enter the 1-letter command code and store it in CD$.

2010 Convert command, "A,R,C,D,L, or E" to a number, CD%, ranging from 1 to 6.

:If the length of the command is not 1 or if the code was not in the list of valid commands,then go to 2000 to force entry of a valid command.

2011 Otherwise, go to the proper line based on the command that was entered.

2200 (Process an <A> command to add a record at the next available disk file location.)

:Set file number, PF%, to 1.
:Call subroutine 58420 to field and get the record to be added.

2210 Let the operator type in a name.

2211 LSET the name into the field, FH$(1).

2212 Let the operator type in a phone number.

2213 LSET the phone number into the field, FH$(2).

2220 Print a message showing which logical record you are using to add the new information.

2230 Call subroutine 58430 to record the new information to disk and to update the file statistics.

2240 Call subroutine 3000 to show the new file statistics.

:Go to line 2000 to let the operator enter another command.

2300 (Process an <R>, <C> or <D> command. Recall and display the contents of a logical record. If a change or delete, go to the proper routine after the record is displayed.)

:Let the operator enter the record number.

2301 Load the record number entered into LR%(1).

:If it's less than 2, (the first record of the file), or if it's past the last record number of the file, then print a message indicating invalid record number, and go to 2000 to allow re-entry of the command.

2310 Now that you know LR%(1) contains a valid record number, load PF% with 1 so the subroutines will know you're using file 1.

:Call subroutine 58200 to get and field the desired record.

2320 We know that the logical record was previously deleted by testing any byte past the first 2 bytes of it for CHR$(255).

:Test on the 3rd byte of the first field. If it's a CHR$(255) then print a message showing it was previously deleted, and
:Go to 2000 to allow entry of a new command.

2330 If the record is not a deleted record, print the name contained in the field, FH$(1).

2331 Print the phone number, contained in field FH$(2).

2340 Now that you've displayed the data, if the command entered was <C> for change, go to 2350,

:Otherwise, if the command was <D>, go to 2360 to delete.
:Otherwise, you only wanted to display the data, so go to 2000 to allow entry of the next command.

2350 (Process a "C" command to change data in an existing logical record.)

:Allow entry of a new name.

2351 If the operator typed something, rather than just pressing <ENTER> ,then LSET the new data into the name field, FH$(1).

2352 Allow entry of a new phone number.

2353 If the operator typed something, rather than just pressing <ENTER> ,then LSET the new data into the phone number field.

2355 Load desired file number, PF%, with 1.

:Call subroutine 58300 to record the changed information.
:Go to line 2000 to allow entry of the next command.

2360 (Process a <D> command to delete the logical record that has just been recalled and displayed.)

:Load the desired file number with 1.
:Call subroutine 58440 to delete the current logical record.

2365 Call subroutine 3000 to display the new file statistics.

:Go to line 2000 to allow entry of a new command.

2400 (Process the <L> command to list all active logical records and their contents.)

:If L0%(1,3) is zero, there are no active records in the file, so go to 2000 to allow entry of another command.

2410 For each logical record in the file, using X as a counter,

2420 Load PF% with 1 to indicate the desired file number.

:Load LR%(1) with the desired record number.
:Call subroutine 58200 to GET and field the logical record.

2430 If it's a deleted record then skip it by jumping to line 2450.

2440 Otherwise, print the record number, and the contents of the name field, FH$(1).

2441 Print the phone number, contained in field FH$(2).

2450 Repeat from line 2410 for the next logical record.

:When done, go to line 2000 to allow entry of a new command.

2500 (Process the <E> command to close the file and end the demo.)

:Load validity indicator, KV%, with 0 to indicate a proper termination.
:Set file number, PF%, equal to 1.
:Call subroutine 58400 to write the file statistics to disk.
(Our reason for doing this now is to change the validity indicator in the file to zero.)
:Call subroutine 58290 to close the file with error handling.
:Clear the screen.
:End the program by going back to READY.

3000 (Subroutine to display certain file statistics.)

:Display "next record end of file", "last deletion" record number and the count of active records for file 1.
:Return to the main program.

Detail Files

You'll find that a large number of computer applications fall into a pattern where you have a "master file" and a "detail file." The master file contains the primary data elements, such as customer names and addresses or inventory item descriptions. The detail file contains data that is related to individual records within the master file, such as purchase or payment transactions. Each record in the detail file relates to a record in the master file. Each record in the master file may have any number of detail records related to it.

In an accounts receivable system, for example, you might have a master file and a transaction detail file. The master file has an individual record for each customer. The detail file contains all charges and payments for all the customers.

For any individual customer in the master file, there may be any number of charges and payments in the detail file. Here's an example of a master file containing 4 logical records, and a detail file containing 5 logical records.

```
Customer File  (MASTER)

1  ABC Supply
2  Jones Distributing
3  Southwest Mfg. Company
4  Z and Z Wholesale

Transaction File  (DETAIL)

1  May 1, Payment Received, $200, Jones Distributing
2  May 1, Charge Purchase,  $150, Z and Z Wholesale
3  May 3, Charge Purchase,  $400, ABC Supply
4  May 4, Charge Purchase,  $100, Jones Distributing
5  May 4, Charge Purchase,  $700, Jones Distributing
```

The task of the program is to quickly recall the detail that is related to any master record for printouts or video display inquiries. For example, if you call up Jones Distributing on the video display, you want to see their charges and purchases:

```
Jones Distributing

May 1, Payment Received,    $200
May 4, Charge Purchase,     $1ØØ
May 4, Charge Purchase,     $7ØØ
```

If you inquire into ABC Supply's account, you want to see:

```
ABC Supply

May 3, Charge Purchase,     $4ØØ
```

Many other applications fall into the same pattern. In some systems, more than one detail file may be related to the same master file. In other systems, a detail file may be related to more than one master file. Sometimes there may be multi-level relationships, where detail files can act as "masters" to other detail files. Here are some examples:

APPLICATION	MASTER FILE	DETAIL FILE
Accounts Receivable	Customers	Charges & Payments
Accounts Receivable	Customers	Alternate Addresses
Manufacturing	Finished Goods	Components
Job Cost	Jobs	Job Phases
Job Cost	Job Phases	Costs Incurred
Church Membership List	Last Names	Members of each family

There are many ways to handle relationships such as these. One way is to design your master file so that each logical record has a fixed number of "buckets" for each detail item that may occur. The disadvantage of this method is that, for some records, space will be wasted, and for others, we may not have enough bucket fields.

Another method is to sort the detail file so it is in the same sequence as the master file to which it is related. Then, for a printout, you can read both files sequentially. You get and print a record from the master file. Then you go to the detail file and get and print records as long as they relate to the current master. When you come to a detail record that doesn't match the master record, you print the next master record, and continue. The disadvantage of this approach is that we need a time-consuming sort whenever we add a batch of detail records. It's not practical to quickly inquire into any account to see the related detail.

The approach that we will discuss in this section involves linking the detail records to the related master records. From each master record, you can follow a chain through each of its detail records. When a new detail record is added, it is linked into the chain. When a detail record is deleted, it is delinked from the chain to its master record. This interactive linking method is made possible by the capablity we have with random files, to refer to, and access any record by logical record number.

Within any master record, you have a "chain head." The chain head consists of two pointers to records in a related detail file. One pointer gives the logical record number of the first detail record that is related. The other pointer gives the logical record number of the last detail record that is related.

Within any detail record, you have three pointers. One pointer tells the logical record number in the master file to which it is related. Another pointer, gives the previous detail record number, if any, that is related to the same master record. The third pointer specifies the next detail record, if any, that is related to the same master record.

If a master record has no related detail records, both of its pointers will be zero. The first detail record in a chain will have a "previous" record pointer equal to zero. The last detail record in a chain will have a "next" record pointer equal to zero. Here's the overall scheme of pointers:

In each master record:

- Logical record number of its first detail record, if any.
- Logical record number of its last detail record, if any.

In each detail record:

- Logical record number of the master record to which it is related.
- Logical record number of the previous detail record that is related to the same master record, if any.
- Logical record number of the next detail record that is related to the same master record, if any.

To illustrate, let's repeat the first example, this time showing how the pointers would be organized:

Customer File (MASTER)

Rec#	First	Last	Data Contained...............
1	3	3	ABC Supply
2	1	5	Jones Distributing
3	Ø	Ø	Southwest Mfg. Company
4	2	2	Z and Z Wholesale

Transaction File (DETAIL)

Rec#	Master	Prev	Next	Data Contained...............
1	2	Ø	4	May 1, Payment Received, $2ØØ
2	4	Ø	Ø	May 1, Charge Purchase, $15Ø
3	1	Ø	Ø	May 3, Charge Purchase, $4ØØ
4	2	1	5	May 4, Charge Purchase, $1ØØ
5	2	4	Ø	May 4, Charge Purchase, $7ØØ

Notice that you don't need to repeat the customer names in the transaction file. Since each detail record contains a pointer to the related master record, the customer name for a transaction can be accessed by getting the indicated record in the customer file.

Now, let's suppose that you want to recall all the transactions for Jones Distributing. First, get record number 2 in the customer file. The "first" pointer indicates that record 1 in the transaction file contains the first transaction for Jones. After you get and display that transaction, follow the "next" pointer to record 4 in the transaction file. After displaying the contents of record 4, follow its "next" pointer to record 5. After displaying it, you find that the "next" pointer is zero, so you know you've displayed all transactions for Jones Distributing.

Let's suppose you want to add another transaction for Jones Distributing as logical record 6 in the detail file. You find that the "last" pointer for Jones in the master file is 5, so go back to logical record 5 in the detail file, and change its "next" pointer to 6. In record 6 make the "previous" pointer 5 and the "next" pointer 0. In logical record 2 of the master file, change the "last" pointer to 6.

The logic is slightly different if you are adding the first transaction for a customer, or if you are deleting a transaction. The detail file handler, which we shall discuss, handles all that logic for you.

The Detail File Handler

The detail file handler is a set of subroutines and procedures for interactively handling "master-detail" file relationships. It is designed to work in conjunction with the random disk file handler subroutines, but you can use the same logic with any program you may write. You may use the detail file handler with labeled or unlabeled files, but in most cases, a master file and its detail file will both be labeled, so that additions and deletions can be handled automatically.

One of the advantages of the detail file handler is that standard variable names can be used. The variables for all the pointers are double dimensioned, so that you can have as many master-detail file relationships in your program as you wish. Programming a complex file structure becomes a matter of specifying the required fields in your master and detail files, and calling the subroutines you need.

The detail file handler also has some important validity-checking features. As you've seen, the pointer relationships are quite critical for the proper functioning of the system. If a "broken-chain" occurs, due to a programming error, an equipment malfunction or the use of a detail file that is not as current as its master file, the subroutines will sense the problem.

Detail File Handler

Get Next Record in Chain:

```
58700 PF=P2:IFLX(PD)=0THENLX(PD)=CVI(FF$(PD,3)):P0=0ELSELX(PD)=C
VI(FF$(PD,2)):P0=LR(P2)
58701 A$="":IFLX(PD)=0THENRETURNELSELR(P2)=LX(PD):GOSUB58200
58702 IFCVI(FF$(PD,0))<>LR(P1)THENA$="E"ELSEA$=""
58703 RETURN
```

Link and Add a Record to Chain:

```
58710 GOSUB58712:P0=LR(P2):PF=P2:GOSUB58430:GOSUB58714:IFA$="E"T
HENRETURNELSEGOSUB58718:RETURN
```

Load Pointers into a Detail Record to be Linked:

```
58712 LSETFF$(PD,0)=MKI$(LR(P1)):LSETFF$(PD,1)=FF$(PD,4):LSETFF$
(PD,2)=MKI$(0):RETURN
```

Link Prior Detail Record to New Detail Record:

```
58714 A$="":PF=P2:LR(PF)=CVI(FF$(PD,4)):IFLR(PF)=0THENRETURN
58715 GOSUB58200:IFCVI(FF$(PD,0))<>LR(P1)THENA$="E":RETURNELSELS
ETFF$(PD,2)=MKI$(P0):GOSUB58300:RETURN
```

Link Master Record to New Detail Record:

```
58718 LSETFF$(PD,4)=MKI$(P0):IFFF$(PD,3)=MKI$(0)THENLSETFF$(PD,3
)=FF$(PD,4)
58719 PF=P1:GOSUB58300:RETURN
```

Delink a Detail Record:

```
58740 PF=P2:P0=LR(P2):P3=CVI(FF$(PD,1)):P4=CVI(FF$(PD,2))
58741 IFP4=0THENLSETFF$(PD,4)=FF$(PD,1)
58742 IFP3=0THENLSETFF$(PD,3)=FF$(PD,2)
58743 IFP3>0THENLR(P2)=P3:GOSUB58200:LSETFF$(PD,2)=MKI$(P4):GOSU
B58300
58744 IFP4>0THENLR(P2)=P4:GOSUB58200:LSETFF$(PD,1)=MKI$(P3):GOSU
B58300
58745 PF=P1:GOSUB58300:PF=P2:LR(P2)=P0:GOSUB58200:RETURN
```

Detail File Handler — Variables Used

Simple Variables

A$ Contains "E" upon return from subroutines 58700, 58710 or 58714, to indicate that a "broken-chain" was encountered and the subroutine was aborted.

PO% Used as a working variable within the subroutines. After a call to subroutine 58700, it holds the previous record number in the chain to aid in error recovery. Before a call to 58714 or 58718, it holds record number of the detail record being linked.

P1% Provided by your mainline program to specify the file number of the master file. (P1% is not modified by the handler subroutines.)

P2% Provided by your mainline program to specify the file number of the detail file. (P2% is not modified by the handler subroutines.)

PD% In systems having multiple "master-detail" file relationships, PD% is provided by your mainline program to specify which relationship to process. In systems with only one "master-detail" relationship, PD% is 0. (PD% is not modified by the handler subroutines.)

P3%,P4% Used as temporary storage by the delink subroutine. Otherwise they are not used. During a delink, P3% stores the "previous", and P4% stores "next" of the detail record that is being delinked.

PF% Loaded by the handler subroutines, based on the content of P1% or P2%. It specifies the file number to be used when a GET or PUT subroutine is called.

Arrays

FF$(PD%,0) A 2-byte field within the detail record, used for storing the pointer to the related master record.

FF$(PD%,1) A 2-byte field within the detail record, used for storing the pointer to the previous detail record.

FF$(PD%,2) A 2-byte field within the detail record, used for storing the pointer to the next detail record.

FF$(PD%,3) A 2-byte field within the master record, used for storing the pointer to its first detail record.

FF$(PD%,4) A 2-byte field within the master record, used for storing the pointer to its last detail record.

LX%(PD%) Loaded with 0 by your main program before a call to subroutine 58700, to get the first record in the detail chain specified by PD%. Upon return from subroutine 58700, it will be 0 if the end of the chain has been encountered; otherwise it will contain the logical record number of the next record in the chain. When it's non-zero, subroutine 58700 reads the next record in the detail chain.

LR%(PF%) Specifies the current or desired logical record number for the file number specified by PF%.

Detail File Handler — Line Comments

58700 (Get Next Record in Chain.)

:Set file number, PF%, to detail file number, as specified by P2%.
:If LX%(PD%) was loaded with zero prior to calling this subroutine, you are to read from the beginning of the chain, so, load LX%(PD%) with the "first" pointer from the master record,
:and load P0% with zero, so you'll know there was no previous record in the chain if a recovery is necessary.
:Otherwise, load LX%(PD%) with the "next" pointer from the current detail record,
:and, load P0% with the current detail record number, so you'll know what the previous record was if a recovery is necessary.

58701

:Null out A$ to indicate no error has been encountered.
:If LX%(PD%) is now zero, there is no "next" record, so you can return to the mainline program.
:Otherwise, the LX%(PD%) has the logical record number you want, you load it into LR%(P2%), and
:call subroutine 58200 to get and field the logical record.

58702

If the "master record number" pointer in the record you've gotten is not equal to the master record number whose chain you thought you were reading, then load A$ with "E" to signify an error condition.
:Otherwise, null out A$ to signify that everything's fine.

58703 Return back to the mainline program.

58710 (Link and Add a Record to the Chain.)

:Call subroutine 58712 to LSET the pointers into the new detail record.
:Save the detail logical record number in P0%.
:Set desired file indicator, PF%, to indicate the detail file.
:Call subroutine 58430 to PUT the detail record and update the file statistics.
:Call subroutine 58714 to link the prior detail record in the chain, if any, to the new detail record.
:If A$ is returned as "E", then the prior detail record specified was invalid, so abort the add by returning to the mainline program.
:Otherwise, everything looks good, so call subroutine 58718 to update the pointers in the master record,
:and we return to the mainline program.

58712 (Load pointers into a detail record to be linked.)

:Point the detail record to the curent master record.
:'Previous" record pointer in detail record is the old "last" record number as specified in the master record.
:"Next" record pointer in detail record is 0, since you are adding the new detail record at the end of the chain.
:Return to subroutine 58710, or to main program.

58714 (Link prior detail record to new detail record.)

:Null out A$ to indicate no linking error has been encountered.
:Load desired file number, PF%, with the number specified as the detail file, P2%.
:The prior detail record is what was previously the "last" detail record.
:If it is zero, then return. No link is required.

58715

:Otherwise, GET and field the prior detail record.
:If it doesn't point to the same master record, you have an error, so load A$ with "E" and return.
:Otherwise, everything's fine, so change its "next" pointer to the logical record number of the new detail record, as specified by P0%.
:Call subroutine 58300 to record the updated detail record to disk.
:Return to subroutine 58710, or to the main program.

58718 (Link Master Record to New Detail Record.)

:Update "last" pointer so that it points to the new detail logical record number, as specified by P0%.
:If the "first" pointer in the master record is zero, then this new detail record is the first, so update the first pointer, same as the "last" pointer.

58719

:Desired file number is master file, as specified by P1%.
:Call subroutine 58300 to PUT the updated master record.
:Return to 58710, or to the main program.

58740 (Delink a Detail Record from the Chain.)

:Desired file number is detail file, as specified by P2%.
:Save current detail logical record number in P0%.
:Save "previous" detail record number in P3%.
:Save "next" detail record number in P4%.

58741

If "next" detail record number is zero, then load "last" pointer in master record with "previous".

58742

If "previous" detail record number is zero, then load "first" pointer in master record with "next".

58743

:If there is a "previous" detail record, then load logical record number with "previous" record number,
:GET and field the previous logical record in the chain,
:load the "next" pointer with the "next" pointer from the delinked record,
:and, PUT the previous logical record.

58744

:If there is a "next" detail record, then load logical record number with "next" record number,
:GET and field the next logical record in the chain,
:load its "previous" pointer with the "previous" pointer from the delinked record,
:and, PUT the next logical record.

58745

:Set file number, PF%, to file specified as master.
:PUT the master record, to record its updated pointers.
:Set file number, PF%, back to file specified as the detail file.
:Load record number with the number of the record delinked.
:GET and field the detail record just delinked, so it will be in the buffer, ready to delete.
:Return to the main program.

Implementing the DFH

The detail file handler subroutines can be merged into your program exactly as they are shown, at lines 58700 through 58745. The detail file handler relies on two major subroutines in the random disk file handler. They are:

- Subroutine 58200 — GET and field logical record, LR%(PF%), where PF% specifies the disk file number.
- Subroutine 58300 — PUT physical record, PR%(PF%), where PF% specifies the disk file number.

To use the detail file handler, these subroutines and the supporting routines used by them need to be present in your program, unless you make modifications to use other routines that perform the same functions.

I've found that whenever I am using master and detail files, it's best to organize them as labeled files, so that additions, deletions, and file statistics can be handled automatically. Most of our discussions will assume that you are using the labeled file handler routines, but we'll mention the different procedures you need if you aren't using them.

Fielding Considerations

When you design a system, you need to decide which files will be used as master files and which will be used as detail files. Each master file to detail file relationship requires a set of 5 standard fields. Two of them go in each master file logical record, and three of them go in each detail file logical record. These 5 fields, each 2 bytes long, will contain the pointers required for linking.

The linking fields use the FF$(X,Y) array. The "X" dimension specifies which master-detail relationship the field is to handle. If you just have one master-detail chain in your system, the "X" dimension will be 0. For the second master-detail chain in your program, the "X" dimension will be 1. For the third, it will be 2, and so forth. The "Y" dimension is always 4, so that the 5 standard fields for any master-detail chain can be handled.

Let's take a simple example first. The accounts receivable system that we discussed earlier has just one master-detail relationship. The customer file is the master file, and the purchase and payment transaction file is the detail. If file 1 is the customer master, the fielding statement for file 1 must contain the following, in addition to the other fields you may have for the customer name, address, and balance:

```
,2 AS FF$(Ø,3), 2 AS FF$(Ø,4)
```

If file 2 is the purchase and payment transaction detail file, the fielding statement for file 2 must contain the following, in addition to the other fields you may have for the date, amount, and a code for purchase or payment:

```
,2 AS FF$(Ø,Ø), 2 AS FF$(Ø,1), 2 AS FF$(Ø,2)
```

These required fields may be at any position within the logical record layouts you are using.

Now let's suppose that you design the accounts receivable system so there are two master-detail relationships. One common approach is to separate the purchase and payment detail into two files. The customer purchase detail file could be considered the first relationship, and the customer payment detail file could be considered the second. In this case, if the customer master is file 1, the logical record fielding for file 1 will include:

```
,2 AS FF$(Ø,3), 2 AS FF$(Ø,4), 2 AS FF$(1,3), 2 AS FF$(1,4)
```

Assuming file 2 is the purchase detail file, its fielding for each logical record will include:

```
,2 AS FF$(Ø,Ø), 2 AS FF$(Ø,1), 2 AS FF$(Ø,2)
```

If file 3 is the payment detail file, its fielding for each logical record will include:

```
,2 AS FF$(1,Ø), 2 AS FF$(1,1), 2 AS FF$(1,2)
```

When you are using the standard methods explained in the discussion of the random disk file handler, these fielding commands can simply be tacked onto the end of lines 58010, 58020 and 58030, according to the file number. The fielding for file 2, for example, is done in line 58020. Assuming it has 2 data fields, DA$ and AM$, for a 3-byte date and an 8-byte amount, line 58020 might read:

```
58Ø2Ø FIELDPF,LS*LL(PF)ASFD$(PF),3ASDA$,8ASAM$,2ASFF$(Ø,Ø),2ASFF
$(Ø,1),2ASFF$(Ø,2)
```

Dimensioning and Variable Naming

Now that we've discussed how the FF$ fields are used, you can see how to dimension the FF$ array. Though you can get by without dimensioning FF$, you can, in most cases, save a few hundred bytes if you do. You should add the FF$ dimensioning command to line 58000, along with the other disk file handler DIM commands. If, as in most cases, you have only one master-detail relationship, you can add the following to line 58000:

```
,FF$(0,4)
```

If you have two master-detail relationships, you use FF$(1,4). For three, you'd use use FF$(2,4), and so forth.

You can save a few bytes by dimensioning the LX% array in line 58000. It is also dimensioned according to the number of master-detail relationships you have, minus 1. So, if you have one master-detail relationsip, the DIM statement in line 58000 includes:

```
,LX(0)
```

There are no other dimensioned variables in the detail file handler, except those we discussed for the random disk file handler, and, if you're using it, the labeled files handler.

As with the random disk file handler, all variables are assumed to be integer, unless they are explicitly defined, so you'll want to make sure that variables starting with "L" and "P" are integers. Variables beginning with "F" are always strings in the handler, so you can remove the "$" from each reference to FF$ if you use a "DEFSTR F" command early in your program.

Adding and Linking a New Detail Record

Subroutine 58710 provides the most automatic method to add a detail record, but you must be using the labeled files handler. It assumes that you have already gotten and fielded the new detail record, and you have already LSET information into the new record's data fields. It also assumes that the related master record is fielded and in its buffer.

Subroutine 58710 then loads the required pointers into the detail record and calls subroutine 58430 (of the labeled files handler) to PUT it and update the file statistics. Then the pointers in the previous detail record are updated. Finally, the pointers in the master record are updated, and it is recorded to disk.

Before calling subroutine 58710, P1% must contain the file number you are using for your master file, and P2% must contain the file number you are using for your detail file. PD% must contain the number that specifies the master-detail chain you are using, since it's used as a subscript for the FF$ array. LR%(P1%) must contain the current logical record number of the master record and LR%(P2%) contains the new logical record number in the detail file.

As an example, let's say that you have 1 master-detail relationship in the system, with file 1 being the master file and file 2 being the detail file. Here's the logic you can use to add a new detail record to the chain for master record 303:

```
PF=1:LR(1)=303:GOSUB58200    'GET AND FIELD THE MASTER RECORD
PF=2:GOSUB58420              'GET NEXT AVAILABLE DETAIL RECORD TO ADD

'Logic to LSET Data into New Detail Record Goes Here...
```

```
'You may also wish to update data in the master record here...

PD=Ø:P1=1:P2=2              'SPECIFY CHAIN, MASTER FILE, DETAIL FILE
GOSUB 5871Ø                 'ADD AND LINK THE NEW DETAIL RECORD
IF A$="E" THEN .....        'IF LINKING ERROR, PRINT MESSAGE
```

If you review the section about labeled files, you'll see that subroutine 58420 GETs and fields the next available unused record in a labeled file. If there are previous deletions in the file, 58420 will use the most recently deleted record as the record to be added. Alternatively, you can call subroutine 58421 to force the new detail record to be the next one at the end of the detail file. Upon the return from 58420 or 58421 in the above example, LR%(2) contains the detail logical record number that the system has selected.

The "PD=0" command in the example tells the handler to LSET the required pointers into fields FF$(0,0) through FF$(0,4) of the master and detail records. (If "PD=1" were used, fields FF$(1,0) through FF$(1,4) would be used.) If you have just one master-detail relationship in your program, you can preload PD%, P1%, and P2%. You don't have to load them before each detail file handler call.

As you can see, if A$ equals "E" after a call to subroutine 58710, a linking error was encountered. This means that the "last" pointer in the master record pointed to a detail record having a "master" pointer different from the current master record. If an error was encountered, the new detail record will have been added to the file, but it will not have been linked. The master record and previous detail record will not have been affected.

Notice that you LSET or RSET the data into the new detail record before calling subroutine 58710. If the detail record were a purchase or payment transaction record of an accounts receivable system, for example, the date and amount would be LSET into the required fields. You might also LSET an updated balance into the master record. Subroutine 58710 does the PUT for you, to the detail record as well as the master record.

In some programs, it may be important for you to know that after a call to subroutine 58710, P0% contains the logical record number of the detail record you added. If you wish to do further processing on the new detail record immediately after calling 58710, you will need to reload it into the buffer. This can be done with the following logic:

```
PF=P2         'SPECIFY DETAIL FILE AS DESIRED FILE
LR(P2)=PØ     'RELOAD RECORD NUMBER OF NEW DETAIL RECORD
GOSUB582ØØ    'GET AND FIELD IT
```

Linking a Detail Record Without Adding

Subroutine 58710 has a built-in call to subroutine 58430 of the labeled files handler. As you'll recall, subroutine 58430 updates the file statistics by adding to the count of active records and modifying the "next end of file" or "last deletion" pointer.

In some cases, you will want to avoid adding to the count of active records when linking a new detail record. In complex systems, for example, a detail record may be linked to more than one master record. In error recovery, the detail record may already exist, and you just want to relink it. In other cases, you may not be using the labeled files handler, so you want to avoid the call to 58430.

Let's assume that file 1 is your master file, file 2 is your detail file, and you have only one master-detail relationship in your program. You want to link logical record 33 of your detail file to logical record 100 of your master file. Here's the logic you can use:

```
PF=1:LR(1)=1ØØ:GOSUB582ØØ   'GET AND FIELD MASTER RECORD
PF=2:LR(2)=33:GOSUB582ØØ    'GET AND FIELD DETAIL RECORD
```

If you want to LSET new values into any data fields of the master or detail record, you can do it here. . .

```
PD=Ø:P1=1:P2=2              'SPECIFY CHAIN, MASTER FILE, DETAIL FILE
GOSUB 58712 :GOSUB 583ØØ    'LSET NEW POINTERS INTO DETAIL RECORD
PØ=LR(P2) : GOSUB 58714     'LINK PRIOR DETAIL RECORD TO NEW DETAIL
IF A$ = "E" THEN ....       'ABORT IF LINKING ERROR ENCOUNTERED
GOSUB 58719                 'UPDATE POINTERS IN MASTER RECORD
```

Now let's look at a system in which a detail record is to be linked to two different master files. Let's suppose that file 3 is the detail. A new record is to be added to the detail file, and it is to be linked to record 332 in file 1 and record 231 in file 2. Your commands are:

```
PF=1:LR(1)=332:GOSUB582ØØ   'GET FIRST MASTER RECORD
PF=2:LR(2)=231:GOSUB582ØØ   'GET THE OTHER MASTER RECORD
PF=3:GOSUB5842Ø             'GET NEXT AVAILABLE DETAIL RECORD

'Logic to LSET data into any of the fields can go here. . .

PD=Ø:P1=1:P2=3              'DEFINE FIRST RELATIONSHIP
GOSUB 5871Ø                 'ADD AND LINK THE DETAIL RECORD
IF A$ = "E" THEN ....       'ABORT IF LINKING ERROR ENCOUNTERED
PF=P2:LR(P2)=PØ:GOSUB582ØØ  'GET THE DETAIL RECORD AGAIN
PD=1:P1=2:P2=3              'DEFINE SECOND RELATIONSHIP
GOSUB 58712 : GOSUB583ØØ    'LSET POINTERS INTO DETAIL RECORD
PØ=LR(P2):GOSUB58714        'LINK PRIOR DETAIL TO NEW DETAIL
IF A$ = "E" THEN ....       'ABORT IF LINKING ERROR ENCOUNTERED
GOSUB 58719                 'UPDATE POINTERS IN MASTER RECORD
```

For faster operation in the above example, you can carefully re-arrange the logic so that disk file PUTs are minimized. When you are working with complex file relationships, it's best to set up a subroutine that does all this for you.

Reading a Chain

From any master record you can follow a chain to each of its detail records. Subroutine 58700 performs this function for you, and as it does so, it checks the validity of the chain.

To read a chain, you must define the master-detail relationship by loading PD%, P1% and P2%, if you haven't already done so. These variables are loaded the same way you'd load them if you were adding a detail record. Then you must get and field the master record, if it's not already in the buffer. LR%(P1%) must be the logical record number of the master record.

To start at the first detail record in the chain, you load LX%(PD%) with 0, and you call subroutine 58700. Upon return, if LX%(PD%) is 0, there were no detail records in the chain. Otherwise, the detail record will have been gotten and fielded, and you can perform your printing or processing logic for the detail record. To read the next record in the chain, you call subroutine 58700 again. (This time, upon calling 58700, LX%(PD%) is non-zero.) If upon return, LX%(PD%) is zero, you have reached the end of the chain. Otherwise, the detail record will have been gotten and fielded, and you can repeat.

As an example, let's suppose that file 3 is your master file, file 4 is your detail file, and you want to print the detail records that are chained to any record in the master file. Here's the logic you can use, assuming that PD% is 0.

```
5ØØØ PD=Ø:P1=3:P2=4                'DEFINE FILE RELATONSHIPS
5Ø1Ø INPUT"MASTER REC NUMBER";A%   'ALLOW OPERATOR ENTRY
5Ø2Ø PF=P1:LR(PF)=A%:GOSUB582ØØ    'GET AND FIELD MASTER RECORD
5Ø3Ø LX(PD)=Ø                      'START FROM FIRST DETAIL RECORD
5Ø4Ø GOSUB 587ØØ                   'GET NEXT RECORD IN CHAIN
5Ø41 IF A$="E"THEN 5Ø91            'ABORT IF LINKING ERROR
5Ø5Ø IF LX(PD) = Ø THEN 5Ø9Ø       'QUIT IF END OF CHAIN

5Ø6Ø 'Logic to print contents of detail record goes here...

5Ø7Ø GOTO 5Ø4Ø                     'REPEAT FOR NEXT DETAIL RECORD

5Ø9Ø GOTO 5Ø1Ø                     'LET OPERATOR ENTER ANOTHER
5Ø91 PRINT"LINKING ERROR":STOP     'HANDLE LINKING ERROR HERE
```

As you can see, subroutine 58700 will return A$ equal to "E" if there is an invalid link in the chain. It senses an error condition if the "master" pointer in the detail record accessed is not equal to the logical record number of the current master record. Upon return from subroutine 58700, P0% will always contain the logical record number of the prior detail record in the chain, if any. LX%(PD%) will contain the record number of the current detail record in the chain, if any.

In some systems, detail records may act as master records for detail records in another file. Because LX%(PD%) is a dimensioned variable, you can read detail files on multiple levels, or you can read any number of detail chains simultaneously.

Delinking a Detail Record

Subroutine 58740 is the delinking subroutine. It is provided so that you can remove a detail record from a chain. In effect, it "bridges" the previous detail record in the chain to the next detail record in the chain. Subroutine 58740 does not delete the detail record from the file or otherwise alter it. You may wish to delink a detail record from one master record's chain, and then link it onto another master record's chain. If you do want to delete the detail record, you should first delink it with subroutine 58740, and then delete it with subroutine 58440 of the labeled files handler. Alternatively, you can use your own deletion routine after you delink.

To delink a detail record, you must get and field the related master record, and you must get and field the detail record. (You can get the detail record by loading its record number and calling subroutine 58200, or you can use subroutine 58700 to read the chain until you get the detail record you want.)

Before calling subroutine 58740, P1%, P2% and PD% must define the chain you are using, just as they did when you added the detail record. LR%(P2%) must contain the logical record number of the record to be delinked.

Let's suppose that logical record 30 of the detail file is linked into the chain for logical record 100 of the master file. If file 1 is the master file, file 2 is the detail file, and the master-detail relationship was set up as 0, your logic to delink and delete the record is:

```
PF=1:LR(1)=1ØØ:GOSUB582ØØ   'GET THE MASTER RECORD
PF=2:LR(2)=3Ø :GOSUB582ØØ   'GET THE DETAIL RECORD
```

```
PD=Ø:P1=1:P2=2              'DEFINE FILE RELATONSHIPS
GOSUB 5874Ø                 'DELINK THE DETAIL RECORD
GOSUB 5844Ø                 'DELETE THE DETAIL RECORD
```

Deleting or Delinking a Chain

The easiest way to clear out a chain of detail records for any master record is to change the "first" and "last" pointers in the master record to zeros. To do this, you get and field the master record by calling subroutine 58200 with the file number in PF% and the logical record number in LR%(PF%). Assuming PD% is the number that specifies which master-detail relationship, you execute the following command:

```
LSET FF$(PD,3)=MKI$(Ø) : LSET FF$(PD,4)=MKI$(Ø) : GOSUB 583ØØ
```

The disadvantage of this method is that the detail records will remain in the detail file, taking up space. Here's a subroutine that you can use to delink and delete all the detail records in a chain for a master record:

```
6ØØØ INPUT"MASTER FILE NUMBER";P1
6Ø1Ø INPUT"DETAIL FILE NUMBER";P2
6Ø2Ø INPUT"MASTER-DETAIL CHAIN NUMBER";PD
6Ø3Ø INPUT"MASTER FILE LOGICAL RECORD NUMBER";LR(P1)
6Ø4Ø PF=P1:GOSUB582ØØ          'GET MASTER FILE RECORD
6Ø5Ø LX(PD)=Ø : GOSUB 587ØØ    'GET FIRST IN CHAIN
6Ø6Ø IF A$="E" THEN ....       'ABORT FOR LINKING ERROR
6Ø7Ø IF LX(PD)=Ø THEN RETURN   'RETURN IF DONE
6Ø8Ø LX(PD)=CVI(FF$(PD,2))     'SAVE ""NEXT" POINTER
6Ø9Ø GOSUB 5874Ø : GOSUB 5844Ø 'DE-LINK AND DELETE
61ØØ GOSUB 587Ø1 : GOTO 6Ø6Ø   'GET NEXT AND REPEAT
```

Adding and Deleting Master File Records

It is very important to LSET zeros into the "first" and "last" pointers of a new master record. This initializes the record with an empty chain. If for example, your master record contains the fields FF$(0,3) and FF$(0,4), your program logic should execute the command:

```
LSET FF$(Ø,3)=MKI$(Ø) : LSET FF$(Ø,4) = MKI$(Ø)
```

Before deleting a master record, your program should verify that the "first" pointer is zero. If it is non-zero, the master record has detail records linked to it, and they should be delinked and deleted first.

Getting a Detail Record's Master

If you have gotten and fielded a detail record, the logical record number of its master is stored in the FF$(PD%,0) field. Let's say, for example, that you are reading a detail file sequentially, and you want to print the contents of the detail file, along with the related data from the master file. If you are using the labeled files handler, the detail file number is 2, and the master file number is 1, you can use the following logic:

```
1ØØØ FOR X = 2 TO LØ(2,1)-1  'FROM FIRST TO LAST DETAIL RECORD
1Ø1Ø PF=2:LR(2)=X:GOSUB582ØØ 'GET THE DETAIL RECORD
1Ø2Ø IF ..... THEN 1Ø9Ø      'SKIP IF IT'S A DELETED RECORD
1Ø3Ø LR(1)=CVI(FF$(PD,Ø))    'LOAD CORRESPONDING MASTER RECORD
1Ø4Ø PF=1:GOSUB582ØØ         'GET THE MASTER RECORD
```

```
1Ø5Ø 'Logic to print data from master and detail record goes here...

1Ø9Ø NEXT                     'REPEAT FOR NEXT DETAIL RECORD IN FILE
```

The ability to read your detail file sequentially can be very important in some applications. In the accounts receivable system we've discussed, for example, the detail file provides a complete journal of the transactions for all accounts.

If you want the detail file to remain in sequence as you add new records, you should insure that each new record is added at the end of the file. You can do this by using a "GOSUB 58421" instead of a "GOSUB 58420" when you are adding detail records.

Clearing a Complete Detail File

In some applications, you may want to build a detail file during the month, and then clear it at the beginning of the next month. (Perhaps you might want to do it on a daily or yearly basis.) To do this, it's a simple matter of setting the count of detail file records back to zero (with a call to subroutine 58408 if you are using the labeled files handler) and changing the "first" and "last" pointers in each record of the master file back to zeros.

You can zero the "first" and "last" pointers in each master record with a FOR-NEXT loop that gets and fields each master record. On each master record, you use the following to clear the pointers:

```
LSET FF$(PD,3)=MKI$(Ø) : LSET FF$(PD,4)=MKI$(Ø) : GOSUB 583ØØ
```

Recovering from Linking Errors

Assuming that your program is fully tested and debugged, linking errors can still occur. Here are the possible causes and some solutions:

1. The detail file is more current than the master file, or the master file is more current than the detail file.

This is normally due to an operator error. The chances for this happening are much greater if the detail file and master file are on separate diskettes.

The "file synchronization" capabilities of the labeled file handler can help you prevent the problem. Your program should verify that the master file and detail file have the same date before processing begins.

2. Additions were made to the detail file, and it wasn't closed properly during the last run.

You can avoid this problem by pre-allocating the detail file and the master file when you originally set up the system. (See the section on pre-allocating files.)

If your program is using the labeled files handler, you can test for the problem just after opening a file by checking the "file validity" statistic. It will be 0 if the file was closed properly in the last run.

3. Erroneous data was put into a physical record by a hardware or DOS malfunction, or a flaw on the diskette.

The best solution in this case is to start over from a backup copy of your diskettes.

If you are writing a complex system, it is a good safety measure to write a recovery utility. The recovery utility clears out the "first" and "last" pointers in each master record. Then it reads the detail file sequentially, relinking each detail record to the corresponding master record.

Modifications to the DFH

There are many modifications that can be made to the detail file handler to add capabilities or to remove unneeded ones. We will discuss some of the most common modifications that are possible.

When Only One Chain is Required

The standard detail file handler is designed so that you can have any number of master-detail file relationships. This is done by double-dimensioning the FF$ array and loading PD% to specify which master-detail chain you want to use. I've found that most programs I write only have one master file which is related to only one detail file.

Some minor speed improvements and memory savings can be achieved by single-dimensioning the FF$ array. This can be done by removing the first dimension in each reference to FF$. Thus, "FF$(PD,0)" becomes "FF$(0)", "FF$(PD,1)" becomes "FF$(1)", "FF$(0,4)" becomes "FF$(4)", and so forth. In line 58000, we can change the dimension command so that the array is dimensioned "FF$(4)".

If your FF$ array is single dimensioned, you can remove all references to PD% since PD% was the variable we used to specify the dimension of FF$ to process. Accordingly, LX%(PD%) can be changed to the simple variable, LX%.

If you have a single master-detail relationship, you can preload P1% and P2% at the beginning of your program so that P1% specifies the master file and P2% specifies the detail file to be used. You don't need to re-load P1% or P2% before each call to the handler. (This applies whether you single-dimension the FF$ array or not.)

Reading a Detail Chain Backwards

For any master record, the standard version of subroutine 58700 follows the chain to each related detail record. It starts with the first detail record and ends with the last. In some applications, you may wish to read the chain by starting at the last detail record (the one that was most recently added.)

In an accounts payable system, for example, you might read a chain in a forward direction to see the history of your payments to a particular vendor. When you want to see your most recent payments first, you can read the chain in a reverse direction.

In an inventory system, you might wish to keep a detail chain of all purchases and the costs you incurred for each item. Depending on whether you read the chain forward or backwards, you can cost your inventory on a "FIFO" (first-in-first-out) or "LIFO" (last-in-first-out) basis.

If you always want to read in a reverse direction, you can change the "FF$(PD,3)" in line 58700 to "FF$(PD,4)" and the "FF$(PD,2)" to "FF$(PD,1)". The effect of this change is to make it appear that you are adding new detail at the beginning of the chain, rather than the end.

To allow for bidirectional chain reading capabilities, you can use a special variable when calling subroutine 58700. If you change the "FF$(PD,3)" in line 58700 to "FF$(PD,3+PX)" and the "FF$(PD,2)" to "FF$(PD,2-PX)", you can load PX with 0 to read forward or 1 to read backwards.

One-way Linking

The standard detail file handler subroutines use a bidirectional linking method. Each detail record in a chain is linked to the previous record as well as the next. This bidirectional linking makes it possible to delink any detail record without reading sequentially through the chain. It also makes it possible to read the chain backwards.

You can save 2 bytes in each detail record if you go to a one-way forward linking system. This can be done without modifying the detail file handler subroutines. You simply eliminate the "FF$(X,1)" field from your fielding command in those files where you want one-way linking. That way, you can have some detail files that are bidirectionally linked, and others that are not. If every detail file in your system is to use one-way linking, you can eliminate each reference to FF$(PD,1) in the detail file handler.

The standard delinking subroutine, 58740, cannot be used if you have one-way linking. Instead, you can use subroutine 58730, shown below:

```
Delinking a One-Way Linked Detail Record


58730 LX(PD)=CVI(FF$(PD,2)):IFPØ>ØTHEN58732ELSELSETFF$(PD,3)=FF$
(PD,2):IFLX(PD)=ØTHENLSETFF$(PD,4)=MKI$(Ø)
58731 PF=P1:GOSUB583ØØ:PF=P2:RETURN
58732 P3=LR(P2):LR(P2)=PØ:PF=P2:GOSUB582ØØ:LSETFF$(PD,2)=MKI$(LX
(PD)):GOSUB583ØØ:LR(P2)=P3:GOSUB582ØØ:IFLX(PD)=ØTHENLSETFF$(PD,4
)=MKI$(PØ):GOTO58731ELSERETURN
```

To use subroutine 58730, you must do a sequential read of the detail chain, so that P0% contains the logical record number of the previous record in the chain. (If you are delinking the the first record in the chain, P0% is 0.) The detail record and its master must both have been gotten and fielded. P1%, P2% and PD% should be pre-loaded, just as they are with any of the other detail file handler subroutines.

Here is a sample routine that can be used to delink and delete any or all detail records from a one-way chain. For the example, we will assume that the master file is file 1, the detail file is file 2, and the master-detail chain number, PD%, is 0. We will allow the operator to enter the desired master file record number and to individually select which detail records are to be delinked and deleted:

```
6ØØØ P1=1:P2=2:PD=Ø                'DEFINE MASTER-DETAIL RELATIONSHIP
6Ø1Ø INPUT"MASTER FILE REC";A%     'SELECT MASTER FILE RECORD
6Ø2Ø PF=P1:LR(P1)=A%:GOSUB582ØØ    'GET AND FIELD IT
6Ø3Ø LX(PD)=Ø                      'NEXT DETAIL RECORD IS FIRST
6Ø4Ø GOSUB 587ØØ                   'GET NEXT DETAIL RECORD
6Ø5Ø IF A$="E" THEN ....           'ABORT IF LINKING ERROR
6Ø6Ø IF LX(PD)=Ø THEN RETURN       'WE'RE DONE IF END OF CHAIN
6Ø7Ø 'Logic to print data from detail record for identification
      purposes goes here...
6Ø8Ø INPUT"DELETE THIS ONE? ";A$   'ALLOW OPTION TO DELETE
6Ø9Ø IF A$="N" THEN 6Ø4Ø           'GET NEXT IF ANSWER IS NO
61ØØ GOSUB 5873Ø                   'DELINK THE RECORD
611Ø GOSUB 5844Ø                   'DELETE THE RECORD
612Ø GOSUB 587Ø1 : GOTO 6Ø5Ø       'GET NEXT AND REPEAT
```

Deletion of Unneeded Routines

In many cases, computer applications are divided into several programs. In programs that simply display or print data from your files, you won't need the linking or delinking routines. You can delete lines 58710 through 58745 for these programs. In some programs, you may have no need to read detail chains. In these, you can delete lines 58700 through 58703.

In other programs, you won't need the delinking capabilities of the detail file handler. In accounting applications, for example, you will usually want to prevent the operator from deleting transactions, so that you'll have a complete audit trail. In these cases you can delete lines 58740 through 58745, or if you are using one-way linking, lines 58730 through 58731.

"DFH/DEM" — Detail File Demonstration

As you'll recall, LFH/DEM was a simple program that we used for testing and demonstrating the random disk file handler with labeled files. With a few modifications, we can "patch" in a detail file capability. The new program is called "DFH/DEM". You can use it to test and demonstrate the detail file handler, and to test any modifications you may wish to make to the handler.

For demonstration purposes, DFH/DEM has one detail file that is related to the name and phone number file, which serves as the master file. The detail file may contain short comments of up to 24 characters each, to keep a record of the phone calls that were made to each of people in the name and phone number file. When you recall any name, all the phone calls for that name and phone number are listed. When you want to reconcile your phone bill to the phone calls you made, you can list the entire detail file for a display of all calls that were made.

When you start the program, it will allow you to specify the master file name and initialize it, just as you did in the LFH/DEM program. You'll need to initialize the master file the first time you use the program because the DFH/DEM program uses a logical record length that is 4 bytes longer, to hold the "first" and "last" pointers.

The program will then let you specify the file name of the detail file, and you have the option of initializing it. You will need to initialize the detail file the first time you run the program. If you initialize the detail file, and the master file has active records in it, the program goes through each master file logical record, changing the "first" and "last" pointers back to zero.

Here's how the original startup dialogue looks on your screen, assuming you use a file name of "TESTFILE:1" for your master file and "DETAIL:1" for your detail file:

```
Random Disk File Handler -- Detail File Demonstration

FILE NAME:                TESTFILE:1
OPENING AS FILE 1...
INITIALIZE FILE?  (Y/N)     Y
INITIALIZED.
NEXT EOF = 2 ,  LAST DEL = Ø ,  COUNT = Ø
DETAIL FILE NAME:          DETAIL:1
OPENING AS FILE 3...
INITIALIZE FILE?  (Y/N)      Y
INITIALIZED.
```

```
DEMONSTRATION COMMANDS ARE:

<A> ADD A RECORD    <R> RECALL A RECORD     <C> CHANGE A RECORD
<D> DELETE A RECORD <L> LIST ACTIVE RECORDS <E> CLOSE AND END
```

To test the program, first you need to add a few names and phone numbers. You can do this by responding to the command request with <A>, just as you did in the LFH/DEM program. The add command operates exactly the same.

We'll assume for now that you've added the same 4 names and phone numbers used in the discussion of the LFH/DEM program. If you respond by typing <L> for the command, the program now asks you whether you want to list the master file or the detail file. Since the detail file is empty at this point, you can enter <M> to list the master. Here's how it will look:

```
COMMAND? (A,R,C,D,L, OR E)  L

ENTER <D> TO LIST DETAIL FILE, <M> TO LIST MASTER... M



RECORD 2                   JONES, JOHN
                           333 333-3333
RECORD 3                   ROBERTS, BILL
                           333 332-3214
RECORD 4                   JOHNSON, SAM
                           432 331-3322
RECORD 5                   ADAMS, SUE
                           931 987-6543
```

Now let's suppose you make a phone call to Sam Johnson and you want to record it. You simply enter <R> in response to the command request. For the record number, you enter <4>, and the following is displayed:

```
COMMAND? (A,R,C,D,L, OR E)  R
RECORD NUMBER?              4
NAME:                       JOHNSON, SAM
PHONE NUMBER:               432 331-3322



COMMENT:                     . . .
```

Below the name and phone number, the computer lets you type in a comment of up to 24 letters, to record the details of the phone call you are making. If you don't want to type a comment, you can just press <ENTER> to go back to enter another command.

Let's suppose you want to record the fact that you made a phone call on January 4th, for 5 minutes. When you type the comment and press <ENTER>, the program adds it and links it into the detail file, and requests another comment. If you don't have another to comment to record, you can simply press <ENTER>, but let's record another call on the same date, for 3 minutes. Here's the display:

```
COMMAND? (A,R,C,D,L, OR E)  R
RECORD NUMBER?              4
NAME:                       JOHNSON, SAM
PHONE NUMBER:               432 331-3322



COMMENT:                    JAN 4, 5 MINUTES
COMMENT:                    JAN 4, 3 MINUTES
COMMENT:                     . . .
```

As you can see, you can enter as many comments as you wish, just as if you were writing information onto an index card or ledger card. Whenever you recall record 4 or any other record number from the master file, all the comments that you've typed for that name are listed, and you have the opportunity to type one or more additional comments.

Record 4, after we've "posted" to it several times, might show the following information when you issue a <R> command:

```
COMMAND? (A,R,C,D,L, OR E)  R
RECORD NUMBER?              4
NAME:                       JOHNSON, SAM
PHONE NUMBER:               432 331-3322



                            JAN 4, 5 MINUTES
                            JAN 4, 3 MINUTES
                            JAN 8, 6 MINUTES
                            JAN 9, 4 MINUTES
COMMENT:                     . . .
```

Assuming you've made some entries to other accounts in the master file, you can use the <L> command to list the complete detail file. You get a complete "history" of all phone calls that were made:

```
 COMMAND? (A,R,C,D,L, or E)  L

ENTER <D> TO LIST DETAIL FILE, <M> TO LIST MASTER... D



 JOHNSON, SAM          432 331-3322: JAN 4, 5 MINUTES
 JOHNSON, SAM          432 331-3322: JAN 4, 3 MINUTES
 JONES, JOHN           333 333-3333: JAN 4, 1Ø MINUTES
 ROBERTS, BILL         333 332-3214: JAN 5, 7 MINUTES
 ADAMS, SUE            931 987-6543: JAN 6, 7 MINUTES
 JOHNSON, SAM          432 331-3322: JAN 8, 6 MINUTES
 JONES, JOHN           333 333-3333: JAN 8, 6 MINUTES
 JOHNSON, SAM          432 331-3322: JAN 9, 4 MINUTES
```

DFH/DEM also demonstrates some of the the delinking and deletion capabilities of the detail file handler. When you answer "D" to the command request, the detail that is related to the master file record number you specify will be deleted. Then you have the option to either delete or retain the master file record. Here's the dialogue as it appears on the screen if we delete all the detail records related to record 4:

```
COMMAND? (A,R,C,D,L, OR E)  D
RECORD NUMBER?              4
NAME:                       JOHNSON, SAM
PHONE NUMBER:               432 331-332
DELETING DETAIL...
DELETE NAME TOO? (Y/N)      N
```

How to Set Up the DFH/DEM Program

To create the DFH/DEM program, you should first write and test the LFH/DEM program as it is shown and explained. Next, you should merge in the detail file handler subroutines, exactly as they are shown. They will occupy line numbers 58700 through 58745. Finally, you should add or replace the program lines shown below.

When you start up from DOS READY, make sure that you specify at least 3 files. In the DFH/DEM program, we are using file 3 for the detail file, and file 1 for the master file. File 2 remains unused, for compatibility with some of the other demonstration programs that we'll be discussing.

Figure 8.1 *DFH/DEM — Modifications to LFH/DEM*

```
Ø 'DFH/DEM

1ØØØ CLS:PRINT"RANDOM DISK FILE HANDLER -- DETAIL FILE DEMONSTRA
TION":PRINTSTRING$(63,"=")

1Ø65 GOSUB4ØØØ

2221 LSETFF$(Ø,3)=MKI$(Ø):LSETFF$(Ø,4)=MKI$(Ø)

234Ø IFCD$="C"THEN235ØELSEIFCD$="D"THEN236ØELSEGOTO41ØØ

236Ø GOSUB43ØØ:LINEINPUT"DELETE NAME TOO? (Y/N)       ";A$:IFA$<>
"Y"THEN2ØØØELSEPRINT"DELETING...":PF=1:GOSUB582ØØ:GOSUB5844Ø

24Ø1 PRINT:LINEINPUT"ENTER <D> TO LIST DETAIL FILE, <M> TO LIST
MASTER... ";A$
24Ø2 PRINTSTRING$(63,"-"):IFA$="D"THEN42ØØ
25ØØ KV%=Ø:PF=1:GOSUB584ØØ:GOSUB5829Ø
25Ø1 PF=3:GOSUB584ØØ:GOSUB5829Ø
251Ø CLS:END
```

```
4ØØØ LINEINPUT"DETAIL FILE NAME:          ";FS$
4Ø1Ø PRINT"OPENING AS FILE 3...":PF=3:GOSUB5825Ø:GOSUB584Ø5
4Ø2Ø LINEINPUT"INITIALIZE FILE?  (Y/N)     ";A$
4Ø21 IFA$<>"Y"THENRETURN
4Ø3Ø PF=3:KV%=Ø:GOSUB584Ø8:PRINT"INITIALIZED.":KV%=1
4Ø4Ø IFLØ(1,3)=ØTHENRETURNELSEPF=1:PRINT"CLEARING POINTERS..."
4Ø5Ø FORX=2TOLØ(1,1)-1
4Ø55 LR(1)=X:GOSUB582ØØ:IFMID$(FH(1),3,1)=CHR$(255)THEN4Ø7Ø
4Ø6Ø LSETFF$(Ø,3)=MKI$(Ø):LSETFF$(Ø,4)=MKI$(Ø):GOSUB583ØØ
4Ø7Ø NEXT:RETURN

41ØØ PD=Ø:P1=1:P2=3:LX(PD)=Ø
41Ø1 PRINTSTRING$(63,"-")
411Ø GOSUB587ØØ:IFA$="E"THENPRINT"LINKING ERROR":STOP
4111 IFLX(PD)=ØTHEN413Ø
412Ø PRINTTAB(28)FH(3):GOTO411Ø
413Ø PRINT"COMMENT:";TAB(28):PF=3:GOSUB5842Ø
414Ø LINEINPUTA$:IFA$=""THEN2ØØØELSELSETFH(3)=A$
415Ø GOSUB5871Ø:IFA$="E"THENPRINT"LINKING ERROR":STOP
416Ø GOTO413Ø

42ØØ IFLØ(3,3)=ØTHEN2ØØØ
421Ø FORX=2TOLØ(3,1)-1
422Ø PF=3:LR(3)=X:GOSUB582ØØ:IFMID$(FH(3),3,1)=CHR$(255)THEN424Ø
4221 PF=1:LR(1)=CVI(FF$(Ø,Ø)):GOSUB582ØØ
423Ø PRINTFH(1);" ";FH(2);": ";FH(3)
424Ø NEXT:GOTO2ØØØ
43ØØ PRINT"DELETING DETAIL..."
431Ø P1=1:P2=3:PD=Ø:LX(PD)=Ø:GOSUB587ØØ
432Ø IFA$="E"THENPRINT"LINKING ERROR":STOP
433Ø IFLX(PD)=ØTHENRETURNELSELX(PD)=CVI(FF(PD,2))
434Ø GOSUB5874Ø:GOSUB5844Ø
435Ø GOSUB587Ø1:GOTO433Ø

58ØØØ A%=3:DIMPR(A%),PP(A%),LR(A%),LL(A%),FD$(A%),LØ(A%,5),FF$(Ø
,4),LX(Ø)
58ØØ1 LL(1)=4Ø:LL(2)=6:LL(3)=3Ø

58Ø1Ø FIELDPF,LS*LL(PF)ASFD$(PF),24ASFH(1),12ASFH(2),2ASFF$(Ø,3)
,2ASFF$(Ø,4)

58Ø3Ø FIELDPF,LS*LL(PF)ASFD$(PF),24ASFH(3),2ASFF$(Ø,Ø),2ASFF$(Ø,
1),2ASFF$(Ø,2)
```

DFH/DEM Comments

1. To simplify the job of typing in the DFH/DEM modifications, note that some of the lines represent only minor changes to lines already in the LFH/DEM program. Rather than retyping them, you can go into edit mode to make the changes

required. The lines that fall into this category are 1000, 2340, 58000, 58001, 58010, and 58030.

2. Notice that lines 4000 through 4070 perform the task of opening the detail file as file 3. If you initialize the detail file when the master file has active records in it, lines 4040 through 4070 go through the master file to zero out the "first" and "last" pointers.

3. Lines 4100 through 4160 are patched onto the "recall" routine. After the name and phone number are displayed, lines 4100 through 4120 provide the logic that follows the chain to display each related detail record. Then, in lines 4130 through 4160, the operator is given the opportunity to enter additional comments to be chained to the current master record.

4. Lines 4200 through 4240 process the "list" command for the detail file. As you can see, a FOR-NEXT loop is used. When CHR$(255) is found in the third byte of a detail record, you know that it has been deleted, so the printing logic is skipped.

5. The deletion of all detail related to a master record is performed in lines 4300 through 4350.

6. Line 58000 is modified to include the dimensioning for the FF$ and the LX% arrays. Line 58001 contains the new logical record lengths. File 1 now has a logical record length of 40 bytes, while file 3 has a record length of 30 bytes.

7. Line 58010 handles the new fielding for file 1. Since file 1 is the master file, you simply added the FF$(0,3) and FF$(0,4) fields, for a total of 4 additional bytes per logical record.

8. Line 58030 handles the fielding for file 3, our detail file. Notice that FH$(3) is the comment field, having a length of 24 bytes. The 6 bytes required by FF$(0,0), FF$(0,1) and FF$(0,2) handle the pointers that link the detail records.

9. Notice that line 2360 handles the deletions. First you do a GOSUB 4300 to to delete the detail. When the operator elects to delete the master record, the "GOSUB 58200" in line 2360 is very important. Even though the proper master record is already fielded and in the buffer, we need to call 58200 to recompute LS% for the current logical record of file 1 before using the deletion routine at 58440.

Random Byte Addressing

Every file, under any of the popular disk operating systems for the TRS-80, can be considered to be a series of 256-byte physical records. This includes random files, as well as sequential files. The first byte of a file is in the first physical record. The 257th byte is the first byte of the second physical record.

The subroutines we will discuss in this section let you access a random file by specifying a desired byte position within the file. Subroutine 58800 is the "string-get" subroutine. It reads up to 255 bytes from any position in the file. Subroutine 58810 is the "string-put" subroutine. It records a string, of any length up to 255 bytes, to any byte position within the file. Both subroutines, when necessary, handle the possibility that the string you are reading or writing may span two physical records.

As an example, let's suppose you want to record the string, "This is a test!", beginning at the 505th byte of a file. Once you have opened the file and specified the file number as PF%, the command is:

```
FV$="This is a test!" : PB!=505 : GOSUB58810
```

Subroutine 58810 computes that the first 7 bytes of the string will occupy byte positions 250 through 256 in the second physical record. It automatically gets physical record 2, if it is not already in the buffer, it does the required fielding, and does an LSET of "This is", beginning at byte 250. Then it puts the second physical record and gets the third. The remaining 8 bytes, " a test!" are LSET into the first 8 bytes of the physical record 3. Finally, subroutine 58810 does the put for the third physical record.

When you want to retrieve the string, you can use the following commands:

```
PB!= 505 : BC%= 15 : GOSUB 58800 : PRINT FV$
```

Upon return from subroutine 58800, FV$ contains the 15 bytes that begin at the 505th byte of the file, so "This is a test!" is printed.

As you can see, the byte position is expressed as a single precision variable, PB!. This lets you specify any position from 1 to 999999, limited only to the capacity of your diskette.

Subroutines 58800 and 58810 use FV$ as the string variable for PUTting and GETting strings. When you call subroutine 58810 to PUT a string to any byte position in a file, the number of bytes recorded is determined by the length of FV$.

No other bytes in the file are disturbed. When you call subroutine 58800 to GET a string from any byte position in a file, the length of the string gotten is specified by BC%. BC% may be from 1 to 255 bytes.

Before we go into more details, let's look at the advantages and applications of the string-get and string-put subroutines.

1. You can design random files to have any logical record length from 1 to 999999 bytes. Your logical records don't need to fit neatly into blocks of 256 bytes. By computing the byte position at which any logical record will start, and adding the offset to the desired field within the record, your program can GET or PUT any field in the file.

2. You can have variable length logical records, or variable length fields within logical records. You can access them by separately maintaining an index of the beginning positions.

3. You can design your own "quasi-sequential" files. You can, for example, design schemes that let you have all the advantages of a sequential file, with several additional capabilities, such as:

   a. The ability to store compressed data in the file. (In a normal sequential file, all data must be uncompressed "ASCII" data.)

   b. The ability to read forward or backward, and the ability to start from any position.

   c. The ability to "re-write" at any position.

   d. The ability to prevent lost data in the event of a power failure, by being able to do "hard-puts". (With a normal sequential file in output mode, you lose everything if the file is not closed.)

4. Your BASIC program can read or write files that are compatible with any word processing or editor/assembler program you may be using. You can also read and modify BASIC program files, without the need to save them in ASCII format.

5. You can "patch" changes into machine language files. If, for example, you want to change the 310th through 315th bytes of a machine language program that is stored on disk, you can do it without re-assembling.

6. You can open sequential files in random mode. Then you have random access to any position within the file.

The "string-get" and "string-put" subroutines operate in conjunction with the random disk file handler. The required random disk file handler subroutines are 58210 and 58300. PF% is the file number; subroutine 58210 gets the physical record specified by PR%(PF%) if it is not already in the buffer. Subroutine 58300 puts the physical record specified by PR%(PF%).

Subroutines 58800 and 58810 both rely on subroutine 58850. For the requested byte position, subroutine 58850 computes the physical record number and the offset within that physical record. Then it gets and fields the record. Subroutines 58800 and 58810 need not both be present in your program, but 58850 is required to use either.

String-Get:

```
588ØØ GOSUB5885Ø:IFLEN(FD$)>=BC%THENFV$=LEFT$(FD$,BC%):RETURNELS
EFV$=FD$:PR(PF)=PR(PF)+1:GOSUB5821Ø:FIELDPF,BC%-LEN(FV$)ASFD$:FV
$=FV$+FD$:RETURN
```

String-Put:

```
5881Ø GOSUB5885Ø:IF256-LS>=LEN(FV$)THENPOKEVARPTR(FD$),LEN(FV$):
LSETFD$=FV$:GOSUB583ØØ:RETURN
58811 LSETFD$=FV$:GOSUB583ØØ:PR(PF)=PR(PF)+1:GOSUB5821Ø:FIELDPF,
LEN(FV$)-LEN(FD$)ASFD$:LSETFD$=MID$(FV$,LEN(FV$)-LEN(FD$)+1):GOS
UB583ØØ:RETURN
```

Compute, GET, and Field:

```
5885Ø PR(PF)=INT((PB!-1)/256)+1:LS=PB!-(PR(PF)-1)*256-1:GOSUB582
1Ø:FIELDPF,(LS)ASA$,ØASFD$:IFLS>ØTHENPOKEVARPTR(FD$),256-LS:RETU
RNELSEPOKEVARPTR(FD$),255:RETURN
```

Variables Used

**A$** Used as a temporary dummy variable when fielding, just to insure that FD$ starts at the proper position. The contents of A$ are not useful to your main program upon return.

**FD$** Used as a temporary string for fielding the disk buffer. The contents of FD$ are not useful to your main program upon return.

**FV$** Provided by your program to contain the string to be written to disk before a call to 58810. (It is left unchanged upon return from 58810.) FV$ contains the string gotten from disk upon return from subroutine 58800.

PB! Provided by your program to specify the starting position within the disk file to PUT or GET the string, FV$.

BC% You load BC% with the length of the string you wish to GET, when calling subroutine 58800. FV$ will be returned with a length equal to BC%. This variable is not used in the string-put subroutine.

LS% Computed as the number of bytes within the physical record, preceding the desired data.

PF% Your mainline program specifies PF% so that the random disk file handler will know which file number to use.

PR%(PF%) Computed by subroutine 58850 as the physical record number within the file, that contains relative byte, PB!. PR%(PF%) is incremented in lines 58800 and 58811 if the string spans two physical records.

PP%(PF%) Used by the random disk file handler to determine whether or not a GET is necessary.

Line Comments

58800 (String-Get Subroutine)

:Call subroutine 58850 to GET and field the proper physical record.
:If the length of the field, FD$, is greater than or equal to the number of bytes you want, BC%,
:then we can load FV$ with the left BC% bytes of the field,
:and return to the main program.
:Otherwise, load FV$ with the entire field,
:and add 1 to the physical record number,
:and call subroutine 58210 to bring the remaining bytes we want into the disk

buffer.
:Field the buffer so that FD$ will contain the first portion of it, for a length equal to the number of bytes you want, minus the number of bytes you've already have.
:Now add FD$ onto what you've already gotten, to get what we want in FV$.
:And we can return to the mainline program.

58810 (String Put Subroutine)

:Call subroutine 58850 to GET and field the proper physical record.
:If the number of bytes in the buffer, minus the number of bytes preceding the field, is greater than or equal to the length of the string you want to PUT, then you can handle the whole string in this physical record,
:so POKE the VARPTR of FD$ to reduce the length of the field down to the length of the string you want to PUT,
:and you load the contents of the string into the field.
:and you call subroutine 58300 to PUT the physical record,
:and return to the main program.

58811

:Otherwise, the entire string won't fit into the first physical record, so LSET the first portion of it,
:and call subroutine 58300 to PUT the physical record.
:Now add 1 to the physical record number,
:and call subroutine 58210 to GET the next physical record.
:make the length of the field, FD$, equal to the total length of what you want to record, minus the length of what was already recorded in the previous physical record.
:Now LSET the remaining portion of FV$ into the field,
:and call subroutine 58300 to put the physical record,
:and, finally, return to the main program.

58850 (Supporting Subroutine — GET and Field the Proper Physical Record.)

:Load PR%(PF%) with the proper physical record number. It is computed as the requested byte position minus 1, divided by bytes per physical record, plus 1.
:Load LS% with the number of bytes that will precede the requested byte position within the physical record.
:Call subroutine 58210 to get the physical record if it is not already in the disk buffer.
:Field the file buffer, using A$ as a dummy field, so the "real" field you're interested in, FD$, will start at the correct position. For now, field FD$ with a length of 0. (The parentheses around "LS" avoid confusing Disk BASIC — you get an illegal function call without them.)
:If the FD$ is to start anywhere other than the first byte of the buffer, then POKE its length so it spans the remainder of the buffer,
:and return, to either 58800 or 58810.
:Otherwise, it starts at the first byte of the buffer. Since you can't have a string length of 256, POKE the VARPTR of FD$ so it has a length of 255,
:and return, to either 58800 or 58810.

Using String-Get and String-Put

Here's a short program you can use to test the string-get and string-put subroutines. It asks you to specify and open a random disk file. Then you can enter a byte-position within the file, and you can elect to GET a string by entering <G>, or to PUT a string by entering <P>.

When you wish to GET a string, you specify the number of bytes. If you are PUTting a string, you simply type it. When you are finished, enter 0 for the byte position to close the file and end.

To use "RBA/DEM", you will need to first load the standard disk file handler subroutines and the string-get and string-put subroutines shown. Then type in the following few lines:

Figure 9.1 — *RBA/DEM*

```
Ø 'RBA/DEM
1 CLEAR1ØØØ:DEFINTA-Z:DIMPR(1),PP(1)
6Ø LINEINPUT"FILE NAME: ";FS$:PF=1:GOSUB5825Ø
1ØØ INPUT"DESIRED BYTE? (Ø TO CLOSE)  ";PB!:IFPB!=ØTHENEND
12Ø LINEINPUT"G=GET P=PUT ?";CD$:IFCD$="P"THEN2ØØØELSE1ØØØ
1ØØØ INPUT"GET HOW MANY BYTES";BC%
1Ø1Ø GOSUB588ØØ:PRINTFV$:GOTO1ØØ
2ØØØ LINEINPUT"ENTER THE STRING: ";FV$
2Ø1Ø GOSUB5881Ø:GOTO1ØØ
```

Modifying for "Soft-Puts"

The string-put subroutine calls subroutine 58300 from line 58810 and 58811. Subroutine 58300 tells the computer to write the contents of the disk buffer onto the diskette. In some applications, you may be writing a large number of short strings into the same physical record, and you won't want the computer to do a "hard-put" except when necessary.

You may remove both calls to subroutine 58300 if you specify that the files to be handled are to use the "soft-put" capability. With a slight modification, you can make the call to subroutine 58300 conditional on whether the "soft-put" capability is activated for the file being used. You activate the "soft-put" capability by setting the bit in PS% that that corresponds to the file number minus 1. (See "Using the Soft-Put Capability" in Chapter Six.)

Reading and Modifying a BASIC Program File

Let's say you want to modify a BASIC program on disk so all PRINT commands from line 10000 through 10999 are changed to LPRINT commands. Assuming the program was saved on disk as "PROGRAM", here are the commands you could use:

```
1Ø FS$="PROGRAM"            'SPECIFY FILE NAME
2Ø PF=1:GOSUB5825Ø          'SPECIFY FILE NUMBER AND OPEN IT
3Ø PB!=2                    'START AT SECOND BYTE OF THE FILE
4Ø BC%=2:GOSUB 588ØØ        'GET 2-BYTE POINTER TO NEXT LINE
5Ø IF CVI(FV$)=2 THEN END   'END IF NO NEXT LINE
```

```
6Ø PB!=PB!+2 :GOSUB 588ØØ     'GET NEXT 2 BYTES, (LINE NUMBER)
7Ø LN!=CVI(FV$)               'GET LINE NUMBER
8Ø IF LN!<Ø
 THEN LN!=LN!+65536          'CONVERT FOR LINES ABOVE 32767
9Ø IF LN!>1Ø999 THEN END      'END IF LINE NUMBER HIGHER THAN DESIRED
1ØØ PRINT LN!                 'DISPLAY CURRENT LINE NUMBER
11Ø PB!=PB!+2                 'ADVANCE BYTE POSITION
12Ø BC%=255 :GOSUB 588ØØ      'GET NEXT 255 BYTES
13Ø A%=INSTR(FV$,CHR$(Ø))     'CHR$(Ø) DENOTES END OF THE LINE
14Ø IF  LN!<1ØØØØ
  THEN PB!=PB!+A%+1:
  GOTO 4Ø               'GET NEXT LINE IF LESS THAN DESIRED
15Ø FV$=LEFT$(FV$,A%-1)       'CHOP OFF ALL BUT TEXT FOR CURRENT LINE
16Ø X%=INSTR(FV$,CHR$(178))   'LOOK FOR "PRINT CODE"
17Ø IF X%=Ø
   THEN PB!=PB!+A%+1:
  GOTO 4Ø               'IF NONE FOUND, GO GET NEXT LINE
18Ø MID$(FV$,X%,1)=CHR$(175) 'CHANGE IT TO AN "LPRINT" CODE
19Ø GOSUB 5881Ø               'WRITE THE MODIFIED LINE TO DISK
2ØØ GOTO 16Ø                  'REPEAT
```

The program "MERGEPRO/BAS", in *BASIC Faster and Better and Other Mysteries*, is another example of how the string-get and string-put subroutines are used. That program automatically extracts selected lines or ranges of lines from BASIC programs, renumbers them, and merges them together to build new programs.

Reading and Modifying Other Types of Files

It's a simple matter to read, create, or modify any other type of file from your BASIC program, once you know how the file is organized on disk. *TRS-80 Disk and Other Mysteries* by H.C. Pennington tells you everything you need to know. The discussion on BASIC program files in that book has been especially helpful to me.

Random Access to Variable Length Text

Let's suppose you want to design your program so that you can instantly access text material from a word processing file, or any other file that has been recorded on disk in sequential mode. For example, you might want to access any one of many instruction paragraphs to display as a "help" screen, or you might want to access definitions, quotations or standard paragraphs to be inserted in letters or contracts.

When creating the text, you can insert special codes to separate the paragraphs or sections. Then, with the "string-get" subroutine, you can find the byte position for each paragraph, or section of text, to create an index of byte positions. Then, in your main program, when you want to print or process from a desired section of text, you can call subroutine 58800 based on the byte position you want.

In most cases, you will want to set BC% equal to 255 when calling 58800 to read text. Upon return from subroutine 58800, you can test the string to see if there are any delimiters in it. A delimiter is a one-byte code, such as a CHR$(13), that separates one line of text from the next. Then you can print the left portion of FV$, up to the delimiter, advance PB! by the number of bytes you printed plus 1, and repeat for the next line of text. Here, for example, is how you might access and print a block of text from byte position 1332 of a normal sequential file. Let's assume that you have created the file so that "END", when located in the first 3 bytes of a line, denotes the end of a block of text:

```
1ØØ PB!=1332 :PF=1          'SPECIFY STARTING POSITION AND FILE
11Ø BC%=255                 'GET 255 BYTES WITH EACH DISK ACCESS
12Ø GOSUB 588ØØ             'GET DATA FROM DISK
13Ø IF LEFT$(FV$,3)="END"
   THEN RETURN              'END THE ROUTINE IF "END" IS FOUND
14Ø A%=INSTR(FV$,CHR$(13))  'OTHERWISE, LOOK FOR A DELIMITER
15Ø IF A%=Ø
   THEN PRINT FV$;:
   PB!=PB!+ 255 : GOTO 12Ø  'IF NO DELIMITER, PRINT & GET NEXT.
16Ø PRINT LEFT$(FV$,A%-1)   'OTHERWISE, PRINT TO DELIMITER
17Ø PB!=PB!+A%+1:GOTO12Ø    'INCREMENT PAST DELIMITER AND REPEAT
```

"Quasi-Sequential" Files

You can design your own file organizations according to your special requirements.

In one of my customer's installations, I had data that normally would have been stored as a sequential file, but I wanted to compress it as much as possible to minimize the phone bill when sending it from one computer to another over the phone lines. The data included a price, quantity and description for each sale made during the day. The price and quantity could each be stored in 4 bytes, but the description was a variable length string. For each sale, I used the string-put subroutine to record the 4-byte price and the 4-byte quantity in MKS$ format. Then I recorded a 1-byte character to specify the length of the description, followed by the variable-length description itself. The result of this organization was at least a 50% time savings over the alternative of using a regular sequential file.

"Catch-All" Files

This idea is useful when you have limited memory, or a large number of items of various lengths that your program needs to keep track of. It's especially good when the information to be stored doesn't fit into the repetitive pattern of logical records that is normally associated with files. Rather, a catch-all file can be considered a file with one logical record having an unlimited number of fields. You PUT and GET data to and from each "field" by byte position. The items can be numbers or anything else, as long as they are converted to strings.

To design such a file, before writing your program (or as you write it), you just make a list of all the data items your program will require. Next to each, write a description of the item, how it is stored (String, MKI$, MKD$, etc.) and the length of the item. In a final column, write down the byte position where that item will be stored. You can start with byte 1 for the first item. The byte position for each other item will be the cumulative lengths of all that precede, plus 1.

Let's say, for example, that you want to accumulate hundreds of different, miscellaneous facts about the status of your company. These facts might come from dozens of different programs, but you want them all at your fingertips in one program, and one file. Your catch-all file might contain such diverse things as:

Current checkbook balance
Current number of employees
Federal employer identification number
Current accounts receivable balance
Accounts receivable balance, beginning of year
A calendar of working days this year

Key employee names and positions
A list of each of the program names in your system
A list of all the files in your system and their fields
Etc.

At the beginning of the catch-all file, you might even want to reserve, say, 1000 bytes of space for a list of all the items of information that are in the file, and their byte positions. This makes it so that your catch-all file has a "table of contents."

The Memory Index Handler

With each of the random file accessing methods we've discussed in the previous chapters, you had to know the physical or logical record number to recall information from the file. To call up "SUE ADAMS" in the LFH/DEM program, for example, you had to know that the information was stored in logical record 5. In many applications, retrieval by record number is fine. If you are writing a payroll program, you can number your employees 1, 2, 3, 4 and so on. If you are writing an accounts payable program and you have 100 vendors to whom you normally pay your bills, you can number them 1 through 100. To have the record numbers handy, you can set up a Rolodex file to index them. If you are writing a check register program, a routine can subtract 1000, 2000, 3000, and so forth, from the check number you enter, to convert to a logical record number. Check number 2015, for example, could be stored as logical record 15.

However, in many systems it's necessary (or at least more convenient) to recall data by an alphanumeric item "number." You might, for example, have products in an inventory file that have numbers like "LX-302", "23031A" and "AB403". In an accounts receivable or accounts payable system, you might want to use easy-to-remember codes such as "ABC" for "ABC VENDING" or "JD" for "JONES DISTRIBUTING." In a general ledger system, your accountant may have already assigned account numbers for you, such as "301.0" for "SALES", "423.0" for "MAINTENANCE EXPENSE" and "110.1" for "CASH IN BANK — CHECKING ACCOUNT".

Account numbers, item numbers and product numbers such as these are called "keys." The task of the program is to convert a key, as entered by the operator, to a logical record number, so the proper disk record can be accessed. The program must be able to search for the record by key very quickly, within a second or two, to either get the record — if the key is valid, or to display a message that the key was not found.

There are many complex schemes for handling search keys. The following chapters will present some of them. In this chapter we will discuss one relatively simple high-speed method for handling keyed access to random disk files.

The memory index handler is a set of BASIC subroutines that work in conjunction with the random disk file handler. The memory index handler accesses disk file logical records by short alphanumeric keys. Let's say, for example, you've got an inventory system and want to access product number "X403A" from a disk file. With these subroutines, your program can look it up in under a second if it's found within the first 500 logical records of your disk file, and under 2 seconds if it's within the first 1000 records. If it's not found, the memory index handler subroutines can tell you it's an invalid number.

I've found that the memory index handler is most practical in applications where the keys are from 2 to 10 characters long, and the file is under 1000 logical records. This is because, after you open your files, all the keys are loaded into an array in memory so that they can be searched quickly. The limiting factors are the size of your main program and the amount of memory you have.

Here are the main ideas involved in the operation of the memory index handler:

1. All the keys required to access one random file are stored in a separate random file. For example, if you have an inventory file that contains product descriptions, prices and quantities, you have a separate index file that contains the stock numbers only. The "nth" stock number in the index file corresponds to the "nth" product in the inventory file. Because of this, the keys are not necessarily in a sorted sequence.

2. After the files have been opened, a subroutine in the memory index handler loads the entire contents of the index file into a memory array. This memory array consists of strings 255 bytes long. Each of the strings contains as many of the keys as can fit into 255 bytes. For example, if you are using a 6-character key to index an accounts receivable file, 42 of these keys can be stored in a 255-byte string. To index 420 customer records, the memory array would have 10 elements.

   The keys are packed into 255-byte strings to make the search faster and to conserve memory. Compared to the alternative of using separate array elements for each key, this method of storage saves 123 bytes for every 42 keys — a savings of 2640 bytes in a file of 1000 6-byte keys!

3. When your program requires a search by key, a few variables are loaded, and the required subroutine is called. The search subroutine in the memory index handler uses the INSTR (instring) function (one of BASIC's fastest features) for the search. If an exact match for the key is found, the logical record number is returned, and your program may access it from disk. If no match is found, 0 is returned.

4. To add or change a key, your BASIC program specifies the logical record number and the file number being used to hold the index. The subroutine then changes the key within the memory array, and it writes the array string that has been modified into the index file on disk.

5. The subroutines in the memory index handler are designed so that you can have any number of memory indices, limited, of course, to memory capacity. The array that holds the keys is double dimensioned for this purpose. I've done programs with as many as 5 different indices in memory. An invoicing program, for example, might have two: one for the customer file, and one for the product file.

Implementing the MIH

The memory index handler subroutines occupy lines 58500 through 58552. As you will see, some of them are optional, depending on the functions your program is to perform. Subroutines from the random disk file handler are called by the memory index handler. You may index labeled or unlabeled files, but the index file itself is never a labeled file. With a few modifications, you can hook the memory index handler into any random disk file program you may have, even if the program doesn't have the standard random disk file handler subroutines.

Supporting Subroutines Required

The memory index handler subroutines do their own computations and disk file fielding, so some of the routines from the random disk file handler are not required unless you are using them for other files. Other random disk file handler subroutines are called, mainly so that disk errors will be properly processed. Here are the subroutines called by the memory index handler:

58210 and 58220 Physical Record Soft-Get and Hard-Get

58300 Physical Record Hard-Put

58250 and 58290 Random File Open and Close

In addition, you'll need random disk file handler lines 58900 through 58996, since these are used when disk errors are encountered.

Variable Type Considerations

The memory index handler subroutines assume that variables starting with "L" and "P" are predefined as integers, (with the DEFINT statement). All other variables within the handler subroutines are explicitly defined as integer, single precision or string. Variables beginning with "F" are always explicitly defined in the handler subroutines as strings, so you can remove the "$" designators if you do a "DEFSTR F" early in your program.

Dimensioning Considerations

As we discussed, the keys are stored in a double dimensioned string array. The array used is FI$(X,Y).

The X dimension of the FI$ array specifies the index, so you can index more than one file with the same handler subroutines. If you have just 1 file to index within your program, the X dimension will be 0. If you have 2 files to index within your program, such as a customer file and a product file, the X dimension will be 1. The X dimension is always 1 less than the number of indices you'll be using.

The Y dimension of the FI$ array specifies the number of "key-holding" strings that will be required, minus 1. Since you may have more than one index (based on how you specified the X dimension), the Y dimension is specified according to the largest index in your application. To compute how many key-holding strings will be required, take the key length you'll be using, and divide it by 255, and round down with the INT function. The number you get will be the number of keys in each string. Divide this into the total number of keys you'll be needing for the index, and round up. For example, to handle 1000 6-byte keys, you need 24 key-holding strings. The computation is:

```
KEY-HOLDING STRINGS REQUIRED = INT( 1ØØØ / INT(255/6) ) + 1
```

Since you subtract 1 when specifying the Y dimension, the Y dimension is 23. If this is the only index to be used in the program, the key-holding array would be dimensioned as:

```
DIM FI$(Ø,23)
```

If you have another file to be indexed in the same program, but the index for it is smaller, the dimensioning statement is:

```
DIM FI$(1,23)
```

You'll also need to dimension the LL%, LR%, PR% and PP% arrays according to the total number of files to be used, including the index files. This is discussed in the section that explains the random disk file handler. All of these dimensioning statements can be be executed in line 58000 of the handler. You can add your FI$ dimensioning command by appending:

```
,FI$(Ø,23)
```

. . . (or whatever your FI$ dimension is), to line 58000.

Specifying the Key Length

You'll need, first, to decide which file number you'll be using for your index file. Then simply modify line 58001 so that the logical record length, LL%(PF%), for your index file, is equal to the key length. For example, if you are using file 2 as an index, and it is to contain keys of 5 bytes each, line 58001 would contain "LL(2)=5". If your program will be using more than one index file, you will need to decide on file numbers and key lengths for the other indices in the same way.

The CLEAR Statement

Your program will need to clear enough memory for string storage to handle the contents of the FI$ array. Suppose your index is to contain 24 strings of 255 bytes each. If your clear statement would otherwise be "CLEAR 500", you will need to add 6120 to it, making it, "CLEAR 6620".

Merging in the Subroutines

Once you've loaded the required random disk file handler subroutines, the optional labeled file handlers, and you've modified your dimensioning and clearing statements, you can type-in or merge the memory index handler subroutines. They are listed and documented on the next few pages.

Memory Index Handler

Load Index from Disk to Memory Array:

```
585ØØ FIELDPF,255ASFD$(PF):FORX=1TOLOF(PF):PR(PF)=X:GOSUB5821Ø:F
I$(PI,PR(PF)-1)=FD$(PF):NEXT:RETURN
```

Search For Key:

```
5851Ø GOSUB58545:A%=Ø:A1%=1
58511 IFINSTR(A1%,FI$(PI,A%),FK$)>ØTHEN58513ELSEA1%=1
58512 A%=A%+1:IFA%<LOF(PF)-1THEN58511
58513 A1%=INSTR(A1%,FI$(PI,A%),FK$):IFA1%=ØTHENA!=Ø:RETURNELSEA!
=(A%*(INT(255/LL(PF))))+(A1%-1)/LL(PF)+1:IFA!<>INT(A!)THENA1%=A1
%+1:GOTO58511ELSERETURN
```

Get Key from Memory Index:

```
5852Ø GOSUB5854Ø:FK$=MID$(FI$(PI,PR(PF)-1),A1%,LL(PF)):RETURN
```

Change or Add a Key:

```
58530 GOSUB58545:GOSUB58540:MID$(FI$(PI,PR(PF)-1),A1%)=FK$
58531 FIELDPF,255ASFD$(PF):LSETFD$(PF)=FI$(PI,PR(PF)-1):PP(PF)=P
R(PF):GOSUB58300:RETURN
```

Compute Location of Key on Disk and in Memory:

```
58540 A%=INT(255/LL(PF)):PR(PF)=INT((LR(PF)-1)/A%)+1:A1%=(LR(PF)
-A%*INT((LR(PF)-1)/A%)-1)*LL(PF)+1:RETURN
```

Force Proper Key Length by Padding Trailing Blanks:

```
58545 FK$=LEFT$(FK$+STRING$(LL(PF)," "),LL(PF)):RETURN
```

Create a New Disk Index File:

```
58550 GOSUB58290:ONERRORGOTO58551:KILLFS$:GOTO58552
58551 RESUME58552
58552 GOSUB58250:FIELDPF,255ASFD$(PF):LSETFD$(PF)="":FORX=XTO1ST
EP-1:PR(PF)=X:GOSUB58300:NEXT:GOSUB58290:RETURN
```

Simple Variables

A%, A1% Temporary working storage. During a search, A% is the current element number within the key array. A1% is the position within the element. During fielding computations, A% is the number of keys per string and A1% is the MID$ position within the string for the current key.

A! For a search, A! returns the logical record number if the key is found. A! returns 0 if the key is not found.

X% Used as a counter when loading the keys from to the memory array. For the "create" subroutine, your program specifies X% to indicate the number of physical records that will be required in the index file.

PI% In multi-index applications, PI% tells the handler which index to use. For single-index applications, PI% is 0.

FK$ Provided by your main progam to tell the handler the key to find, or the key to be added. Returned by the handler when your program wants to know what the "nth" key is.

PF% Specifies the disk file number you are using for the current memory index handler command.

Array Variables

FI$(X,Y) Holds all the keys in 255-byte strings. The X dimension specifies the index number, PI%. The Y dimension specifies the element number. FI$(0,0) is the first 255-byte key-holding string for index 0.

LL%(PF%) Specifies the key length for the index file, PF%.

LR%(PF%) When your program wants to recall the "nth" key LR%(PF%) specifies the key number.

PR%(PF%), PP%(PF%) The current and previous physical record number (as documented for the random disk
file handler).

Line Comments

58500 (Load index from disk to memory array.)

:Field the index file buffer for 255 bytes.
:For each physical record in the index file, using X as a counter,
:The desired physical record is X.
:Call subroutine 58210 to GET the physical record.
:Load the 255-byte contents of the physical record into the corresponding element of the FI$ array.
:Repeat for the entire file length.
:Return to the main program.

58510 (Search for the key, FK$, in the index specified by PI%.)

:Call subroutine 58545 to insure that the key is the proper length.
:Load A% with zero to start at the first index array element.
:Load A1% with 1 to start our instring search at the first byte of the index string.

58511

:If the key is found in the current index string, you have a possible match, so go to line 58513.
:Otherwise, reset A1% to 1 so the next instring search will start at the first byte.

58512

:Add 1 to the index array element number, A%.
:If you haven't yet reached the end of the index array, as specified by the length of the corresponding index disk file, then go to 58511 to search the next index array element. Otherwise, you fall through to 58513.

58513

:Now you've either found a possible match, or you've reached the end of the index array. Check again by loading A1% with the position of the key in the index string.
:If it's zero, you didn't find the key, so load A! with zero to return a "not-found" condition.
:Otherwise, you've got a possible match. To see if it's a real match, you must verify that the key found starts at a position that is evenly divisible by the key length. To do this, compute A! as the relative key number.
:If A! is not an integer, it didn't divide evenly, so resume the search from the next byte of the current index string, by adding 1 to the current byte and going to 58511.
:Otherwise, you've found the key and A! has the record number, so return to the main program.

58520 (Get key, specified by LR%(PF%), from the memory index specified by PI%.)

:Call subroutine 58540 to do the computations.
:Extract the key, FK$, from the proper index array string.
:Return to the main program.

58530 (Change or add a key, FK$)

:Call subroutine 58545 to insure that FK$ is the proper length.
:Call subroutine 58540 to do the computations.
:Replace the proper section of the required index array string with the new key.

58531

:Field the disk buffer for the index file.
:Copy the index array string that has been modified into the disk buffer.
:Since you didn't do a GET, load PP%(PF%) with the current physical record number.
:Call subroutine 58300 to record the modified array string onto the diskette.
:Return to the main program.

58540 (Compute location of key on disk and in the memory array.)

:Compute A% as the number of keys per index string.
:Compute the corresponding physical record on disk, based on the logical record number of the current or requested key. The index array element will be the disk record minus 1.
:Compute the position that the key will start at, within the index string.
:Return, normally to 58520 or 58530.

58545 (Insure that the key, FK$ is the proper length.)

:Pad a string of blanks onto the key, and take the left portion of the padded key so it is the proper length, LL%(PF%).
:Return, normally to 58510 or 58530.

58550 (Create a new index file on disk.)

:Call subroutine 58290 to make sure the file, specified by PF%, is closed before you kill it.
:You need an ON ERROR GOTO in case the file is not found on disk.
:Kill the file whose name is specified by FS$. You are killing the file so you will be able to re-create it with the desired length.
:Jump over the "resume" line, to line 58552.

58551

:In the event there was no file to kill, resume at 58552.

58552

:Call subroutine 58250 to re-open the file. (The error handling routines in 58250 will cancel out the one just used.)
:Field the file buffer to hold a 255 byte string.
:LSET the string to null, which, in effect, loads 255 blanks into it.
:Starting from the highest physical record number, as specified by X, and working backwards to the first,
:Load PR%(PF%) with the record number.
:Call subroutine 58300, which will record the 255 blanks currently in the buffer, onto the current diskette physical record.
:Repeat until you get down to physical record 1.
:Call subroutine 58290 to close the file and lock in its length.
:Return to the main program.

Creating a New Index File on Disk

In most applications you'll only need to do this when you are originally setting up. The creation process involves building a disk file that contains nothing but blanks. The length of the file, in physical records, must be the length required to store the number of keys you plan to allow for. You can use the following formula:

```
INDEX FILE LENGTH =
INT ( NUMBER OF KEYS / INT ( 255 / KEY LENGTH ) ) + 1
```

To create the index file, load the desired index file length into X%, the file number to be used into PF%, and the file name to be used into FS$. Then call subroutine 58550. You will need to do this for each different index your application requires. Let's suppose, for example, you want to index the "PRODUCTS:1" file with an index file named "PRODINDX:1". The "PRODUCTS" file will use file buffer 1, and you plan to make "PRODINDX" file 2. Assuming you are planning for 1000 6-byte keys, the PRODINDX file will need to contain 24 physical records. Your commands are:

```
X=24                'SPECIFY INDEX FILE LENGTH
PF=2                'SPECIFY FILE NUMBER TO BE USED FOR THE INDEX
FS$="PRODINDX:1"    'SPECIFY THE INDEX FILE NAME
GOSUB 5855Ø         'CALL THE CREATE SUBROUTINE
```

You can delete lines 58550 through 58552 if don't need the "create" capability on a day-to-day basis. Then you can set up a separate program that handles the create function.

Loading the Index into Memory

It is up to your main program to open your files, including any index files that you may be using. It can do this by calling subroutine 58250 of the random disk file handler.

Once your files are open, your first task is to load the required index or indices into the FI$ array. To do this, first load PI% with the index number. The index number will normally be different from the file number. If your application only has one index, PI% should be loaded with 0.

Next, load PF% with the file number that contains your index. (Don't confuse this with the file number you are using for the file to be indexed.) With PI% and PF% loaded, simply GOSUB to subroutine 58500. Upon return, the FF$ array will contain the index.

Let's suppose, for example, that you've opened 2 index files. "PRODINDX:1" is handled as file 2 and "CUSTINDX:1" is handled as file 4. The commands required are:

```
PI=Ø:PF=2:GOSUB585ØØ   'LOAD THE PRODUCT INDEX
PI=1:PF=4:GOSUB585ØØ   'LOAD THE CUSTOMER INDEX
```

Searching for a Key

To search for a key, load it into FK$, load PI% with the index number, and PF% with the index disk file number. Then, call subroutine 58510. Upon return, A! will be zero if the key was not found. Otherwise, it will contain the logical record number.

To continue with the example we are using, we can search for product number "X302Z", and access the proper logical record from the "PRODUCTS" master file, with the following commands:

```
FK$="X3Ø2Z"        'LOAD THE KEY TO BE FOUND
PI=Ø : PF=2        'LOAD THE INDEX NUMBER, AND INDEX FILE NUMBER
GOSUB 5851Ø        'CALL THE SEARCH SUBROUTINE
IF A!=Ø THEN 1ØØØ  'JUMP TO A LINE TO HANDLE A  "NOT FOUND" CONDITION
PF=1 : LR(PF)=A!   'LOAD MASTER FILE NUMBER AND LOGICAL RECORD
GOSUB 582ØØ        'GET AND FIELD THE RECORD FROM THE MASTER FILE
```

In some applications you might not require a master file. In those cases, you might want to use the index for purposes of validating keys as entered by the operator and translating them into record numbers.

Adding or Changing a Key

We will discuss additions and changes together because, as the memory index handler sees it, an "add" command is really just a "change." This is because a new index contains strings of 255 blanks each. When you are adding a key, you are just replacing a series of blanks with the new key. When you change a key, you are just replacing a section in the index with the new key.

To add a key or change a key, specify the file number holding the index as PF%, the index number as PI%, load FK$ with the key to be added, and load LR%(PF%) with the logical record number. Then, call subroutine 58530. Upon return, the key will be in the specified position of your memory index array and will also be recorded onto the diskette.

Let's say that you have recorded a product description and price into logical record 325 of file 1, and now you want to record its product number into the index, being handled as file 2. Assuming, PF% is now 1, LR%(PF%) is now 325, and FK$ contains the key you want to use, the commands are:

```
PF=2           'LOAD THE DISK FILE NUMBER CONTAINING THE INDEX
PI=Ø           'LOAD THE MEMORY INDEX NUMBER
LR(2)=LR(1)    'INDEX RECORD NUMBER = MASTER FILE RECORD NUMBER
GOSUB 5853Ø    'RECORD THE KEY, FK$
```

As another example, let's say that you want to change the 200th key in the "CUSTINDX" file, which is being handled as file 4. If the memory array for the "CUSTINDX‘ file was specified as index 1, and you want to change the key to "L103", the commands are:

```
FK="L1Ø3"      'LOAD THE KEY
PI=1 : PF=4    'LOAD INDEX NUMBER AND INDEX FILE NUMBER
LR(PF)=2ØØ     'LOAD NUMBER OF THE KEY TO BE CHANGED
GOSUB 5853Ø    'RECORD THE NEW KEY
```

It is important to note that the memory index handler subroutines as they are shown are designed to handle unique keys. If you have 2 "L102" keys in an index, the handler will find only the first one. Because of this, it is important to do a search before you add or change a key, to verify that the new key is not already in the file. (In the case of a change, you'll need to delete the old key first, and then search to verify that there are no other keys in the file with the same value.)

Deleting a Key

To delete a key, you simply change it back to blanks and perform a call to subroutine 58530, just as you would if you were doing an add or change. You can change the key to blanks by simply setting FK$ equal to a null string. Subroutine 58530 will then change FK$ to the number of blanks equal to the key length.

Recalling a Key by Number

Subroutine 58520 looks up the key you wish to recall and loads it into the FK$ string. You specify the key to be recalled by loading PF% with the index file number, and LR%(PF%) with the relative number of the key you want to recall. Subroutine 58520 is provided so you can recall any key without doing a disk access. It comes in handy when you are reading the master file sequentially and you want to also retrieve the key that corresponds to any logical record in the master file.

To get and print the 567th key from the index being handled as file 2 and array index 0, your commands are:

```
PF=2 : PI=0          'LOAD INDEX FILE NUMBER AND INDEX ARRAY NUMBER
LR(PF)=567           'LOAD THE DESIRED KEY NUMBER
GOSUB 58520          'CALL THE SUBROUTINE TO GET THE KEY
PRINT FK$            'PRINT IT
```

Closing an Index File

There are no special considerations for closing an index file. Do it just as you would close any other file, by loading PF% with the file number, and issuing a "GOSUB 58290". Since the handler does a hard-put after each new key is added or changed, and the length of the index file has been pre-allocated, you can fail to close an index file and no harm will be done.

"MIH/DEM" — Memory Index Handler Demonstration

To test and demonstrate the memory index handler subroutines, you can modify the LFH/DEM program used to test the labeled random disk file handler. With the modifications we'll discuss, you can access any name and telephone number from disk by a short key of up to 6 characters. You might, for example, access a person's name and phone number by his initials.

If you haven't tried the LFH/DEM program, I recommend that you do so at this point. At minimum, you should read through the description of how it works.

For the MIH/DEM program, you will change the operation slightly. Instead of asking for a command, such as "A", "R", "C" or "D" (to add, recall, change or delete), the MIH/DEM program will just ask for the accessing key. So, if you want to access "JOHN JONES", you can type in "JJ", or whatever you want to use for the key. Then if the key is found, the information for "JOHN JONES" will be displayed, and you'll have the opportunity to change it or delete it. If the key is not found, the computer will ask you if you want to add it to the disk files. If you answer yes, by typing "A" to add, you can type in the name and phone number corresponding to the key, and the record will be added at the next available location in the file.

To list the contents of your file, you'll be able to type "L" instead of entering a key. To end the program and close the files, you'll be able to type "E". This design makes it so you can't use "L" or "E" for a key, unless you use a blank following the "L" or "E". Though the program may not be too fancy from an operator's viewpoint, MIH/DEM will at least show you how the memory index handler can be used.

Upon startup, the memory index demonstration program will ask you for the file name of your master file, and it will allow you to initialize it to zero records. Then it will let you enter the file name for your index file, and it will allow you to initialize (create) a new index. In most real-world applications, these functions would, of course, be "hard-coded" into the program. Here's how the startup sequence will look if you are creating new files:

```
RANDOM DISK FILE HANDLER -- MEMORY INDEX DEMONSTRATION

FILE NAME:                  TESTFILE:1
OPENING AS FILE 1...
INITIALIZE FILE?  (Y/N)     Y
INITIALIZED.
INDEX FILE NAME:            TESTINDX:1
INITIALIZE INDEX? (Y/N)     Y
LOADING INDEX...
NEXT EOF = 2 ,  LAST DEL = Ø ,  COUNT = Ø
ENTER THE KEY, OR
<L> TO LIST, <E> TO END...  ......
```

Now, if you enter a key, you know it won't be found, because you've just initialized the files. Let's go ahead and enter "JJ" as the key. Here's what happens:

```
ENTER THE KEY, OR
<L> TO LIST, <E> TO END...   JJ
KEY NOT FOUND!
ENTER <A> TO ADD IT...       ..
```

When you type <A> and press <ENTER>, the computer asks for the name and phone number. After you type them, the name and phone number are added to the master file ("TESTFILE" in this case), and the key is added to the index file, ("TESTINDX" in this example). Here's how it looks for the first record you add:

```
TYPE THE NAME:               JONES, JOHN
TYPE THE PHONE NUMBER:       333 333-3333
ADDING AS RECORD NUMBER:     2
NEXT EOF = 3 ,  LAST DEL = Ø ,  COUNT = 1
```

Notice that the labeled file handler subroutines took care of the decision of which logical record number to use, and they updated the file statistics.

Now that you have one record in the file, you can demonstrate the search capabilities. You can type in "JJ" whenever you want to recall the telephone number for John Jones. Here's what happens when you do:

```
ENTER THE KEY, OR
<L> TO LIST, <E> TO END...     JJ
NAME:                           JONES, JOHN
PHONE NUMBER:                   333 333-3333
PRESS <ENTER>, OR
<C> TO CHANGE, <D> TO DELETE... ..
```

Having recalled and displayed the record corresponding to the "JJ" key, simply press <ENTER> if you want to add or recall another record. Type <C> if you want to make a change to the name or phone number. Type <D> if you want to delete the record entirely.

After you've added several records, you can demonstrate the "list" command. It operates as it does in the LFH/DEM program, except now the keys are shown along with each logical record. Notice that no sorting is done. The records will be shown in the sequence that they are found in the files:

```
ENTER THE KEY, OR
<L> TO LIST, <E> TO END...  L
RECORD 2          JJ      JONES, JOHN
                          333 333-3333
RECORD 3          BR      ROBERTS, BILL
                          333 332-3214
RECORD 4          JS      JOHNSON, SAM
                          332 3Ø3-3322
RECORD 5          SA      ADAMS, SUE
                          931 987-6543
```

If you press <BREAK> at this point, you can see how the keys are stored in the memory array. Since you've allowed for 6-byte keys and only have one index in this demonstration, the first 42 keys can be shown by typing:

```
PRINT FI$(Ø,Ø)
```

Here's what we get:

```
      JJ    BR    JS    SA
```

Notice that the positions of the keys correspond to the logical record numbers used when recording the records to disk. Since logical record 1 isn't being used, you have 6 blanks instead of a first key. Then "JJ" starts at position 7, and "BR" starts at position 13. After the "SA" you have a series of blanks, so FI$(0,0) has a total length of 255 bytes. The key corresponding to logical record 43 will be at the first position in FI$(0,1). Since the demo program has allowed for 420 6-byte keys, you can inquire into FI$(0,0) through FI$(0,9) to see all of them.

The only other command provided in the MIH/DEM program is <E>. It simply closes the files, and returns you to "READY".

How to Set Up the MIH/DEM Program

Here are the steps you'll need to write the MIH/DEM program.

1. Load and test the LFH/DEM program, exactly as it is shown and explained in Chapter 7.

2. Merge or type-in the memory index handler subroutines, exactly as they are shown. They will occupy lines 58500 through 58552.

3. Add the following to the DIM statement in line 58000, so that the memory index will be dimensioned to hold 10 255-byte index strings:

```
FI$(0,9)
```

4. You will be using file 2 for the index file. To specify a key length of 6 bytes, line 58001 should contain the command:

```
LL(2)=6
```

5. Change the CLEAR command in line 1 so 4000 bytes are reserved for string storage, instead of 1000 bytes. Line 1 will now read:

```
1 CLEAR 4000
```

5. Delete the following lines from the LFH/DEM program:

```
1000, 1070 - 1075, 2000-2011, 2300, 2301, 2440, and 2500
```

6. Add the lines shown below.

Figure 10.1 *MIH/DEM — Modifications to LFH/DEM*

```
0 'MIH/DEM
1000 CLS:PRINT"RANDOM DISK FILE HANDLER -- MEMORY INDEX DEMONSTR
ATION":PRINTSTRING$(63,"=")

1051 LINEINPUT"INDEX FILE NAME:            ";FS$
1052 LINEINPUT"INITIALIZE INDEX? (Y/N)     ";A$
1053 IFA$="Y"THENPF=2:X=10:GOSUB58550
1054 PF=2:GOSUB58250
1055 PI=0:PRINT"LOADING INDEX...":GOSUB58500

2000 PRINT:PRINT"ENTER THE KEY, OR":LINEINPUT" <L> TO LIST, <E>
TO END... ";FK
2001 IFFK=""THEN2000ELSEIFFK="L"THEN2400ELSEIFFK="E"THEN2500

2010 PF=2:PI=0:GOSUB58510:IFA!>0THENLR(1)=A!:GOTO2310
2020 PRINT"KEY NOT FOUND!":LINEINPUT"ENTER <A> TO ADD IT... ";A$
2021 IFA$<>"A"THEN2000ELSE2200

2235 PF=2:LR(2)=LR(1):PI=0:GOSUB58530

2335 PRINT:PRINT"PRESS <ENTER>, OR":LINEINPUT"<C> TO CHANGE, <D>
 TO DELETE... ";CD$

2361 LR(2)=LR(1):PF=2:PI=0:FK="":GOSUB58530

2435 PF=2:LR(2)=LR(1):PI=0:GOSUB58520
2440 PRINT"RECORD";X;TAB(20);FK;TAB(28);FH(1)

2500 KV%=0:PF=1:GOSUB58400:GOSUB58290:PF=2:GOSUB58290:END
```

Index Sequential Accessing Methods

You often have two main requirements when dealing with large random access disk files. The first is to be able to access individual records quickly by specifying a search key, such as a customer account number or an inventory item code. The second is to be able to print or display records in that file in a given sequence. It is very common to want the printout to be sequenced alphabetically by search key.

In computer programming, ISAM is a term that stands for Index Sequential Accessing Method. An index to a book is an alphabetical list of topics covered in the book, along with the page numbers where each topic can be found. An index to a random disk file is similar. It is a list of the search keys that a user may enter to find the desired data within the file. Instead of referencing a page number, each search key in the index is coupled with a record number that tells the computer where to look in the main disk file being accessed.

The index itself is usually a separate smaller disk file. To the operator at the keyboard the process is quick, invisible, and automatic, but the program first looks in the index for the desired key, and then it goes to the main file to get or update the information related to the key.

Suppose, for example, that you want to keep information about a portfolio of investment stocks from the New York Stock Exchange in a random disk file. The information you want to maintain in the file might include such things as the date the stock was purchased, the purchase price, the number of shares, the price-earnings ratio, and the current closing price. You want your video display to instantly show the current information for any stock you request when you type in a short code, such as TAN for Tandy Corporation, or GM for General Motors. Furthermore, you want to be able to add and delete stocks at any time, and to be able to make printouts in alphabetical order by accessing code. Organizing your disk files with ISAM for access might be ideal in this case, especially if the file is large enough to make other methods of searching and sorting impractical.

The sequential aspect of ISAM accessing refers to its capability for keeping the index in an ascending alphabetical or numerical sequence. It does not refer to the other use of the word "sequential" in the context of "sequential files."

Though, in some varieties of ISAM, the index itself might be a sequential file, more often it is not. The file being indexed, (we'll call it the master file), is almost always a random access file. Since the index contains the accessing keys in ascending sequence, ISAM relieves you from the need to sort the master file.

The "M" in ISAM stands for "method." This may incorrectly suggest that there is just one standard ISAM. Actually, ISAM would more properly be called a capability, because there are dozens of methods for achieving it. Depending on the application, some are better than others because they all involve trade-offs in speed, disk storage requirements, maintenance and programming complexity.

Some versions of various programming languages, such as Radio Shack's Compiler BASIC for the TRS-80, provide ISAM as a built-in capability. Most people, however, elect to use the regular interpretive Disk BASIC provided with the TRS-80 because it is easy to use and there is no extra cost. It, like most languages implemented on microcomputers, does not have ISAM. You've got to devise your own schemes for disk accessing, or you may purchase certain software packages that add the capability through BASIC or machine language subroutines.

In the next few chapters, I will present my own ISAM subroutines. I think you will discover that they are comparable in performance, and in many ways, superior to the costly alternatives you may find. The development of these routines has been one of the most complex and time-consuming challenges that I have undertaken in BASIC programming. Even after developing a working prototype it took several months of work to refine the routines into a package of subroutines that are easy to implement.

I found that there is very little written on the subject. It seems that most vendors who are marketing ISAM capabilities want to keep the workings of their products a deep, dark secret. Most of the academic discussions published are just that. They don't go into the practical details and they are often incomplete. My goal is to give you the best discussion of the topic that you will find anywhere so you can implement the capabilities in your own programs, and if necessary, modify or translate them.

I call my index sequental accessing method TREESAM. It is called TREESAM because the index uses a particular organization on disk called a B-tree. Before we go into the specifics of TREESAM, let's look at various approaches to index sequential accessing to see the advantages and disadvantages of each. We'll follow the path that led me to the conclusion that the logic you'll find in TREESAM is the best approach for most ISAM applications.

I Meet ISAM

My first experience with index sequential accessing was several years ago. I was using a minicomputer to develop an inventory control program for a carpeting retailer. He wanted to keep track of about 6000 rolls of carpet. As each roll was received from the mill it was assigned a unique roll number and the description, length, cost, and other data was recorded in the computer. After carpet was taken from any roll it was necessary to access the roll on the computer by number and to deduct the yardage used. When a roll was depleted it was necessary to delete it from the file.

In addition, it was necessary to print and display individual rolls alphabetically by any of several hundred patterns and colors. Rather than using the manufacturer's code numbers, the patterns were commonly identified by descriptive or imaginative names such as "Northern Lights" or "Chocolate Chip." Since the names were long and sometimes prone to misspellings, we wanted to be able to type in just the first few letters to search out a particular pattern alphabetically. When the desired style was found, there was always the possibility that there was more than one roll, so we wanted our screen to be able to step through the file, forward and reverse.

As you can see, we needed two ways to access the file by key, and for either way, speed was important. A high volume of additions and deletions made our programming problem more difficult. The built-in index sequential facility that I had with that computer was quite limited. First, it required that the master file on disk be in ascending sequence according to the keys to be used. Since I wanted access by two alternative keys, my solution was to have two copies of the file on disk — one in roll number sequence, and the other in pattern and color sequence. A second limitation was that it was unable to handle the additions and deletions interactively. After new rolls were added or old ones were deleted, it was necessary to re-sort the files.

The index sequential accessing method that this particular minicomputer used was typical of some of the early approaches to the problem. I had to reserve an area of memory, the more the better, to hold each index. Upon opening the file to be indexed, a subroutine would be called. That subroutine would figure how many keys would fit in memory. Then it would read through the file and, depending on the amount of memory I reserved, it would store every fifth, tenth, or whatever'th key in memory.

To search for a particular record by its key, I would call a subroutine that searched the index in memory. This part was very fast. The memory index would tell the system where to start looking in the disk file. It would go to that point and read sequentially until the proper record was found. From the standpoint of speed it would have been ideal to have every key in memory, but since memory was limited, compromises in the number of disk accesses were necessary.

There were three main drawbacks of this indexing system. First, there was a delay of about a minute as the indices were built in memory when the files were opened. The second was the rather large memory requirement for holding each index, and the headaches that resulted when it was time to make modifications to the program. Finally, there was the problem of maintenance. When we speak of maintenance in respect to ISAM files, we are talking about any additional procedure that is required periodically to rebuild or reorganize the index. In this case, it was the sorting operation that was required after sessions in which there were additions and deletions.

MISAM — In-Memory Index Sequential Accessing Method

A few years later, I wrote a general ledger program for a client using a TRS-80 Model III. For data entry we needed a fast way to verify the validity of general ledger account numbers. The master file contained over 400 accounts, and anything more than a half second would have been unacceptable.

We needed to accommodate their existing numbering system in which the numbers in the chart of accounts consisted of 4 digits, a decimal, and 2 digits to the right of the decimal. To illustrate, "4101.21", "3213.01", "3201.02" and "311.00" could all be valid numbers that might be used as keys to access accounts in the master file. One digit position would signify a particular department or division, another would indicate the account classification, and so forth.

Beyond the requirement for fast access at the keyboard, we needed printed reports and financial statements. Some reports, such as the trial balance, were always to be printed in account number sequence. Others, such as the income statement and balance sheet, were to be customized into departments and categories that wouldn't necessarily be in account-number sequence. Within a given category, however, it was usually desirable to print the accounts in ascending sequence. We wanted to be able to tell the program to display a heading, such as "Administration Expenses", print all accounts within a certain number range in sequence, and then to display a subtotal.

To do all this without time-consuming sorts or searches through the disk file required an index sequential accessing method. My solution was combine the KWKARRAY, SEARCH2, and Move Data techniques shown in *BASIC Faster and Better and Other Mysteries* to develop a method which I called MISAM, for in-Memory Index Sequential Accessing Method. The idea behind MISAM was to store all the keys in protected memory. With each key I stored a 2-byte pointer. The pointer was a number that told the system which record in the master file corresponded to the search key. To save memory and to insure that they would sequence properly, the 7-position account numbers were compressed to three bytes each. The first two digits were stored in one byte, the second two in the next byte, the decimal was ignored, and the final two digits were stored in the third byte. In total, 5 bytes of memory were required per key, so 500 accounts meant we needed to protect 2500 bytes.

Upon starting a session, all the account numbers and their corresponding pointers would be copied to high memory from a special disk file that contained them. This operation just took a few seconds since the keys and pointers were contained in 10 consecutive 256-byte records. As each record was accessed, a block move was done to copy 256 bytes from the disk buffer to the intended region in protected memory.

All additions of new accounts were made at the next available record position in the master file, according to the logic of the random disk file handler we've discussed. There was no need to sort it. After each addition on disk, MISAM inserted the key and its pointer at the proper position in the memory index by moving up all subsequent keys. When a record was deleted from the disk file, MISAM moved down all subsequent keys in memory. At the end of a session in which there were additions or deletions, the new index was copied in its entirety, from the protected memory area back into the disk file.

To achieve speed, all key insertions and deletions were handled in memory by the machine language subroutine. This presented a potential problem. In the event the updated index was not recorded back to disk due to a power falure or some other abnormal end to a sesson, the index would be incorrect.

I solved this problem by designing a flag that would be set in the disk file if an addition or deletion was made. After properly ending a session that flag was reset. If, when starting the next time, the program found that the flag hadn't been reset, it would know that there was a possibility the index on disk was incorrect. In such a case, a special program would be run to re-build the index. It could do this because each key was also stored in the master file. The recovery program would clear the old index. Then it would read sequentially through the master file, re-inserting each key into the index. Finally, the recovered index was written back to disk and the indicator flag was reset.

The only drawback to MISAM was that it was memory-dependent. The larger the keys required, and the more keys I had, the more memory required for the index. It was unsuitable for applications with a very large number of items to be indexed because of memory capacity limitations.

Sequential Index Files

Over the past few years I've seen other approaches to index sequential accessing. The most common approach is to leave the master file unsorted. Generally it is left in the sequence that evolves as new records are added. A separate random file contains the search keys, one for each record in the master file. With each search key is a number that points to a record in the master file. Since the index file contains keys and pointers only, it is much faster to sort and search it than to sort or search the master file, which contains more data.

Under this method, the index file is kept in ascending sequence. After sessions in which there have been additions or deletions it must be re-sorted. To find a record in the master file by key, the program searches the index. It may do this by reading the index from the beginning, until a key is found that is greater than or equal to the search key. Then, if a match is found, the indicated record in the master file is accessed. This approach is fine for small indices, but it can be quite time-consuming when the index contains more than a few dozen items.

A sequential search for a key in a disk file index containing "n" items would use the following logic:

1. Current-position in index file is 0.

2. Current-position equals current-position + 1.

3. Is current-position greater than "n"?
If yes, go to step 6.

4. Get current-key from the index file.

5. Is the search-key less than the current-key?
If yes, go to step 2.
If no, go to step 7.

6. Report a "not found" condition and end the search.

7. If search-key is equal to the current-key, get the record from the master file that is specified by the pointer stored with the key. If the current-key is not equal to the search key, report a "not found" condition.

Binary Search of Index Files

A more advanced variation is to use a binary search of the index file. Again we assume that the index file is in ascending sequence because of a sort that the program did following the last run. To illustrate a binary search process, pretend for a moment that you are a robot, and you want to look up the word "reflex" in a dictionary. First you would determine that there are, say 1041 pages in your dictionary. Then you'd divide 1041 by 2 to find the middle page, 520. Turning to that page you'd see that "masthead" is the last word. Since "reflex" is greater than "masthead" you can halve the size of your search by making the left limit equal to the middle page plus 1, or 521. You find a new mid-point for your search by adding 521 and 1041 and dividing by 2. That tells you to look on page 781. The last word on page 781 is "segno". Now, since "segno" is alphabetically greater than "reflex", you can eliminate everything to the right of "segno". Again you halve the size of your search by making the right limit equal to the prior mid point minus 1. This process of comparing and halving the size of your search continues until you find the page that contains the word you are seeking.

A binary search for a key in a disk file index containing "n" keys would follow the logic shown below:

1. Left-limit for the search is record 1.

2. Right-limit for the search is record "n".

3. Midpoint equals INT((left-limit + right-limit) / 2).

4. Get key at the midpoint. Is search-key greater?
If yes, left-limit equals midpoint + 1.
If no, right-limit equals midpoint - 1.

5. Is left-limit still less than the right-limit?
If yes, go to step 3.
If no, you've found the first key that is greater than or equal to the search key!

6. If the key is equal to the search key, get the record that is specified by the pointer from the master file. If the key is not equal to the search key, report a "not found" condition.

On average, a binary search of an index file can provide an improvement over the sequential search, but you still have the problem of additions and deletions. Though you can easily mark a record as deleted (recovering the space it occupies later), in most cases it just isn't practical to add new keys by inserting at the proper position and moving the others up.

It is more practical to store the new keys at the end of the file (or in a separate file) and to sort or merge them into proper sequence at the close of the session. One approach is to re-sort the entire key index, including the keys that have been added. The other way, more acceptable if the index is large, is to just sort the new keys that have been added, and then to merge them into the key index.

Index Files Organized as Linked Lists

One solution to the problem of adding and deleting keys is to organize the index as a linked list. In this organization the index is "logically" in ascending sequence, but "physically" it probably is not. In the linked list organization, each key is stored along with the usual pointer to a record in the master file, as well as a second pointer.

That pointer tells the program the position of the next key in sequence. (To avoid confusion, we'll call this number a "link" rather than a "pointer".) To read the master file in the desired sequence without a sort, the program simply follows the path through the index file, back and forth from one key to the next. After each key is accessed, the master file record it points to is gotten. The information is printed or processed, and the next key (if any) is gotten by following the link from the current key.

Though a linked list automatically keeps your index in sequence as you add and delete keys, this technique is usually unsuitable when you need to make quick searches. A sequential search through a linked list is time consuming because there is a lot of work your program has to do when following the links, and if the keys were added in anything other than an ascending sequence, there will probably be plenty of back and forth movement (or "thrashing") that your disk drive head will have to do during the search.

Binary Search of Linked Lists

A big disappointment is that a binary search cannot be used efficiently on a linked list. If you look at the binary search process we discussed earlier, you'll see that it relies on the physical placement of keys within a file. From start to end the keys must be in ascending sequence for a binary search to work.

There is one compromise that we can make if we want to combine the advantages of a linked list and a binary search. The solution is to periodically reorganize or sort the file so that, though it is a linked list, it is in ascending sequence. After reorganizing, our program can make a notation in a special disk field to record the position of the current last key in the file. As new keys are added, we link them as we would normally do with a linked list, but we make sure that they are stored beyond our prior end-of-file. With this plan we must forgo the luxury of reusing deleted records until the file is next reorganized. Deleted keys must be left in the file, and coded for later deletion.

Now, with this special breed of linked list, whenever we want to search the index we can use a binary search, as long as we confine it to the front portion of the file. That is, our binary search can be reorganized. At the conclusion of the binary search, we must check to see if any new keys have been linked in to the immediate left of the key we've found. Assuming our binary search has located the first key that is greater than or equal to the search key, if it shows that another key has been added, we must follow the links to see if one of the recently added keys is greater than or equal to our search key.

Tree Structures

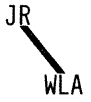
Combining the idea of a linked list and a binary search leads us to another approach to the problem of keeping an index in ascending sequence while making it easy to search.

Computer science texts are full of discussions about trees and tree structures. To illustrate the basics, let's assume that you want to index a file of New York Stock Exchange investments. To call up information on any common stock in a portfolio, you want to be able to use a standard 3-letter ticker tape code. As a further requirement, let's insist that, without regard to the order in which they were added to the files, the keys must be readable in ascending alphabetical sequence at any time without the need to stop and sort the index.

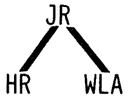
To get the concepts, picture the process graphically first. You are starting with an empty index and you want to add the following keys and the information related to them:

```
JR  WLA  HR  IBM  ELG  CDA
```

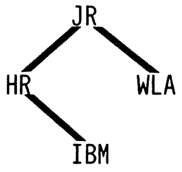
The computer program first adds JR to the index. There's no problem with that as long as it's the first and only key. Now add WLA. The program sees that WLA is greater than JR, so it links WLA as the right son of JR:



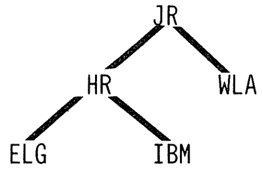
Now add HR as a key. The program starts with JR, the root of the tree. It compares HR to JR. Since HR is less and JR has no left son, it adds HR as the left son:



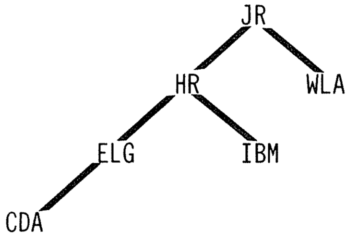
Next add IBM. The program again starts at the root key (JR), it compares IBM to JR, and since IBM is less, it follows the path down to HR. Since IBM is greater than HR, the program adds IBM as HR's right son:



Now add ELG. The program follows the path from the root key down to HR, sees that ELG is less, and adds it as a left son to HR:



The last key to add in this session is CDA. Using the same logic, the tree is now:



Each position where a key is stored in the tree is called a "node." If you consult your Webster's dictionary, a node is defined as "an entangling complication, a predicament." When you continue through the definitions you'll find that a node is also "a point at which subsidiary parts originate or center." That's the meaning we are using. A node is a decision point. In the binary tree, each node poses the question of going left or right.

To search for any key the program starts at the root node. At each node, the search key is compared. If a match is found, the search is done. If the search key is greater, you follow the link to the right son. If it is less, follow the link to the left son. If the comparison has shown that you should follow a link to a left or right son and no link exists, the search has ended with a "not found" condition.

Let's trace a search for the key "IBM". Start at the root, JR. Since IBM is less, follow the left pointer to HR. Since IBM is greater, follow the right pointer from this node. Now having arrived at the node containing IBM, the program sees that it is equal to the search key. The search is successful, so the program gets the external pointer stored with the IBM key that tells which record to access in the master file.

To record this tree on disk, you need to handle the organization with numbers. As each key is added to the tree, it is added at the end of our file. Then the left or right pointer for its successor in the search is adjusted to point to the new key. Actually we've talked of three pointers that are associated with each key. In addition to the left and right son pointers we have what I called an external pointer.

The external pointer has nothing to do with the organization of the tree. Instead it points to the location in the master file that is indexed by a given key.

To make this clear, let's look at how this tree would be organized on disk. You started at the first record position. As each new key was added you recorded it in the next position in the tree file. The related data that contained information about the common stock was stored in the next record of the master file:

Record	Key	Left Son	Right Son	External Ptr
1	JR	3	2	1
2	WLA	Ø	Ø	2
3	HR	5	4	3
4	IBM	Ø	Ø	4
5	ELG	6	Ø	5
6	CDA	Ø	Ø	6

Notice that each external pointer in the tree is equal to the record number for the key. In more complex examples this might not be the case because of deletion handling and operations that might prompt you to reorganize the tree.

Before we continue our discussion, it will be helpful to introduce some more terminolgy. Each node in the binary tree can have 0, 1 or 2 sons. If a node has two sons, they are called brothers. Looking the other direction, two brother nodes always have one father node. Each position going up or down in the tree is called a level. The example shows a tree with 4 levels. The number of levels corresponds to the largest number of disk accesses that will be required to find any key in the tree. You number the levels by starting at the root. Thus, level 1 contains JR. Level 2 contains HR and WLA. Level 3 contains ELG and IBM. Level 4 contains CDA.

In larger trees, nodes on the same level that share the same grandfather can be considered cousins. Those that share the same great grandfather can be called second cousins. A leaf node is one that has no sons. In our example, CDA, IBM, and WLA are in leaf nodes. All other nodes, including the root node, may be called branch nodes.

Traversing a Tree

One of the goals in using a tree structure was to be able to read the keys in sequence without having to sort the index. This operation is called traversing a tree. As a practical matter in programming the traversal of a tree, I've found the easiest way is to develop a subroutine whose sole function is to get the next key. Each time you call it, wherever you are in the tree, it returns the next key.

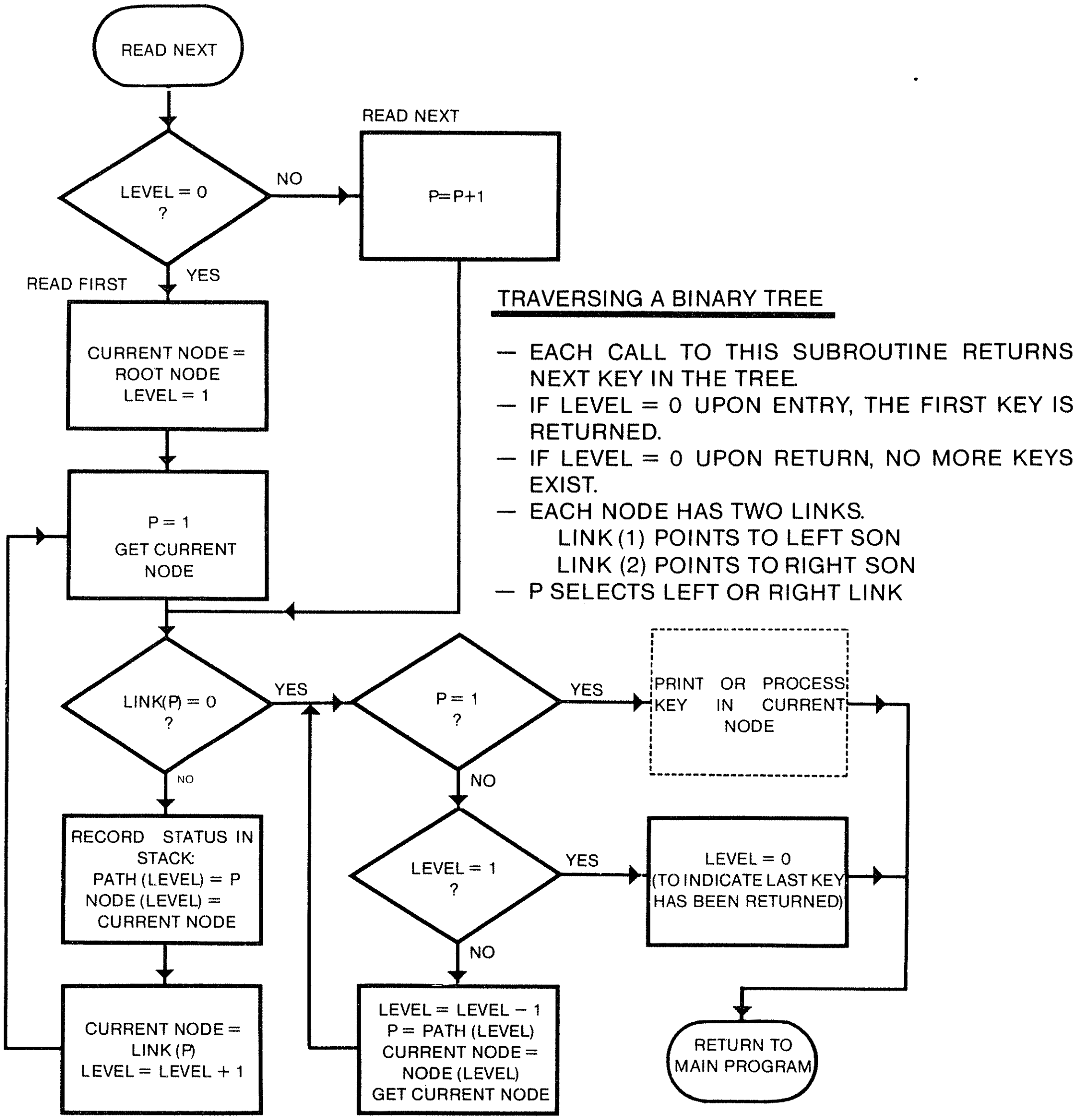
Remember, when traversing a tree, your program only takes one step at a time. Wherever it is in the tree, it can't see the whole picture. It is like exploring a dark cave where you can't see your surroundings. One requirement in a program like this, as when exploring a cave, is to keep exact notations of how you got where you are. At each decision point, you have got to note whether you turned right or left, so when you come back to it you don't go the same way again.

The operations in traversing a tree are repetitive in that the same decisions must be made from each large branch, down to smaller branches, and finally to each leaf. You start at the root and use the same routines over and over again, going down to the lowest levels, until eventually you end up back at the root. This type of repetition is called recursion.

Some programming languages, such as Pascal, handle recursion automatically, but in BASIC you need to construct a structure to hold the required information. Though the term is not often used in BASIC programming, we will call the construct a "stack". In a program to handle the traversal of a tree the stack takes the form of two integer arrays. Both of the arrays are dimensioned so one number can be stored for each level in the tree. Thus, if the tree has 10 levels, you give each of the arrays a dimension of 10. The "node" array maintains a record of the nodes, by number, that are between the current position and the root node. The "path" array maintains a record of which path you have been following, left or right, on each level between the current position and the root node.

Shown below is a flow chart of a subroutine that handles the traversal of a binary tree. To make the subroutine work, you need a way to initialize it so the first key returned will be the lowest key alphabetically in the tree. You also need a way for it to report back to the main program when the end of the index has been reached. To return the first key, before calling the subroutine, you set the variable that keeps track of the current level to zero. After each call you must leave the contents of the stack, as stored in the path array and the node array, unaltered, so that the subroutine can continue next time at the correct position. If the program finds that the current level equals zero after any call, it knows that there are no more keys in the index.

Figure 11.1 — *Traversing a Binary Tree*



TRAVERSING A BINARY TREE

- — EACH CALL TO THIS SUBROUTINE RETURNS NEXT KEY IN THE TREE.
- — IF LEVEL = 0 UPON ENTRY, THE FIRST KEY IS RETURNED.
- — IF LEVEL = 0 UPON RETURN, NO MORE KEYS EXIST.
- — EACH NODE HAS TWO LINKS.
  LINK (1) POINTS TO LEFT SON
  LINK (2) POINTS TO RIGHT SON
- — P SELECTS LEFT OR RIGHT LINK

READ NEXT
LEVEL = 0 ?
NO
READ NEXT
P=P+1
READ FIRST
YES
CURRENT NODE = ROOT NODE LEVEL = 1
P = 1 GET CURRENT NODE
LINK(P) = 0 ?
YES
NO
P = 1 ?
YES
PRINT OR PROCESS KEY IN CURRENT NODE
NO
LEVEL = 1 ?
YES
LEVEL = 0 (TO INDICATE LAST KEY HAS BEEN RETURNED)
NO
RECORD STATUS IN STACK: PATH (LEVEL) = P NODE (LEVEL) = CURRENT NODE
CURRENT NODE = LINK (P) LEVEL = LEVEL + 1
LEVEL = LEVEL − 1 P = PATH (LEVEL) CURRENT NODE = NODE (LEVEL) GET CURRENT NODE
RETURN TO MAIN PROGRAM

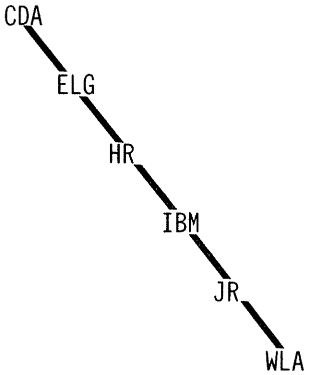
Problems With Binary Trees

At first glance, the binary tree might seem to be an ideal solution to the problem of organizing an index so it can be searched quickly, and so additions and deletions can be made, while making the index readable in ascending sequence without sorting. If you spend a few minutes "playing computer" and drawing the trees on paper, though, you soon notice some problems.

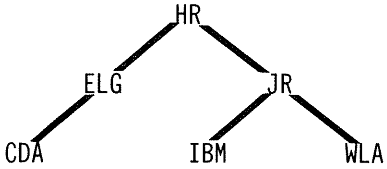
What if the keys are added in a sequence that is already ascending? Suppose you added them in this order:

```
CDA  ELG  HR  IBM  JR  WLA
```

Your tree would look like this:

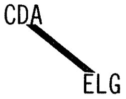


To search for WLA, you must go though the five preceding keys. That's quite inefficient and defeats the purpose. As you can see, the sequence that keys are added determines how the tree will grow. Ideally, you want the tree to be balanced. That is, you want it to grow in an even, triangular pattern from top to bottom to minimize the maximum path length from the root to the lowest level. When the keys come in at random, you are more likely to achieve a balanced tree. The first example was a little better, but it still wasn't as good as it could be. The following illustration shows the same keys organized in a balanced tree:

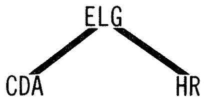


The problem of balancing a tree can be solved by more complex program logic that is called when insertions and deletions are made. To give a simple example, let's build a tree by adding CDA, ELG, and HR.

CDA and ELG make the following tree:



Now, when you add HR, you can roll ELG into the root node position to keep the tree in balance:



It looks simple on paper, but as the tree gets more complex the balancing of one section of the tree can cause an imbalance in another. The addition of one key may initiate a ripple effect that in a recursive subroutine might re-arrange the whole tree.

The balancing act for a binary tree can get so burdensome that it is impractical. Even with a balanced tree you still may require a large number of time-consuming accesses when you do a search. Assuming that each node is a disk record, each level in the tree represents one GET. Though the number of keys handled doubles with each level, the worst-case performance in a balanced binary tree is likely to be unacceptable. To illustrate, if you have a tree with only 63 keys, 32 of them will require the expense of 6 accesses to reach.

B-Trees

In a binary tree you store one key per node, and each node has two sons. If it is balanced, each level increases the capacity by a factor of 2. What if you have, say, ten keys per node? Now you can find any one of ten keys in 1 access, any one of 110 keys in 1 or 2 accesses, and any one of 1110 keys in, at most, 3 disk accesses. You can store many keys in a single 256-byte record, so why not take advantage of the dramatic jumps in magnitude that you can get!

Such a tree, having more than one key per node, is called a multiway tree. I first "discovered" the idea, as applied to index sequential accessing methods, when reading through a user manual for Data General minicomputers. The thought of a practical seaching method that efficiently handled insertions and deletions without time-consuming sorts or reorganizations amazed me.

After playing with multiway trees on my own I discovered that I needed more help. Two books stood out as having the best discussions on the subject, and I recommend them to you if you want more background information. One is *The Art of Computer Programming, Volume 3, Sorting and Searching* by Donald E. Knuth. The most complete practical discussion is in *Algorithms + Data Structures = Programs* by Niklaus Wirth, but you'll need some grounding in PASCAL to understand it.

The particular breed of multiway tree they discussed, called a B-tree, is clearly superior for use as an accessing system. A B-tree has some special characteristics:

- The programmer, within some constraints, decides the maximum number of keys that will be held in each node. This number, "m", must be evenly divisible by 2. For our purposes, the maximum node size will be determined by the number of items (keys and their pointers) that can be stored in a single physical record of 256 bytes.

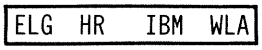
- To minimize the balancing required during insertions and deletions, each node need not contain "m" keys, but every node, except the root, must be at least half full. The root node can be less than half full.

- All leaf nodes are on the same level, so in this respect, a B-tree is always completely balanced. Every key in a branch node has a left son and a right son. Thus, if there are "k" keys in a branch node, that node will have k + 1 sons.

To see how these concepts work, let's picture the growth of a B-tree in which there will be a maximum of 4 keys per node. During the example, you will be starting with an empty tree. Then you will add the following list of keys:

```
WLA  HR  IBM  ELG  CDA  JR  WCI  PPW  ROR  CHH  ABT  BGE  AMR  KMG
ARC  PZL OXY  IPS  SOH
```

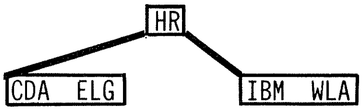
As the first 4 keys are added, they fill the root node. As each key arrives within a node, it is inserted into the proper position, and subsequent keys in that node slide over to make space. Since WLA arrives first, it is stored in the leftmost position of the node. When HR arrives, WLA is moved over into the second position, and HR is stored in the first. IBM is then inserted by moving WLA over to make space in the second position. Finally, ELG is recorded by sliding HR, IBM, and WLA one more position to the right. The resulting node is:



The most important characteristic of B-tree growth is illustrated when you add the the fifth key, CDA. Since its insertion would cause the first and only node to overflow, you split the node into 2 nodes and pass up the middle item, which in this case forms a third node. To find the middle item, the program temporarily allows the node to exceed its allowed capacity, and the new key, CDA is put into its proper position:

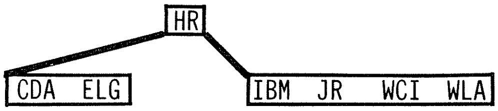
```
CDA   ELG   HR    IBM   WLA
```

Now it can determine that HR is the middle item. After the split, here's the tree:

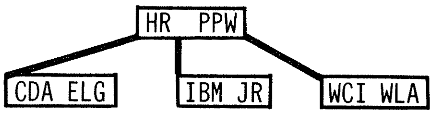


Notice that you've followed the rules. Each node, except the root, is at least half full. The tree has two levels, and both leaf nodes are on the same level. The only branch node (in this case, the root node) has 1 item, and 1 + 1 sons.

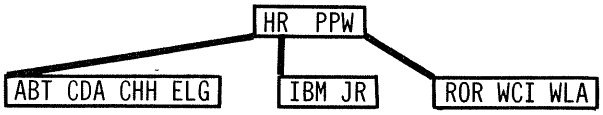
To continue the example, let's insert two more keys from the list: JR and WCI. Notice that all insertions occur at the lowest level in the tree, except, as we will see, when an insertion results from the splitting of a node.



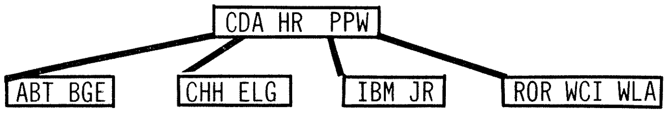
Now you want to insert PPW. Since PPW is greater than HR, it should be inserted in the node that is HR's right son. Such an insertion would cause an overflow of the node containing IBM, JR, WCI and WLA, so you split it and pass up the middle item. In this case, the middle item happens to be PPW, the key you are adding:



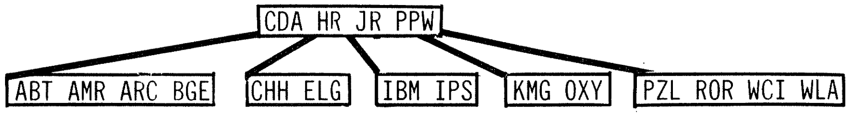
The next three keys to arrive from the list are ROR, CHH and ABT. The insertion of these keys does not require splitting:



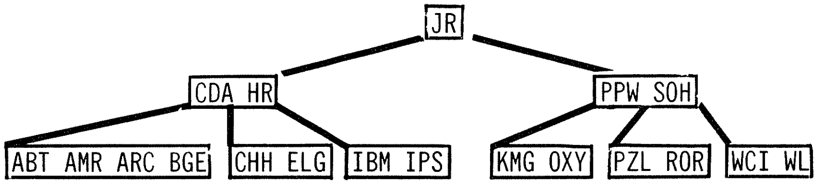
Now, when you add BGE, it causes the node containing ABT, CDA, CHH, and ELG to overfill. You split and pass the middle item up to the root. In this case the middle item is CDA. Notice that it is placed in its correct position within the root node:



The next keys to arrive are AMR, KMG, ARC, PZL, OXY, and IPS. Notice that, after they've been inserted, the root node in the tree has become full. Also notice that the next key, SOH, will cause the node that contains PZL, ROR, WCI, and WLA to overflow:



In this case you will be splitting a node, and the addition of the key being passed up will cause the root node to split. Any time the root node splits, you create a new root and the tree grows by another level. Here's how the tree looks after SOH is added:



Notice again that you've followed the rules. All leaf nodes are on the same, and lowest level. Each key in the branch nodes has a left and right son. All nodes except the root node are at least half full.

Since the B-tree example uses a node size of 4 keys, you can calculate that a three-level tree has a maximum capacity of 149 keys. (That's 4 on the first level, plus 20 on the second level and 125 keys on the third level.) Since each node except the root must be at least half full, you can calculate that a three-level tree stores a minimum of 17 keys. (That's 1 on the first level, plus 4 on the second level and 12 on the third level.) In actual practice, the point at which a B-tree will split and grow by another level will be somewhere between, depending on the sequence in which keys are added or deleted.

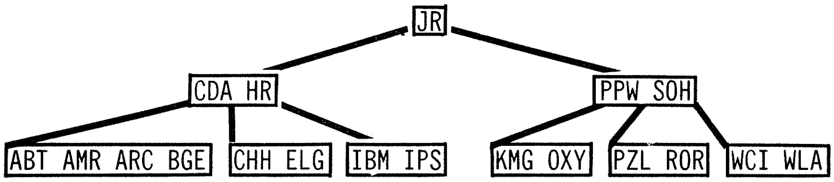
Another calculation that you can make is to determine how many nodes will be required to store a given number of keys. Let's say you want to allow for a maximum of 1000 keys and you have a maximum node size of 20 keys. In the worst-case situation all the nodes would be half full with 10 keys, and the root node would contain only 1 key. The maximum number of nodes required is equal to 101. That's gotten by subtracting 1 from 1000, dividing the result by 10, rounding up to the nearest whole number, and adding 1 for the root node. The minimum number of nodes required is 50, gotten by dividing 1000 by 20 and rounding up to the nearest whole number.

Again, actual performance will be somewhere between the minimum and maximum.

Searching a B-tree

The searching operation on a B-tree is similar to the process used to search a binary tree. You start at the root node, make comparisons and follow the pointers until either you find the key you are seeking or determine that the key isn't in the tree. The big difference is that you now have more than one key per node. A single disk access brings anywhere from two to a few dozen keys into memory where they can be examined much more quickly. Because each node may have many sons, the search spans fewer levels, and consequently, fewer disk accesses are required.

Let's follow the path that the computer follows in a typical search. Using the tree from the last example, search for ROR:



First the program gets the root node, which contains the key JR. Since ROR is greater than JR, it follows the right pointer down one level. It gets the node containing PPW and SOH. Since SOH is the first key in the node that is greater than the search key, ROR, you follow SOH's left pointer down to the node that contains PZL and ROR. Finally you search that node, and the program determines that ROR is present.

Assuming that the B-tree is being used to index another file, ROR has an external pointer stored with it (not shown) that tells the program which record to get in the master file.

Notice that the search required three disk accesses, one for each level. This was a worst-case search of the tree. Had you been searching for CDA, HR, PPW or SOH, the search would have only required two accesses. If you were looking for JR, the search would have terminated immediately at the root node, with only one disk access.

As an example of an unsuccessful search, try AMC. First compare AMC to JR. That sends you down to the node which is JR's left son. Since CDA is the first key in the CDA, HR node that is greater than the search key, you follow CDA's pointer down to its left son. That node contains ABT, AMR, ARC, and BGE.

The first key in the node that is greater than the search key is AMR. Since this node is a leaf node (it has no sons), the search is over.

If you try other search keys that are not in the tree, you will find that an unsuccessful search will always end at a node on the lowest level of the tree. Within the node, the search terminates at a position that may be to the immediate left of any one of the keys. If the search key is greater than any in the node, the search terminates at the immediate right of the highest key in the node.

The final position in an unsuccessful search is important if you want to add the new key to the tree. That position is where you will be inserting it. The program also needs to remember how it got to the final position. For each level that you covered, it has to know which node you passed through and which pointer within that node you followed. This information can be maintained in a stack like you used in the description of traversing a binary tree. The stack is actually two arrays (or a double dimensioned array). One element in each array is required for each level of the tree.

If the node is not yet full, the new key can be inserted at the proper position without further complication. If the node is full, as it is in the example of the search for AMC, it will need to be split, with the middle item being passed up to the father node. Here's where the stack comes in. As you go up one level to insert the emerging item, one array tells you which node was the father so you can get it again. The other array tells you the position you were at when you exited the father node during the search. The item being passed up is inserted there. If subsequent splits result, even if they ripple all the way up to the root node, the information in the stack guides the program to the correct positions within the correct nodes.

As a general rule, a search is always required before an insertion or deletion in a B-tree so that you can get to the node and position that will be affected. The search also sets up the stack so it will contain the information the program may require in the event you need to split one or more nodes.

B-Tree Deletions

One of the most important features of a B-tree as an index is that keys can be deleted at any time, and the index remains balanced and in sequence. A program or subroutine that handles B-tree deletions can perform logical steps that, when necessary, merge or dispose of nodes that have become underfilled. If a large number of keys are deleted a tree can eventually decay back to one node. During the process, all the rules for B-tree validity are followed. Any space on disk that is made available by deletions is immediately recoverable for subsequent insertions, should they occur.

Before a deletion, a search is necessary so that the node and position of the key to be deleted can be identified. Alternatively, the key to be deleted can be located by operations that read the keys of the tree in sequence. In either case it is important that the information in the stack be valid so that the program can go back from the key, through its ancestors, all the way back to the root if required.

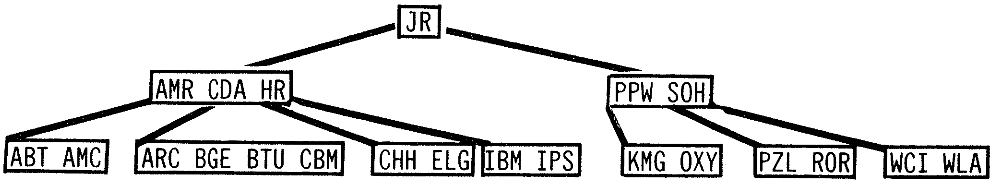
The most simple deletion situation occurs when you want to delete a key from a leaf node that is more than half full. The program simply removes that key and moves any keys that were to the right of it in the node back one position. Since the node will remain at least half full, no other operations are necessary.

When the deletion of a key from a non-root leaf node causes it to become less than half full the process gets a bit more complex. The program must perform an operation called underflowing. If the node has a right brother, you get it. If it has no right brother, you get the left brother. Now, if the brother node is more than half full, you can move one or more keys from it into the node that is less than half full. As you do this, you move one key from the brother node up to where the father was. The key from the father node, and any excess keys from the brother, are then rolled down to the node you are replenishing.

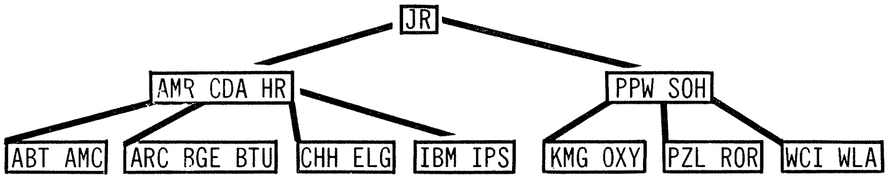
In the case where the brother node is, itself, only half full you cannot borrow any keys. Instead, you have to merge the two brother nodes, along with the key that separates them from the father node, into one node. Should this operation make the father node less than half full, you go up a level and perform a similar underflow operation on it. In an extreme case, the underflow operations will ripple up to the root node.

When a key to be deleted is not in a leaf node, you replace it with its successor, which is always in a leaf node. That is, you move the next item that follows in alphabetical sequence up to the branch node where the deletion is taking place. Then you delete the successor. If the root node becomes empty as the result of any deletion, the tree shrinks by one level.

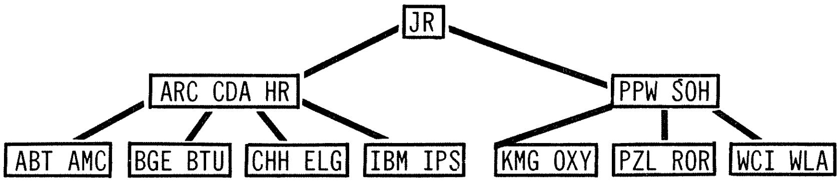
To illustrate B-tree deletions, start with the tree shown below. It is obtained by adding the keys AMC, BTU, and CBM to the prior example so that you can see more situations:



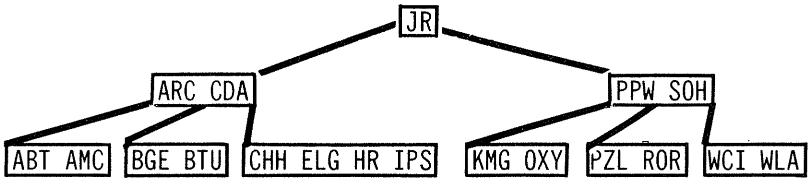
Let's say that the first key the operator wants to delete is CBM. As you can see, it is in a leaf node that is more than half full, so the deletion is easy. The node where it resided now contains ARC, BGE and BTU:



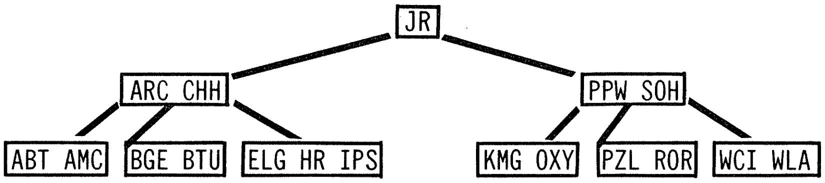
Now try something a little trickier. Delete AMR. It is in a branch node, so replace it by its successor, ARC. Then delete ARC. After deleting ARC from its original node, the node is still at least half full, so you are done. The resulting tree is:



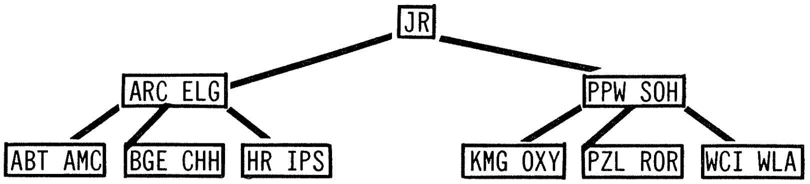
Now delete IBM. Notice that it is in a leaf node that is half full and that leaf node has no right brother. Its left brother is also minimally full so you can't borrow any keys. Instead, you combine the two nodes with the key in the father position, HR. The removal of the HR from the father node left it no less than half full, so the process is complete:



Next, let's see what happens when you delete CDA. Since CDA is in a branch node, you replace it by its successor, CHH. Removal of CHH from the leaf node it occupied leaves it more than half full, so you are okay:



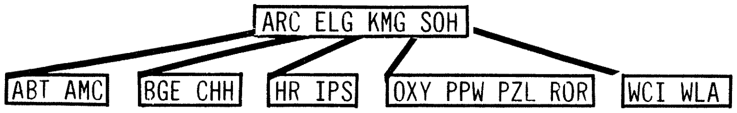
To illustrate the process of borrowing excess keys from a brother, you can delete BTU. The deletion of BTU makes its node less than half full, so look at the right brother, which contains ELG, HR and IPS. That node can spare 1 key, ELG, so you roll it up to the position where CHH is, and roll CHH down to restore the node where the deletion took place to half full:



Notice now that the tree is as small as it can get with three levels. The deletion of any key at this point will cause the tree to go to two levels. Let's delete JR just for the fun of it.

Though JR is in the root node, the root node is also a branch node. So the first thing the program does is replace JR by its successor, KMG. When the program goes down to delete KMG from its original position, its node becomes less than half full. A merge with its father key and right brother creates a node containing OXY, PPW, PZL and ROR.

That leaves the father node with only 1 key, SOH. Since it is less than half full, it merges with its parent key, now KMG, and its left brother node, containing ARC and ELG. That merge removes KMG from the root node, so the root is now empty. The resulting node, containing ARC, ELG, KMG and SOH, becomes the root and the tree is pruned to two levels. The result of all these manipulations is the tree shown below:

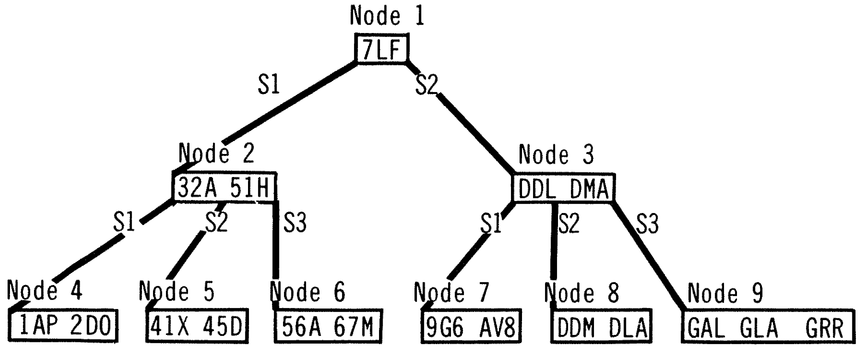


If you want to continue, subsequent deletions following the same logic will eventually collapse the tree down to one level. All the time you've been following procedures that preserve the integrity of the B-tree. Additions and deletions may occur in any order, but the tree retains its properties so you can read the keys sequentially or search the tree at any time you wish.

Reading a B-Tree

There are many occasions for reading the contents of a B-tree in ascending or descending sequence. The most common is when you want to make a printout that is sequenced by the keys. In that case you start with the first key and print each key that follows along with information from the file that is being indexed. Another situation is when you display information on the video about a record accessed by a key. It's often handy to be able to display the previous or next account with a simple keystroke. This capability is also helpful when an exact key is not known. You may want to type in a key that is close to the one you want, and find the desired record by reading forward or backward from that point.

The first problem in traversing a B-tree sequentially is how to find the beginning key. Consider the tree shown below that is made up of 3-digit combinations of letters and numbers. To aid in the description, each of the nodes is numbered. For every branch node the sons are labeled as S1 and S2, or S1, S2, and S3:



To find the first key, you must start at the root node, which is labeled "Node 1". Then you must follow the left pointers down to the first node that doesn't have a left son. Your path leads you to the first son of node 1, which is node 2. From there you go to the first son of node 2 which is node 4. Since node 4 doesn't have a left son, it contains the first key in the tree. That key is 1AP.

During the journey down to 1AP it is important for the program to record how it got there. That information is stored in the stack, which takes the form of two arrays: the node array and the path array. Each array has an entry for each level. Upon arrival at the first key, the stack looks like this:

		Node	Path
	Level 1	1	1
	Level 2	2	1
Current ->	Level 3	4	1

After printing or processing 1AP you can read the next key. To go to the next key, the program continues with the current node, 4, and adds 1 to the position indicator in the path array for the current level, 3. Now the stack shows:

		Node	Path
	Level 1	1	1
	Level 2	2	1
Current ->	Level 3	4	2

Based on the current status of the stack, the program gets 2DO from node 4 and prints or processes it.

When you go back to the tree to find the next key, the program adds 1 to the path count for the current level. This makes it 3, which is greater than the number of keys in the current node. The program now knows that it must go up a level to the correct position in the father node. The stack is the guide:

		Node	Path
	Level 1	1	1
Current ->	Level 2	2	1
	Level 3	4	3

Go to node 2, first position, and print or process the key at that point, 32A.

To get the next key, follow the path down to the second son of node 2. That is node 5. When you get there, the stack has been updated to show the status:

		Node	Path
	Level 1	1	1
	Level 2	2	2
Current ->	Level 3	5	1

Now you have the key 41X. From here we go to 45D, 51H, 56A, and 67M. After 67M our stack shows:

		Node	Path
	Level 1	1	1
	Level 2	2	3
Current ->	Level 3	6	3

When the program returns to node 2 on the second level, it sees that all paths have been followed, so it moves up a level to node 1 and gives 7LF as the next key.

		Node	Path
Current ->	Level 1	1	1
	Level 2	2	3
	Level 3	6	3

Now follow the path to the second son of node 1 and the first son of node 3. Notice that each time you go down a level, you reset the path indicator to 1 for the corresponding level in the stack. That gives you the first position of node 7, where you get the 9G6 key. The stack shows the status:

		Node	Path
	Level 1	1	2
	Level 2	3	1
Current ->	Level 3	7	1

You can continue getting keys with the same logic. After 9G6, you get AV8, DDL, DDM, DLA, DMA, GAL, GLA and GRR. The program determines that GRR is the last key by returning to node 3, seeing that the last path has been followed, and finally to node 1, where it verifies that the last path has been followed. Since node 1 is on level 1, the program can go no higher, so it's done.

Suppose you don't want to start with the first key, but rather, you want to start reading the tree from a given key that's somewhere in the middle. You can do this by searching for the key that represents the desired starting point, provided the stack is properly built on the way down to the key. To start reading from the key, DLA, search and find it. The search in this case sets up the stack as shown below:

		Node	Path
	Level 1	1	2
	Level 2	3	2
Current ->	Level 3	8	2

Since the stack has been prepared, you're ready to continue reading from the current key.

Reading a B-tree in Reverse

When you want to read the keys in the B-tree from largest to smallest, you can use logic that is quite similar to the process for reading forward. To find the last key in the B-tree, start at the root node and follow the right pointers down to the lowest level. The rightmost key in that node is the last key in the tree. In each node you start at the right and read the keys to the left. The stack is maintained exactly as it is in a search or a forward read so you can freely move about the tree.

In some cases, you might wish to maintain the tree in descending sequence. It's just not practical to rewrite a B-tree routine so it handles everything backwards. It's more convenient to store the tree in ascending sequence to take advantage of standard routines, and to read it backwards to achieve the same result.

The TREESAM Handler

TREESAM is the name I've given to an index sequential handler that you can implement in any BASIC program. The name comes from the fact that it provides ISAM capabilities based on a tree structure. It consists of several BASIC subroutines and the procedures you need to take advantage of them. Here are the main features:

- Everything is in BASIC to make the TREESAM routines easily understandable, easily modifiable, and readily transportable to various computer makes and models.

- A B-tree organization is used so all keys are dynamically kept in ascending sequence. Insertions and deletions are automatically handled. Maintenance, such as periodic reorganization or sorting, is not required.

- You have complete flexibility in defining the key length, up to 124 bytes long. With each key, you can store a numerical pointer that indexes records in another file. If you prefer, you can store other data related to the key in the B-tree itself.

- Other than the space required for the TREESAM subroutines and a few working variables, TREESAM is not memory dependent. If your disk drives have the capacity, you can have quick access to up to 32767 records on disk. Simple modifications to the subroutines or the way they are implemented can extend the accessing capacity indefinitely, making it ideal for hard disk users.

- The TREESAM subroutines are modular. If a program you are developing doesn't require a particular TREESAM capability, you can save memory by omitting the subroutine that provides it. The program lines are packed and optimized to use the least memory possible. All subroutines in TREESAM use line numbers that range from 59000 to 59600, so they are unlikely to interfere with other parts of your program. You can renumber them if you wish.

- The BASIC variables used in TREESAM all begin with the letter "T" to avoid conflict with other variables you are using in your program. You can change them to start with another letter if you wish. All TREESAM subroutines are self-contained. No other subroutines are required, and you are free to handle the organization of the file indexed, and any other files, as you wish.

- Search, insert, and delete times are quite good. In a typical application, a search will take from 1/2 to 2 seconds. Insertions and deletions generally take from

1/2 to 4 seconds. Search times can be improved with an optional machine-language assist modification. All times can be improved if a hard disk is used.

- TREESAM can handle unique or non-unique keys. Searches may be made on partial or approximate keys. Multiple keys can index the same record. Any number of TREESAM indices can be used in the same program. Keys can be read in ascending or descending sequence from any point in the index.

- A utility program, TREESAM/BAS is provided that creates an index, tests the various capabilities, and examines existing TREESAM indices. This program also gives you ways to automatically build TREESAM indices from existing data on disk, redefine key lengths and other parameters, and repair and rebuild damaged TREESAM files.

Disk Organization

The primary purpose of TREESAM is to provide a keyed accessing method for any random disk file. By a random disk file, I mean any file in which any record can be accessed by number, based on its position in the file. In most cases those records will be of uniform length. It is possible, however, to use TREESAM to keep track of byte positions, by key, in a file that uses non-uniform record lengths. To keep the discussions clear we will call the random file that is being indexed a "master" file.

TREESAM creates and maintains a file that operates as an index. We will call it an index file. In a searching situation, your program gives TREESAM a search key, and it returns a record number (or "pointer") if the search key is in its index. Then it is up to your program to use that record number to get and print or process that record from a separate file that is being indexed. If you are adding a record to your file, you give TREESAM the key that you want to use to access the new record and the record number that locates it. TREESAM will insert the new key at the proper position in ascending sequence and it will remember the record number for you.

As you can see, most TREESAM applications will use two files: the index file and the master file. In some cases more than one index will reference the same master file. In other cases you may have more than one master file, each being referenced by one or more index files. With special techniques and modifications that we will discuss, you can store the index file and master file within one physical file to save memory, but for most discussions, assume they are separate.

It will be up to your program to open and close the index file and the other files that are to be used. The detailed procedures will be discussed later. After your index is open, certain subroutines are called that initialize and field the TREESAM file buffer for use.

The first record of the TREESAM index is used to keep track of the parameters used to define it and the current status of the index. This record is called the statistics record. The organization of the statistics record is shown below. For each statistic the byte position in record 1 is shown, followed by the string variable name used to field it and the integer variable name used within the TREESAM subroutines. Finally, each is described:

Bytes	Name	Description
1-2	TN$,TN%	Half-node capacity. If the TREESAM index holds 3Ø keys per node this number is 15.

3-4	TK$,TK%	Length of each key in the index. If you want to allow a maximum key length of 6 this number is 6.
5-6	TD$,TD%	Length of the data field associated with each key in this index. Usually this number is 2 to allow for a 2-byte string to store a master file record number in MKI$ format.
7-8	TI$,TI%	Length of each item in the index. This is the sum of TK% + TD% + 2. (The 2 extra bytes account for the right son pointer stored with each key in a node.)
9-1Ø	TZ$,TZ%	Options indicator. This is a number obtained by starting with Ø. Add 1 if the data field with each key is a pointer in MKI$ format. Add 2 if non-unique keys are permitted.
11-12	TR$,TR%	Current record number of the root node.
13-14	T1$,T1%	Record number of the next unused node at the end of the index.
15-16	T2$,T2%	Record number of last node that was deleted.
17-18	T3$,T3%	Number of nodes active in the index.
19-2Ø	T4$,T4%	Number of keys active in the index.

The nodes of the B-tree are stored beginning with the second physical record of the index file. There is no "node 1" because that position stores the statistics record. The first node is "node 2". All nodes in an index have the same fielding which only needs to be done once after the index is opened. A typical node having a key length of 6 and a data length of 2 is diagrammed below:

```
     ---Item 1--- ---Item 2--- ---Item 3---     -Item TN*2 -
M PP KKKKKK DD PP KKKKKK DD PP KKKKKK DD PP ... KKKKKK DD PP

<------------------ Up to 256 bytes ------------------------>
```

The first byte in a node, shown here as M, stores a number that tells how many keys are active currently. This number may range from 0 to TN% * 2. (Again, TN% is the half-node capacity defined in our statistics record. TN% * 2 is the maximum number of items that can be stored in a 256-byte node.) In the TREESAM subroutines, this byte is fielded as TM$. ASC(TM$) gives the number that it stores. TREESAM makes no attempt to erase keys that are no longer active from a node. Instead, it stores all active keys in the leftmost fields and changes TM$ to indicate how many are active.

If ASC(TM$) is 0, the node has been deleted. In that case, the next field, shown as PP, is a 2-byte number in MKI$ format that is a link to the prior node, if any that was deleted. If no prior node was deleted, this link is recorded as 0. If this node is the last deleted, it is referenced by number in the statistics record as T2$ and in memory as T2%. The purpose of this method is to allow dynamic recovery of deleted nodes when new additions are made. The logic is the same as discussed for the labeled random disk file handler.

The 2-byte field that starts at the second byte of the node, labeled in the diagram as PP, is the left son pointer for the node. It stores a 2-byte number in MKI$ format, fielded as TP$(0).

The area to the right of the first 3 bytes in a node stores all the items. An item is defined as a key field plus a data field plus a right son pointer. These are shown in the diagram as KKKKKK DD PP. The half-node capacity parameter, given by TN%, when multiplied by 2, tells how many item fields there are in a node. Each item field has a length equal to the key length, given by TK%, plus the data length, given by TD%, plus 2 for the right son pointer. For speed and memory efficiency, the item length is stored as a separate variable, TI%. Items are fielded with the array variable TI$. TI$(1) is the first item, TI$(2) is the second, on up to TI$(TN%*2), which is the last. TI$(ASC(TM$)) is the last active item field in a given node.

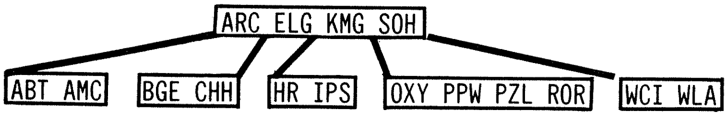
A key field exists within each item field. Each key field has a uniform length, given by TK%. Keys are fielded with the array variable TK$. TK$(1) is the first key in a node, TK$(2) is the second, on up to TK$(TN%*2) which is the last key position. TK$(ASC(TM$)) is the last active key in a given node. Keys in any node are always in ascending sequence from left to right.

Each item field also contains a data field. Each data field has a uniform length, given by TD%. Most often the data field will be 2 bytes long so it can store a 2-byte pointer. If you want to store nothing but keys in the TREESAM file, the data field may have a length of 0. If you want to store information related to a each key in the B-tree itself the data field may be any uniform length between 1 and about 120 bytes.

Data fields use the array variable TD$. TD$(1) is the first data field. TD$(TN%*2) is the last. There is a 1-to-1 correspondence between each item, key, and data field.

Finally, each item field contains a right son pointer. They are shown as PP in the diagram. Each right son pointer is a 2-byte field that stores a number from 0 to 32767 in MKI$ format. If the right son pointers are all 0 in any given node, the node is a leaf node. The right son pointers use the TP$ array. TP$(1) is the first key's right son pointer, TP$(2) is the second, and TP$(TN%*2) is the last. TP$(ASC(TM$)) is the last active right son pointer in the node. It can be considered the right son pointer for the entire node. Any key position in the node can be considered to have a right son pointer and a left son pointer. Given a key TK$(TX%), TP$(TX%) is that key's right son pointer. TP$(TX%-1) is that key's left son pointer. The right son pointer for one key is the left son pointer for the next key in the node.

To show how these concepts fit together for an actual tree, consider the B-tree from our discussion of deletions on page 253.



That tree on disk might be organized as shown below:

```
   Node 2,  2 Items:  (Ø) ABT  (Ø) AMC  (Ø)
   Node 3,  2 Items:  (Ø) HR   (Ø) IPS  (Ø)
** Node 4,  4 Items:  (2) ARC  (11)ELG  (3) KMG  (7) SOH  (8)
 * Node 5,  Ø Items:  (6)
 * Node 6,  Ø Items:  (Ø)
   Node 7,  4 Items:  (Ø) OXY  (Ø) PPW  (Ø) PZL  (Ø) ROR  (Ø)
   Node 8,  2 Items:  (Ø) WCI  (Ø) WLA  (Ø)
 * Node 9,  Ø Items:  (5)
 * Node 1Ø, Ø Items:  (9)
   Node 11, 2 Items:  (Ø) BGE  (Ø) CHH  (Ø)

 * Nodes 5, 6, 9, and 1Ø are deleted.
** Note 4 is the current root node.
```

The statistics for the tree are contained in record 1 of the index. They are shown below:

Field	Content	Description
TN$	2	Half-node capacity.
TK$	4	Maximum Length of each key in the index.
TD$	Ø	Length of the data field for each key.
TI$	6	Length of each item in the index.
TZ$	Ø	Options indicator.
TR$	4	Current record number of the root node.
T1$	12	Record number of the next unused node at the end of the index.
T2$	1Ø	Record number of last node that was deleted.
T3$	6	Number of nodes active in the index.
T4$	16	Number of keys active in the index.

The example shows a tree after many insert and delete operations have been performed. As you can see, TREESAM lets the nodes fall where they may. It doesn't, for example, try to keep the root node in the first position. Instead it updates the TR$ statistic to reflect the current position of the root. From the root and (if applicable) any other branch nodes, it can follow the pointers to the other nodes, wherever they happen to be.

Also notice the way that TREESAM handles deleted nodes. Should another node be needed for an insertion, the T2$ statistic is examined. If it is non-zero, the node indicated is re-used. The next node to be re-used in the example is node 10. After that, assuming there are no intervening node deletions, nodes 9, 5 and 6 will be recovered. When the internally-deleted nodes are exhausted, TREESAM will allocate new nodes by extending the index, using the T1$ statistic to mark the next node available at the end of the file.

To simplify things, this illustration did not show a data field with each key. Normally you would also show a number to the right of each key that points to the corresponding record in the related master file. In this case the TD$ statistic was set to zero. Also, to make the example easy to show, the number of keys each node could hold was limited to 4. With an item length of 6 you could have stored up to 42 keys in each 256-byte node.

TREESAM Subroutine Organization

The TREESAM subroutines occupy lines 59000 through 59600. They are outlined below by group. Subroutines from the initialization group are used just after you open a TREESAM index. In an application where you have multiple indices in the same program, certain of these will be also called during a session as you switch from one index to another.

Initialization Subroutines

59ØØØ	Get statistics and load them into integer variables.
59ØØ5	Field and get statistics record.
59Ø1Ø	Dimension Arrays and field node buffer.
59Ø11	Field node buffer.

The subroutines in the "common" group are those used by two or more subroutines in the other groups. They are always required in a TREESAM implementation:

Common Subroutines

```
59Ø2Ø  Get information from current level of stack and get
       the current node indicated.
59Ø3Ø  Go up one level, then perform 59Ø2Ø.
59Ø4Ø  Load information into current level of the stack.
59Ø5Ø  Write updated statistics to disk.
59Ø6Ø  Get current node if it is not presently in buffer.
```

The subroutines in the "major" group are the ones your application program will most often call. Depending on the functions of a given program, some of these may not be required:

Major Subroutines

```
591ØØ  Search for key, KY$.
592ØØ  Add key, KY$. (Must follow a search.)
593ØØ  Read next key. (Or read first if level is Ø.)
594ØØ  Read previous key. (Or read last if level is Ø.)
595ØØ  Delete current key. (Must follow a search or read.)
596ØØ  Add non-unique key, KY$.
```

The entire set of TREESAM subroutines requires about 3900 bytes of program memory. In some programs you might wish to make all functions available. In others, you can save memory by including only those that are needed.

A program that just allows inquiries based on TREESAM keys only requires the initialization subroutines, 59000 – 59011, the common subroutines 59040 and 59060, and the search subroutine, 59100 – 59160. The total program memory required in this case is 775 bytes. If you want a read-next or read-previous capability, you'll need to add subroutines 59020 and 59030 which consume 58 bytes. The read-next subroutine, 59300 – 59320 is 170 bytes. The read-previous subroutine is 164 bytes.

If a program is to allow additions into the B-tree, all the initialization and common subroutines, 59000 – 59060, are required, as is the search subroutine, 59100 – 59160, and the add subroutine, 59200 – 59250. The total usage of program memory comes to 1821 bytes. If non-unique keys are to be added in the tree, subroutines 59600 and 59300 – 59320 must be included, making the total 2017 bytes.

Programs that print lists or reports may need only the capability of reading the TREESAM index from beginning to end. You'll need the initialization subroutines, 59000 – 59011, the common subroutines, 59020, 59030, 59040 and 59060, and the read-next subroutine, 59300 – 59320. The total is 678 bytes.

To include a deletion capability in your program you must have the all the initialization routines, all the common routines, the search subroutine, the read-next and read-previous subroutines, and the deletion subroutine that occupies lines 59500 – 59568. The TREESAM subroutines for a program that allows you to do everything but add keys require a total of 2966 bytes. This includes the deletion subroutine itself, which is 1719 bytes.

Variables Used in TREESAM

All variable names in TREESAM begin with the letter "T". There are two exceptions. KY$ is a string variable that passes your key to TREESAM during a search or add. DA$ is a string variable that holds the data that you want stored with your key. It is used when you are adding to the tree.

Every variable in TREESAM is of the integer or string type. All string variables are marked by the "$" indicator within the subroutines. All the others are integers and the "%" indicator is omitted to conserve memory. Because of this, your program must declare that all variables beginning with "T" are integers. You can do this with a "DEFINT T" statement somewhere near the beginning of your program. If this causes a conflict with your scheme of things, you can do the DEFINT just before each call to TREESAM. A "DEFINT A-Z" statement will do the same job.

Simple Variables

T1% Next node at the end of the index. This is the record number where the next node will be allocated if there have been no internal deletions.

T1$ Used to field the buffer for T1%.

T2% Record number of the last internally deleted node.

T2$ Used to field the buffer for T2%.

T3% Number of nodes active.

T3$ Used to field the buffer for T3%.

T4% Number of keys active in the tree.

T4$ Used to field the buffer for T4%.

TA$ Used internally to build and hold an item to be added.

TC% Record number of the current node.

TD% Length of the data field stored with each key.

TD$ Used to field the buffer for TD%.

TE$ During an add, holds the emerging item when a node is split and the middle item is passed up a level. Used as working storage during a delete.

TF% Specifies the file buffer used for the TREESAM index.

TH% During the binary search of a node indicates the half point among the keys that are being compared. Used as working storage during a delete.

TI% Item length. Key length + data length + 2.

TI$ Used to field the buffer for TI%.

TK% Key length.

TK$ Used to field the buffer for TK%.

TL% Current level in the tree and the stack.

TM$ A 1-byte field whose ASCII value specifies the number of keys active in the current node.

TN% Half-node capacity. Items per node / 2.

TN$ Used to field the buffer for TN%.

TQ% Returns status after a search or read. Equals -1 if the operation was successful. Equals 0 if not found or end of file.

TR% Record number of the root node.

TR$ Used to field the buffer for TR%.

TS$ Temporary working storage. Most often used to hold items from a node during a split or merge operation.

TW$ Temporary working storage.

TW% Temporary working stoarge. During a normal search, TW is 0. During a search in preparation for a non-unique add, TW is -1.

TX% Used to subscript arrays to identify a given item or key in a node. In a search or read, identifies the item that was found or read.

TY% Used in FOR NEXT loops as a counter, most often when shifting items within a node.

TZ% Options indicator. Bits 1 and 2 are used to specify if the data fields contain pointers and if non-unique keys are allowed.

Array Variables

TI$() Used to field the buffer for items. Each item in a node contains space for a key, a data field, and a right son pointer.

TK$() Used to field the buffer for the keys in a node.

TD$() Used to field the buffer for the data fields in a node.

TP$() Used to field the buffer for the internode pointers.

TS%(,) The stack array. The first dimension is the level. For any level, TL, TS%(TL,0) is the node. TS%(TL,1) is the current position (or path taken) in that node.

Unless you make modifications to TREESAM, you'll only be concerned with a few of the variables listed above. Many of them are used within the execution of a TREESAM subroutine, and their content is not important before or after your GOSUB command. Any working storage strings that TREESAM uses are automatically nulled upon return so that string storage space in memory is available to your main program.

As you get more familiar with the internal workings of TREESAM, you may want to use certain of these variables elsewhere in your program or to make substitutions so that you get the most mileage (and memory efficiency) out of each variable. For now, it will suffice to say that you are completely free to use TA$, TE$, TH%, TQ%, TS$, TW$, TW% and TY% for temporary storage elsewhere in your program, but remember that calls to certain TREESAM subroutines will alter their contents.

TREESAM Handler Subroutines

Get and Load Statistics

```
59ØØØ GOSUB59ØØ5:TN=CVI(TN$):TK=CVI(TK$):TD=CVI(TD$):TI=CVI(TI$)
:TZ=CVI(TZ$):TR=CVI(TR$):T1=CVI(T1$):T2=CVI(T2$):T3=CVI(T3$):T4=
CVI(T4$):RETURN
```

Field and Get Statistics Record

```
59ØØ5 FIELDTF,2ASTN$,2ASTK$,2ASTD$,2ASTI$,2ASTZ$,2AS TR$,2AST1$,
2AST2$,2AST3$,2AST4$:TC=1:GOSUB59Ø6Ø:RETURN
```

Dimension Arrays and Field Node Buffer

```
59Ø1Ø DIMTK$(TN*2),TP$(TN*2),TS(5,1),TI$(TN*2),TD$(TN*2)
59Ø11 FIELDTF,1ASTM$,2ASTP$(Ø):FORTY=1TOTN*2:FIELDTF,3+TI*(TY-1)
ASTW$,(TK)ASTK$(TY),(TD)ASTD$(TY),2ASTP$(TY):FIELDTF,3+TI*(TY-1)
ASTW$,(TI)ASTI$(TY):NEXT:RETURN
```

Get Data From Stack and Get Node Indicated

```
59Ø2Ø TC=TS(TL,Ø):TX=TS(TL,1):GOSUB59Ø6Ø:RETURN
```

Go Up One Level

```
59Ø3Ø TL=TL-1:GOSUB59Ø2Ø:RETURN
```

Load Information into Current Level of Stack

```
59Ø4Ø TS(TL,Ø)=TC:TS(TL,1)=TX:RETURN
```

Write Updated Statistics to Disk

```
59Ø5Ø TC=1:GETTF,TC:LSETTR$=MKI$(TR):LSETT1$=MKI$(T1):LSETT2$=MK
I$(T2):LSETT3$=MKI$(T3):LSETT4$=MKI$(T4):PUTTF,TC:RETURN
```

Get Current Node if Not Present in Buffer

```
59Ø6Ø IFLOC(TF)=TCTHENRETURNELSEGETTF,TC:RETURN
```

Search For Key, KY$

```
591ØØ TW=Ø
5911Ø TQ=Ø:TL=1:TC=TR:IFLEN(KY$)<>TKTHENKY$=LEFT$(KY$+STRING$(TK
," "),TK)
5912Ø GOSUB59Ø6Ø:TX=1:TY=ASC(TM$)
5913Ø TH=INT((TX+TY)/2):IFKY$>TK$(TH)OR(TWANDKY$=TK$(TH))THENTX=
TH+1ELSETY=TH-1
5914Ø IFTY>=TXTHEN5913ØELSEGOSUB59Ø4Ø:IFTX<=ASC(TM$)THENIFKY$=TK
$(TX)THENTQ=-1:IF(TZAND2)=ØORCVI(TP$(TX-1))=ØTHENRETURN
5915Ø IFCVI(TP$(TX-1))>ØTHENTC=CVI(TP$(TX-1)):TL=TL+1:GOTO5912ØE
LSEIF(TZAND2)=ØTHENRETURN
5916Ø IFTQ=ØTHENRETURNELSE5931Ø
```

Add Key and Data at Current Position

```
592ØØ TA$=LEFT$(KY$+STRING$(TK," "),TK)+LEFT$(DA$+STRING$(TD," "
),TD)+MKI$(Ø):TW=Ø
5921Ø IFASC(TM$)<TN*2THENLSETTM$=CHR$(ASC(TM$)+1):TY=ASC(TM$):GO
SUB5922Ø:LSETTP$(TX-1)=MKI$(TW):PUTTF,TC:T4=T4+1:TA$="":TS$="":T
E$="":RETURN
59211 IFTX>TN+1THEN59213ELSEIFTX=TN+1THENTE$=TA$ELSETE$=TI$(TN):
TY=TN:GOSUB5922Ø
```

```
59212 LSETTP$(TX-1)=MKI$(TW):LSETTM$=CHR$(TN):FIELDTF,3+TN*(TI)A
STW$,TN*(TI)ASTW$:TS$=TW$:PUTTF,TC:TW=TC:GOSUB5923Ø:GOSUB5925Ø:G
OSUB5924Ø:GOTO59215
59213 FIELDTF,1ASTW$,2+TN*(TI)ASTW$:TS$=TW$:TX=TX-TN:TE$=TI$(TN+
1):FORTY=1TOTX-2:LSETTI$(TY)=TI$(TY+TN+1):NEXT:LSETTI$(TX-1)=TA$
:IFTX>TNTHEN59214ELSEFORTY=TXTOTN:LSETTI$(TY)=TI$(TY+TN):NEXT
59214 GOSUB5924Ø:LSETTP$(TX-2)=MKI$(TW):PUTTF,TC:GOSUB5923Ø:FIEL
DTF,1ASTW$,LEN(TS$)ASTW$:LSETTW$=TS$:LSETTM$=CHR$(TN):TW=TC
59215 PUTTF,TC:TA$=TE$:TL=TL-1:IFTL=ØTHENGOSUB5923Ø:TX=1:TR=TC:L
SETTP$(Ø)=MKI$(TW):GOTO5921ØELSEGOSUB59Ø2Ø:GOTO5921Ø
5922Ø FORTY=TYTOTX+1STEP-1:LSETTI$(TY)=TI$(TY-1):NEXT:LSETTI$(TX
)=TA$:RETURN
5923Ø IFT2>ØTHENTC=T2:GOSUB59Ø6Ø:T2=CVI(TP$(Ø))ELSETC=T1:GOSUB59
Ø6Ø:T1=T1+1
59231 T3=T3+1:LSETTM$=CHR$(Ø):RETURN
5924Ø LSETTM$=CHR$(TN):LSETTP$(Ø)=RIGHT$(TE$,2):MID$(TE$,LEN(TE$
)-1,2)=MKI$(TC):RETURN
5925Ø FIELDTF,3ASTW$,LEN(TS$)ASTW$:LSETTW$=TS$:RETURN
```

Read Next Key (or Read First)

```
593ØØ IFTL=ØTHENTC=TR:TL=1:TX=1ELSETX=TX+1
5931Ø GOSUB59Ø6Ø:IFCVI(TP$(TX-1))=ØTHEN5932ØELSEGOSUB59Ø4Ø:TC=CV
I(TP$(TX-1)):TL=TL+1:TX=1:GOTO5931Ø
5932Ø IFTX<=ASC(TM$)THENTQ=-1:RETURNELSEIFTL=1THENTL=Ø:TQ=Ø:RETU
RNELSEGOSUB59Ø3Ø:GOTO5932Ø
```

Read Previous Key (or Read Last)

```
594ØØ IFTL=ØTHENTC=TRELSE5942Ø
5941Ø TL=TL+1:GOSUB59Ø6Ø:TX=ASC(TM$)+1
5942Ø GOSUB59Ø4Ø:IFCVI(TP$(TX-1))>ØTHENTC=CVI(TP$(TX-1)):GOTO594
1Ø
5943Ø IFTX>1THENTX=TX-1:TQ=-1:RETURNELSEIFTL=1THENTQ=Ø:TL=Ø:RETU
RNELSEGOSUB59Ø3Ø:GOTO5943Ø
```

Delete Key and Data at Current Position

```
595ØØ GOSUB59Ø4Ø:IFCVI(TP$(TX))>ØTHEN5951ØELSEGOSUB59564:IFTX-1<
>ASC(TM$)THENGOSUB5956Ø
595Ø1 PUTTF,TC:IF(TC=TR)OR(ASC(TM$)>=TN)THEN595Ø3
595Ø2 GOSUB5952Ø:IFTQTHEN595Ø2
595Ø3 T4=T4-1:RETURN
5951Ø GOSUB593ØØ:TA$=TI$(TX):GOSUB594ØØ:GOSUB59568:PUTTF,TC:GOSU
B593ØØ:GOTO595ØØ
5952Ø TQ=-1:GOSUB59Ø3Ø:IFASC(TM$)=TX-1THEN5954ØELSETC=CVI(TP$(TX
)):GOSUB59566:TE$=TP$(Ø)
59521 IFTH<=ØTHEN5953ØELSEFIELDTF,3ASTW$,TI*(TH-1)ASTW$:TS$=TW$:
TA$=TI$(TH):LSETTP$(Ø)=TP$(TH):LSETTM$=CHR$(ASC(TM$)-TH):IFASC(T
M$)>ØTHENFORTY=1TOASC(TM$):LSETTI$(TY)=TI$(TY+TH):NEXT
59522 PUTTF,TC:GOSUB59Ø2Ø:TS$=TI$(TX)+TS$:GOSUB59568:GOSUB59562:
FIELDTF,3+TI*(TN-1)ASTW$,LEN(TS$)ASTW$:LSETTW$=TS$:LSETTP$(TN)=T
E$:GOTO59544
```

```
5953Ø FIELDTF,3ASTW$,TN*(TI)ASTW$:TS$=TW$:TW$=TP$(Ø):LSETTM$=CHR
$(Ø):LSETTP$(Ø)=MKI$(T2):T2=TC:T3=T3-1:PUTTF,TC:GOSUB59Ø2Ø:LSETT
P$(TX)=TW$:TS$=TI$(TX)+TS$:GOSUB59564
59531 IFTC=TRANDASC(TM$)=ØTHENTR=TS(TL+1,Ø):LSETTP$(Ø)=MKI$(T2):
T2=TC:T3=T3-1:TQ=Ø:GOTO59534
59532 IF(ASC(TM$)>=TN)OR(TC=TR)THENTQ=Ø
59533 IFASC(TM$)>=TXTHENGOSUB5956Ø
59534 GOSUB59562:FIELDTF,3+TI*(TN-1)ASTW$,LEN(TS$)ASTW$:GOTO5955
3
5954Ø TC=CVI(TP$(TX-2)):GOSUB59566:IFTH<=ØTHEN5955Ø
59541 LSETTM$=CHR$(ASC(TM$)-TH):TA$=TI$(ASC(TM$)+1):FIELDTF,3+TI
*(ASC(TM$)+1)ASTW$,TI*(TH-1)ASTW$:TS$=TW$:TE$=TP$(ASC(TM$)+1):PU
TTF,TC:GOSUB59Ø2Ø
59542 TS$=TS$+TI$(TX-1):LSETTI$(TX-1)=TA$:LSETTP$(TX-1)=MKI$(TS(
TL+1,Ø)):GOSUB59562:IFTN>1THENFORTY=TN-1TO1STEP-1:LSETTI$(TY+TH)
=TI$(TY):NEXT
59543 GOSUB5925Ø:LSETTP$(TH)=TP$(Ø):LSETTP$(Ø)=TE$
59544 LSETTM$=CHR$(TN-1+TH):PUTTF,TC:TQ=Ø:RETURN
5955Ø FIELDTF,1ASTW$,2+ASC(TM$)*(TI)ASTW$:TS$=TW$:LSETTM$=CHR$(Ø
):LSETTP$(Ø)=MKI$(T2):T2=TC:T3=T3-1:PUTTF,TC:GOSUB59Ø2Ø:TS$=TS$+
LEFT$(TI$(TX-1),TI-2):LSETTP$(TX-2)=MKI$(TS(TL+1,Ø)):GOSUB59564
59551 TQ=Ø:IFTC=TRANDASC(TM$)=ØTHENTR=TS(TL+1,Ø):LSETTP$(Ø)=MKI$
(T2):T2=TC:T3=T3-1ELSEIF(TC<>TR)AND(ASC(TM$)<TN)THENTQ=-1
59552 GOSUB59562:FIELDTF,3ASTW$,TI*ASC(TM$)ASTW$:TS$=TS$+TP$(Ø)+
TW$:FIELDTF,1ASTW$,LEN(TS$)ASTW$
59553 LSETTW$=TS$:LSETTM$=CHR$(TN*2):PUTTF,TC:IFTQTHENGOSUB59Ø3Ø
:RETURNELSERETURN
5956Ø FORTY=TXTOASC(TM$):LSETTI$(TY)=TI$(TY+1):NEXT:RETURN
59562 PUTTF,TC:TL=TL+1:GOSUB59Ø2Ø:RETURN
59564 LSETTM$=CHR$(ASC(TM$)-1):RETURN
59566 GOSUB59Ø6Ø:TH=INT((ASC(TM$)-TN+1)/2):RETURN
59568 TW$=TP$(TX):LSETTI$(TX)=TA$:LSETTP$(TX)=TW$:RETURN
```

Add Non-Unique Key

```
596ØØ TW=-1:GOSUB5911Ø:GOSUB592ØØ:RETURN
```

TREESAM Handler — Line Comments

59000 (Get and Load Statistics)

:Call 59005 to get statistics record and field it.
:Load half-node parameter, TN, from buffer.
:Load key length parameter, TK, from buffer.
:Load data length parameter, TD, from buffer.
:Load item length parameter, TI, from buffer.
:Load options indicator, TZ, from buffer.
:Get root node number, TR, from buffer.
:Get number of next node at end of tree, T1, from buffer.
:Get number of last deleted node, T2, from buffer.
:Get count of active nodes, T3, from buffer.
:Get count of active keys, T4, from buffer.
:Return to main program.

59005 (Field and get statistics record)

:Point TN$, TK$, TD$, TI$, TZ$, TR$, T1$, T2$, T3$, and T4$ to the buffer for the file number specified by TF.
:The statistics record is record 1 in the file.
:Call 59060 to get it from disk.
:Return (usually back to subroutine 59000.)

59010 (Dimension arrays and field node buffer)

:Dimension the arrays that will be required, and do so in order of usage frequency for speed. Each of the arrays used to field the node are dimensioned at 2 times the half node capacity, TN. The stack array, TS, is dimensioned for up to 5 levels. (Change if necessary.)

59011

Field the buffer, 1 byte for the key count as TM$, 2 bytes for the left son pointer, TP$(0).
:Now, for each item in a maximum-sized node, using TY as a counter,
:use TW$ as a dummy field to align, field TK$(TY) with the standard key length, TD$(TY) with the standard data length, and the right son pointer for the item, TP$(TY) with a length of 2, so integer internode pointers can be held.
:Use TW$ as a dummy field again to realign to the same position, and field TI$(TY) so that it overlays the fields for the corresponding key field, data field and right son pointer field. (The parentheses around TK, TD and TI are there because some versions of Disk BASIC get confused without them.)
:Repeat for the next position in the node buffer.
:Return to the main program.

59020 (Get data from stack and get node indicated)

:Get current node number, TC, from current level of stack.
:Get position in current node, TX from current stack level.
:Call 59060 to get the current node from disk.
:Return to the calling subroutine: 59030, 59215, 59522,
59530, 59541, 59550 or 59562.

59030 (Go up one level)

:Subtract 1 from the level number, TL.
:Call 59020 to get node and position on that level.
:Return to 59320, 59430, 59520 or 59553

59040 (Load information into current level of stack)

:Record current node number in current level of stack.
:Record position within node in current level of stack.
:Return to 59140, 59310, 59420, or 59500.

59050 (Write Updated Statistics to Disk)

:Statistics record is record 1 of the file.

:Get record 1 (to preserve any contents it might have beyond the first 20 bytes.)
:Load the root node number, RN, into the buffer.
:Load T1 statistic into the buffer.
:Load T2 statistic into the buffer.
:Load T3 statistic into the buffer.
:Load T4 statistic into the buffer.
:Write contents of the memory buffer to disk.
:Return to the main program.

59060 (Get current node if not present)

:If the last record gotten for this file is equal to the record you want now, TC, then no new GET is necessary, so
:Return to the calling subroutine,
:Otherwise, get the record from disk and put its contents in the disk buffer specified by TF,
:and return to the calling subroutine.

59100 (Search for the key, KY$)

:This line, 59100, is the normal entry point for a search, so load TW with 0 to indicate a normal search.

59110 (This line, 59110, is the entry point from 59600 for a search before non-unique add. In that case TW is -1.)

:Preload TQ with 0 to indicate a not-found condition.
:All searches start on level 1, so load level, TL, with 1.
:All searches start at the root node, so current node equals root node.
:If the length of the key supplied, KY$, is not equal to the standard key-length for this tree, TK, then pad or truncate KY$ to make it TK bytes long.

59120

Call 59060 to get the current node, specified by TC.
Use TX as a left limit in a binary search. TX starts out as 1, to point to the left-most key in the node.
Use TY as a right limit in a binary search. TY starts out as the position of the rightmost active key in this node.

59130

Compute TH as the halfway point between TX and TY. If the search key is greater than the key at the halfway point or, if this is a search in preparation for a non-unique add and the search key is equal to the key at the halfway point,
then confine the search to everything in the node that is to the right of the halfway point,
otherwise, confine the search to everything in the node that is to the left of the halfway point.

59140

If after confining the left and right limits, the left limit is still greater than or equal to

the right, then repeat the process from line 59130.
Otherwise, you're done searching this node. TX points to the first key in the node that's greater or equal to the search key, and, in the case that the search key is greater than any key in the node, TX points to the position in the node beyond the last active key. We call 59040 to make a note of the current node and position in the stack.
:If at least one key in the node is greater than or equal to the search key, and if it is, in fact, equal, then load the found indicator, TQ, with–1. You've found a match for the key! If this is a unique-key tree, or if it's a non-unique key tree and you're in a leaf node,
:then the search is over. Return to the main program.

59150 (If you haven't found a match yet, or if this is a non-unique key tree and you need to check the lower levels, you have arrived here.)

:If the left-son pointer for the key at the position found during the binary search is greater than zero, then you'll want to get the node that's the current key's left son. Load the current node number from that pointer.
:Since you'll be going down a level, add 1 to the level number, and go to 59120 to do the binary search on the new current node. Otherwise, in the case there's no left son pointer, you're on a leaf node. So if this is a unique-key tree, then return to the main program (with TQ still zero to indicate not-found).

59160 (If the program gets to this point, you must be in a non-unique key tree, and you must be at the lowest level.)

If the found indicator, TQ, is 0, it means that you didn't find the key on a higher level, so we return, with a not-found condition. Otherwise, we did find a match at a higher level. A read-next will get node that contains it back to us, so we jump into the read-next subroutine at line 59310.

59200 (Add key, KY$, at current position in tree)

Build the item to be added, key field plus data field plus, (for now), a right son pointer of zero. Hold it as TA$. Use TW to store a left son pointer. For now, load it with zero.

59210

If current node is not full yet, then add 1 to the count of keys in the node, and put the new count in the buffer,
:make TY equal to the new rightmost position in the node,
:call 59220 to insert the new item by moving all items between TX, (the position where the new item will be added), and TY, one position to the right,
:replace the left-son pointer for the new item with the contents of TW,
:write the current buffer to disk,
:add 1 to the count of keys active in the entire tree,
:clear TA$ to free up the memory it used,
:clear TS$ to free up the memory it used,
:clear TE$ to free up the memory it used,
:and finally, return to the main program.

59211 (At this point, you've found that the current node is full, so you can't add the new item in TA$ without splitting.)

:If the position, TX, where you want to add the new item is past what would be the middle of the node if its capacity were 1 greater, then go to 59213. (New item will go in right half.)
:Otherwise, if the position, TX, is what will be the middle then the new item will be the one to be passed up to the next level when you split. The new item is the "emerging" item, so store it in TE$.
:Otherwise, the new item will go in the left half of the current node. The emerging item to be passed up to the next level is the one just to the left of the midpoint. Store it as TE$ so you can handle it later.
:Make TY equal to the rightmost position in the left half
of the node.
:Call 59220 to insert the item stored in TA$ at the proper position in the left half of the node by moving other items right if necessary.

59212

Replace the left son pointer of the new item with TW.
:Mark the current node as half-full.
:Field the node so TW$ contains the right half.
:Store the right half in string storage memory as TS$.
:Write the current node to disk.
:Load TW with the record number of the current node. TW will be used later to provide the left son pointer for the emerging item when it's inserted in the father node.
:Call 59230 to allocate and get a new node to store the right half of the old node.
:Call 59250 to put the right half of the old node, now held in TS$, into the proper position in the file buffer.
:Call 59240. It marks the new node as half full, loads the new node's left son pointer with the emerging item's right son pointer, and makes the emerging item's right son pointer point to the new node.
:The logic from this point is the same as the logic for when you add a new item to the right half, so continue at line 59215.

59213 (You reach this point if the item to be added should go in the right half of the node to be split.)

:Field the buffer so that TW$ contains the left half of the node.
:Store the left half, including the left son pointer, in string storage memory as TS$.
:Compute what the position of the new item will be when everything is moved to the left side of the buffer. Hold it as TX.
:Store the item to be passed up one level, now located at the leftmost postion of the right half, in TE$.
:For each position in the left half of the node that is to the left of where the new item will go,
:copy the corresponding item that is now in the right half of the node. (Allow 1 extra position in the offset to account for the item that's being passed up.)
:Repeat until each item in the group is copied.
:Now put the new item, TA$, just to the right of the last old item copied over.

:Jump to 59214 if everything necessary is in place, Otherwise, for each position to the right of the new item,
:Move any remaining items from the right half of the node, to the left.
:Repeat until all items have been moved.

59214

Call 59240. It marks the node as half full, loads the node's left son pointer with the emerging item's right son pointer, and makes the emerging item's right son pointer point to the node.
:Record the left son pointer of the new item, as stored in TW.
:Write the node to disk.
:Call 59230 to allocate and get a new node (which will store items from the old node's left half).
:Field the new node you we can put the data now in TS$ into it.
:Put the keys transferred from the old node into the buffer.
:Mark the node as half-full.
:Use TW to store the record number of this node. It will be the left-son pointer of the emerging item.

59215 Write the node to disk.

:You will now want to insert the emerging item, stored in TE$ into the proper position within the father of the current node, so make TA$ equal to TE$.
:Subtract 1 from the level indicator to go up a level.
:If that makes the level equal to 0, then the tree has to grow by one level. Call 59230 to allocate and get the node that will be the new root,
:the position in the root node will be 1,
:the root node statistic, TR, is changed to equal the record number of the new (and current) node,
:you make its left son pointer equal to what's in TW,
:and go to 59210, where you'll store the item that has emerged in the new root.
:Otherwise, the level is not 0 so you don't have to make a new root. We call 59020 to make the father node our current node, and to load TX with the proper position,
:and you go to 59210, where you'll insert the item that has emerged as a result of the split.

59220 (Insert TA$ in a node by moving subsequent items right.)

Starting from the right and working toward the left, for each position between TX+1 and TY,
:move the item right one position.
:repeat until all have been moved.
:Put the new item in the position that's been opened up.
:Return to 59210 or 59211

59230 (Allocate and get a new record.)

If T2 is non-zero, you can recover a previously deleted node, so make the current node equal to T2, the record number of the last node deleted.
:call 59060 to get it,
:and change T2 to the number of the node deleted before this one.

:Otherwise, you have to extend the B-tree by taking the next node beyond the last you used. Its number is stored in T1. Make current node equal to T1,
:call subroutine 59060 to get it,
:and, add 1 to T1 to point to the new end of file.

59231

Add 1 to T3 to update the count of nodes active.
:Mark the new node as empty.
:Return to 59212, 59214, or 59215.

59240

Mark current node as half full.
:Replace its left pointer with the right pointer of the emerging item.
:Replace the right pointer in the emerging item with the record number of the current node.
:Return to 59212 or 59214.

59250

Field the current node so that TS$ can be copied into it.
:Put TS$ into the field.
:Return to 59212 or 59543.

59300 (Read next key, or read first.)

:If level is 0, you need to read the first key, so load current node, TC with the record number of the root,
:make the level equal to 1,
:and set the position indicator, TX, equal to 1 so you can read the first key in the root.
:Otherwise, add 1 to the current position indicator.

59310

Call 59060 to get the current node into the buffer, in case it is not present.
:If there is no left-son pointer, then jump to line 59320.
:Otherwise, call 59040 to record the number of the current node and the position into the current level of the stack.
:Make the current node equal to the left-son pointer.
:Add 1 to the current level.
:Reset position indicator to 1.
:And, jump to 59310 to follow left-son pointer down to next level.

59320

If the position in the node is less than or equal to the position of the last active key in the node,
:then load TQ with -1 to indicate that you've retrieved the next key,
:and return to the main program or 59510.

:Otherwise, you are past the last active key in the node, so if the level is 1, then change the level number to 0,
:and load TQ with –1 to indicate that you are at the end of the tree, and no more keys exist.
:Return to the main program.
:Otherwise, you are past the last active key in the node and we are not on level 1, so call 59030 to go up one level.
:and, go to 59320 to continue at the next position in the father node.

59400 (Read previous key, or read last)

:If level indicator is 0, you are to read the last key, so, load current node indicator, TC, with the number of the root node.
:Otherwise, you need to read previous key, so jump to 59420.

59410

Add 1 to level indicator to go down in the tree.
:Call 59060 to get the current node into the buffer.
:Set position indicator to point to the position just beyond the last active key in the node.

59420

Call 59040 to record the position into the current level of the stack.
:If the right-son pointer is non-zero (or in the event you're at the position just beyond the last active key in the node, if the left-son pointer of the node is non-zero),
:then make the current node indicator equal to that pointer.
:and go to 59410 to get the follow the pointer down to the next level.

59430

If you are not in the leftmost position of the node, then subtract 1 from the position.
:Load TQ with –1 to indicate you've retrieved the key.
:and return to the main program, or 59510.
:Otherwise, you are in the first position of the node, so if you are on level 1,
:then load TQ with 0 to indicate you've reached the beginning of the tree, and no previous keys exist,
:and reset the level indicator to 0,
:and return to the main program.
:Otherwise, you are in the first position of the node, and you're not on level 1, so call 59030 to go up a level to the father node.
:Go to 59430 to retrieve the key in the father node.

59500 (Delete key and data at current position.)

:Call 59040 to record the current position and node into the current level of the stack.
:If you are not in a leaf node, then jump to 59510.
:Otherwise, you are in a leaf node. Call 59564 to subtract 1 from the count of active keys in this node.

:If you're not in the last position of the node, then call 59560 to shift left all of the items that are to the right of the item in the position to be deleted.

59501

Write the modified node from the buffer to disk.
:If you're in the root node or if the node is still at least half-full, then the deletion is complete. Go to 59503. Otherwise, continue at 59502. You must underflow this node.

59502

Call the underflow subroutine, 59520.
:If TQ is –1 there's more to underflow, so repeat line 59502

59503

Otherwise, subtract 1 from count of active keys, and
:Return to the main program.

59510 (You're not in a leaf node so you must replace the item to be deleted with the next item in the tree.)

:Call the read-next subroutine, 59300.
:Store next item as TA$.
:Call the read-previous subroutine, 59400, to get back to original node and position of item to be deleted.
:Call 59568 to replace deleted item with next item in TA$.
:Write the buffer to disk.
:Call the read-next subroutine, 59300, so you can position to the item that you used to replace the item to be deleted.
:Now that you've deleted the item from the branch node, you have to go delete the item from the leaf node that you used to replace it. Go to 59500.

59520 (Underflow subroutine, for nodes less than half full.)

Use TQ as a flag. A value of –1 indicates an underflow condition still exists.
:Call 59030 to move up to the father node.
:If there's no right brother, then go to 59540 to underflow to left brother.
:Otherwise, current node equals right brother.
:Call 59566 to get right brother and to figure how many items, TH, can be moved from right brother to underfilled node.
:Store left son pointer of this node as TE$.

59521

If there aren't any excess items in this node that can be spared, then you will have to merge the two nodes. Go to 59530 to do it.
:Otherwise, field the buffer so that TW$ contains the items to be moved from this node to the underfilled node.
:Store them outside the buffer, in TS$.
:Store the item to be rolled up to the father node as TA$.

:Replace the left-son pointer.
:Change count of active items to account for the items you're taking from this node.
:Move all items that are to the right of the items you're taking back to the left side of the node.

59522

Write from buffer to disk the node from which you borrowed.
:Call 59020 to get the father node.
:You'll be rolling the parent item from the father node to the underfilled node also, so add it in front of the items that are being moved.
:Call 59568 to put the item that we rolled up from the right brother into the father node.
:Call 59562 to write the modified father node from the buffer to disk, and to get the underfilled node again.
:Field the underfilled node so the already existing items are undisturbed, and TW$ provides a place to put the items that are being moved from the father and right brother.
:Put the items being moved into the buffer.
:Change the left pointer of the middle item so it's the left son pointer of the prior right brother, (as stored in TE$ by line 59520.)
:Go to 59544 to finish up.

59530 (The right brother can't spare any items, so you have to merge the right brother, the item from the father node, and the underfilled node.)

:Field the buffer so that TW$ holds all the items.
:Store them outside the buffer in TS$.
:Now use TW$ as temporary storage for the node's left-son pointer.
:Mark the right brother node as empty.
:Use bytes 2 and 3 of the node as a pointer to the prior deleted node so that deleted nodes can be recovered later. Do it by putting the "last deleted node" statistic, T2, in TP$(0).
:Now this is the "last deleted node", so put the number of this node in T2.
:Subtract 1 from the count of active nodes, T3.
:Write this node from the buffer to disk.
:Call 59020 to get the father node again.
:Replace left son pointer of father item with TW$, the left son pointer of its right son that has now been deleted.
:Add the item to be taken from the father node onto the right side of TS$, which contains the items taken from the right brother.
:Call 59564 to subtract 1 from the number of items in the father node, because you're taking an item from it.

59531

You're in the father node. If it's the root, and if by taking the item you've made it empty, then the root, TR, becomes the underfilled node you will be filling. (Find TR by looking at the next level on the stack.)
:Use bytes 2 and 3 of the node as a pointer to the prior deleted node,
:make this node the "last deleted node", T2,
:subtract 1 from the count of active nodes, T3,

:clear the underflow flag, TQ,
:and jump to 59534 to build the previously underfilled node.

59532

You reached this point if the father node is not the root; if it is, it's not empty. If after taking the item from the father node, it's at least half full or it's the root (in which case you don't care if it is less than half full),
:then clear the underflow flag, TQ. (Otherwise, TQ remains –1, meaning that you'll need to come back later to take care of the father node which is now underfilled.)

59533

If the item you're taking from the father is anywhere in the node other than in the rightmost position,
:then call 59560 to move each subsequent item left.

59534

Write the modified father node from the buffer to disk.
:Field the buffer so that TW$ points to a place to store the items borrowed from the father the right brother.
:Go to 59553 to finish up on the underfilled node.

59540 (You've gotten here from 59520. There was no right brother for the underfilled node, so you've got to underflow to the left brother.) Current node equals left brother of underfilled node.

:Call 59566 to get left brother and compute how many items, TH, it can spare.
:If the left brother is only half full, it can't spare any, so go to 59550 to merge the left brother with the underfilled node.

59541

Subtract the number of items to be taken from left brother.
:Store the item to be rolled up to the father node as TA$.
:Field the buffer so TW$ contains the items to be moved from this node to the underfilled node.
:Store them outside the buffer, in TS$.
:Hold the left son pointer of the items being moved in TE$.
:Write the modified node from which you borrowed, from the buffer to disk.
:Call 59020 to get the father node again.

59542

Append the item you're moving from the father node onto the items you've taken from the left brother, in TS$.
:Replace it with the item in TA$ that you got from the left brother.
:Make the left son pointer of the father node item equal to the record number of the underfilled node you're filling.
:Call 59562 to write the father from the buffer to disk, and to get the underfilled node again.

:Move all items in the underfilled node right to make space for the items being moved from the left brother and father node.

59543

Call 59250 to put the items being moved into the current node.
:Move the old left son pointer over so it stays with the corresponding item that has been moved.
:Now make the left son pointer for the node equal to TE$, the left son pointer of the items that were moved from the left brother. (They are now on the left side of this node.)

59544

Adjust the count of active items in this node by adding the number of items you borrowed from its brother.
:Now that you've restored it to at least half full, write it to disk.
:Since you were able to handle the underflow condition by moving items from a brother, and nothing had to be deleted from the father, the deletion is complete. Clear the underflow flag, TQ.
:Return to 59502.

59550 (If you've reached this point, the left brother can't spare any items, so you need to merge the left brother, the item from the father node, and the underfilled node.)

:Field the buffer so TW$ holds all the items.
:Store them outside the buffer in TS$.
:Mark the left brother node as empty.
:Put the "last deleted node" statistic, T2, in bytes 2 and 3 of this node, using the TP$(0) field.
:Now this is the "last deleted node".
:Subtract 1 from the count of active nodes in T3.
:Write this (now deleted) node from the buffer to disk.
:Call 59020 to get the father node again.
:Add the item from the father node onto the items that are being merged from the left brother. (Don't include its 2-byte right son pointer.)
:Make the left son pointer of the rightmost item in the father node point to the node that will be built from our merge. (It is the old underfilled node, so you can get its number from the next lower level of the stack.)
:Call 59564 to subtract 1 from the number of items in the father node.

59551

Clear the underflow flag, TQ. (Note: TQ is cleared at this point for programming convenience only. It may be set to –1 again in this line.)
:If the current node is the root and it is empty, then the root will now be the underfilled node you are replenishing. (Get its number from the stack.)
:Use bytes 2 and 3 of the node as a pointer to the prior deleted node, T2,
:make this node the last deleted node, T2,
:and subtract 1 from the count of active nodes, T3.
:Otherwise, if the father node is not the root, and it is now less than half full, then set the underflow flag, TQ, back to –1. (The father node is now underfilled so you will need to come back to it.)

59552

Call 59562 to write the father node to disk and to get the underfilled node on the next level back into the buffer.
:Field the underfilled node to isolate the items it contains as TW$.
:Combine the items that will go into the node as TS$.
:Field the buffer so that the contents of TS$ can be put into it.

59553

Put the merged items into the node's buffer.
:Mark it as full.
:Write the node from the buffer to disk.
:If TQ is still –1, it means that you've left the father node less than half full. So call 59030 to get the father node into the buffer, and to point TX to the proper position in the node,
:and return to 59502 to underflow the father node.
:Otherwise, TQ is 0, so the father node is okay. You don't need to get it again. Just return to 59502.

59560 (Shift left all the items that are to the right of the item to be deleted.)

:For each item position from the current to the rightmost active position in the node,
:copy data from the item to the immediate right,
:and repeat until all positions have been copied.
:Return to 59522, 59534, 59542 or 59552.

59562 (Write modified father node from buffer to disk, and get its underfilled son.)

:Write the buffer to disk.
:Add 1 to the level number, TL.
:Call 59020 to get the son.
:Return to 59522, 5534, 59542 or 59552.

59564 (Subtract 1 from count of active items in current node.)

:Subtract 1 from the ASCII value of the first byte of the node. Put the result back in the buffer as the first byte.
:Return to 59500, 59530 or 59550.

59566 (Get node and compute how many items can be moved to brother.)

:Call 59060 to get the node.
:Compute how many items can be spared by finding how much more than half-full it is. The excess is divided by 2 for even distribution between the 2 brothers. TH holds the result. (TH is 0 if this node is only half full.)
:Return to 59520 or 59540.

59568 (Replace item in father node with item stored in TA$.)

:Hold its right son pointer in TW$.
:Replace the item with TA$.

:Restore its right son pointer back to what it was.
:Return to 59510 or 59522.

59600 (Add non-unique key.)

:Load TW with –1 to make the search routine find the first position in the tree that is beyond all duplicates of the key to be added.
:Call the search subroutine, entering at 59110 to preserve contents of TW.
:Call 0 to add the new key into the node and position found by the search.
:Return to the main program.

Modifiying TREESAM for TRSDOS 2.3

The TREESAM subroutines as shown use the LOC function as a way to avoid unnecessary disk GET operations. This makes them operate faster than they would otherwise. The version of Disk BASIC that is supplied with Radio Shack's TRSDOS 2.3 on the Model I doesn't have a LOC function. One simple change cures the problem. Change line 59060 to read:

```
59Ø6Ø GETTF,TC:RETURN
```

If you've purchased a disk containing the demonstration programs and subroutines for this book, check line 59060. If your system has a LOC function, you'll want to change line 59060 on the disk back to the way it is shown in the listing:

```
59Ø6Ø IFLOC(TF)=TCTHENRETURNELSEGETTF,TC:RETURN
```

Machine Language Assist

Everything in the TREESAM handler is done in BASIC. This includes the binary search of each node. You can get about a 30% speed improvement for operations involving searches if you modify TREESAM to use the SEARCH1 USR subroutine from *BASIC Faster and Better & Other Mysteries*. SEARCH1 searches a string array for a key, and you can specify whether you want it to find the first string in the array that is greater than (or the first string that is greater than or equal to) your search key.

SEARCH1 requires a 4-element integer control array. To illustrate how you'd modify TREESAM to use SEARCH1, I'll assume you are using the C% array. I'll also assume you've defined C as integer.

First, your program should dimension the control array for 4 elements, 0 through 3:

```
DIM C(4)
```

You can do this anywhere in your program, as long as you do it before you use the array, and after you've done your DEFINT.

Next, you need a DEFUSR command in your program that assigns a USR subroutine to the address where you've stored the 133-byte SEARCH1 routine. I'll assume that you've protected the top 133 bytes of memory, and you've POKEd or loaded SEARCH1 starting at address –133. (That's memory address 65403.) To assign SEARCH1 as USR0, your command is:

```
DEFUSRØ=-133
```

To use it, the following lines replace the standard lines of the TREESAM subroutines:

```
59100 TW=0:C(3)=5
59110 C(2)=VARPTR(KY$):TQ=0:TL=1:TC=TR:IFLEN(KY$)<>TKTHENKY$=LEF
T$(KY$+STRING$(TK," "),TK)
59120 GOSUB59060
59130 C(0)=VARPTR(TK$(0)):C(1)=ASC(TM$):TX=USR0(VARPTR(C(0))):IF
TX=-1THENTX=ASC(TM$)+1
59140 TY=TX-1:GOSUB59040:IFTX<=ASC(TM$)THENIFKY$=TK$(TX)THENTQ=-
1:IF(TZAND2)=0ORCVI(TP$(TX-1))=0THENRETURN

59600 C(3)=5:TW=-1:GOSUB59110:GOSUB59200:RETURN
```

Planning Your TREESAM Index

Before you can implement a TREESAM application you've got to do some planning. Is an index sequential accessing method necessary and appropriate for your job? Do you have enough disk space available for the index or indices you want to use? How many records will you need to access? How long is each key? How will the keys be constructed? Must each key be unique, or will non-unique keys be permitted?

Let's take a simple application. Suppose you are the manager of a large apartment complex. Your parking lot has a limited number of spaces. They are available only to tenants and any tenant may take any space. You want to make sure that no unauthorized vehicles park in your lot. From time to time, tenants buy new cars and sell old ones, or they move in and out of your complex.

You've decided that the solution to your problem is to keep a list of the license plate numbers for all the authorized vehicles. You want to be able to walk out into the lot periodically to check all license plates against the list. To make your job easy you want to keep the list in alphabetical (or numerical) sequence. As changes occur, you want to be able to make additions and deletions quickly. At this point, that's all you want. You don't care who owns a particular car, as long as the license is on your list.

This problem is typical of one of the simplest applications for the TREESAM handler. You have keys that can be assumed to be unique. In this case they are 6 or 7 characters long. You don't need to keep any other information. You just want to keep them in sequence and to be able to make additions and deletions easily. A similar problem is keeping a printed list of lost or stolen credit cards by number, and being able to inquire into the computer for an even more up-to-date verification. You may own a small shop and want to keep a list of the driver license numbers of those customers who have authorization to cash checks. A dictionary of commonly misspelled words might be another application.

For the license plate example, you can allow 7 bytes for each key. There is no data field required. Compute the maximum number of keys per node with the following formula:

```
Keys Per Node = (256 - 3) / (Key Length + Data Length + 2)
```

The "256" in the formula is the maximum size of a node, 256 bytes. The "3" accounts for the 1-byte key-count and the 2-byte left-son pointer stored in each node. The "2" accounts for the 2-byte right-son pointer stored with each key. To make certain that the result is evenly divisible by 2, as TREESAM requires, apply the following formula:

```
Keys Per Node = INT (INT (Keys per node / 2)) * 2
```

Since the key length is 7 and the data length is 0, you can conclude that a maximum of 28 keys can be stored in a node. Now you want to know the maximum number of nodes required to be sure you will have enough disk space. You know that each node, except the root node, will be at least half full. That is, each node will contain at least 14 items. The root node will contain at least 1 item. Assume that you want to allow for a maximum of 1000 keys. The number of nodes required is given by the following formula:

```
Nodes Required = ((Key Capacity - 1) / Half Node Capacity) + 1
```

The first "1" subtracts the single key that may be stored in the root node. The second "1" adds one node for the root. If you get an answer that has a decimal or fraction, round up to the nearest whole number. Substituting the numbers in the example gives you ((1000 - 1) / 14) + 1, or 71.357, which rounds up to 72. At most, 72 nodes will be required.

Now you add 1 to account for the statistics record in the TREESAM index. In total, 73 records are required in the file to allow for a worst-case tree organization.

You can make another calculation to get better disk efficiency. Most of the newer TRS-80 disk operating systems let you have variable length records. That is, instead of using the standard 256-byte record, you can use, say, 192 or 200 byte records. To take advantage of this, figure out how many bytes are required per node with the following formula:

```
Bytes Per Node = 3 + Keys Per Node * Item Length
```

In the license plate example the item length is 7 + 0 + 2, or 9 bytes. That gives 3 + 28 * 9, or 255 bytes per node. There's really not much value in complicating the program to avoid wasting 1 byte per node in this case, but in certain cases, especially with large indices storing large items, there can be a significant savings. The key-plus-data length that wastes the most space per 256-byte record is 62 bytes. That calculates out to only 131 bytes required per node. It can be worth while to make the calculation or to make a chart that shows you the most efficient key-plus-data lengths.

Storing Data in a Tree

The first example showed how to use TREESAM to store keys only. Now let's suppose you want to store some data along with each key. The TREESAM subroutines let you specify how much data to be with each key in a tree. Normally, you will only store 2 bytes of data with each key, those 2 bytes being a pointer (in MKI$ format) to a record in your master file. At the expense of disk efficiency, but with the luxury of programming convenience, you can store everything that goes along with the key right in the tree.

Suppose the apartment manager in the example wanted to list a name along with each license plate number. That way, if a car is double-parked the owner can be quickly notified. Now you have got two pieces of data: the license number, which is a 7-byte key, and the name, which, let's say, is a 24-byte data field. Instead of spending the time to program a separate master file to hold the names, you can decide to store everything in the tree.

Now the item length is 33 bytes. That's 7 for the key, 24 for the data field, plus 2 for the right son pointer. You can calculate that there will a maximum of 6 items per node, and if you want a capacity of 1000 names and license numbers, you can store the B-tree in 334 records. It will just barely fit on a 35 track single-density TRS-80 Model I diskette.

There are other applications that can use the same approach. A name and phone number list, for instance, could use a 22-byte name as a key, and a 10-byte phone number as the data field. The system could make alphabetical lists and do quick look-ups, just by typing in a name or the first few letters of a name. Likewise, you could use TREESAM to build a subject index for a book. The keys could be short descriptions of each subject covered. The data fields would store the page numbers. Perhaps you want to build a small dictionary. The words are the keys; the data fields can store short definitions or synonyms. Perhaps there's been a disaster and we want to assist authorities in maintaining a list of all survivors, by name, that have been accounted for. The data field with each name might store that person's age for further identification. The possibilities are endless, and very little programming is involved.

Indexing a Master File

Storing data in the tree itself makes it easy on the programmer, but not so efficient in terms of disk space utilization. Since each node can be as little as half-full, you're better off if you keep the item length as short as possible. It's better to confine the inefficiency to key storage than to spread it over all the data being handled by your application. More keys per node also means faster accessing because the tree will span fewer levels.

A 2-byte data field can store an integer ranging from –32768 to 32767. Record number pointers are always positive, so you've got a range of from 0 to 32767. If your master file is more than 32767 records, you can use a function that converts a 2-byte negative or positive integer into a positive number from 0 to 65535. If that is not sufficient, you can use other methods for storing numbers in compressed space. Many ways are explained in my book, *BASIC Faster and Better and Other Mysteries.*

To show the considerations in indexing a disk file, assume that the apartment manager already has a disk file in which there is 1 record for each apartment. Each record in the master file contains the following data fields:

```
Unit Number      4 Bytes, string.
Tenant Name     24 Bytes, string.
Deposits Held    4 Bytes, dollar amount in MKS$ format.
Monthly Rental   4 Bytes, dollar amount in MKS$ format.
```

You can use the apartment unit file to provide the name that goes with each license number key. In fact, if a particular tenant has more than one car, you can store each license number in a TREESAM index and have it index the same name. You can do all this without changing the apartment unit file. The B-tree stores each license number along with a pointer that gives the record number in the master file where information about the apartment unit is stored.

This application is an example of a case where you have unique keys in the index, but more than one key may be used to access the same record. Assume for now that you'll have no need to perform the access in the opposite direction. That is, you don't expect to access an apartment unit record and then want to find all the license numbers that go with it. TREESAM doesn't do that. (To delete license numbers in this example, the tenants would have to report them to you when they are no longer being used.)

In the apartment management application you will also probably want to access each apartment by unit number. A second TREESAM index can be used for that purpose. That index would be the primary means of access, since it contains all the apartment numbers with pointers to the master file. Most likely you'd want to store the apartment number in the master file as well as the index. That way, if you have an inquiry by license plate number, you can display the name as well as the unit number of the owner. Also, if you are reading the master file in record-by-record sequence, you have the unit number available.

The ideas mentioned in the last few paragraphs are important to keep in mind when planning an application that uses TREESAM. During additions and deletions, the keys in a TREESAM index move within the nodes, and they move from node to node. It is not practical to go from a master file record back to the key that indexes it. If that is a requirement for your program, you need to have a field in each master file record that duplicates the contents of the key stored in the index. This is most often a consideration when you want to sort and print the master file in sequences other than the sequence that is built into the TREESAM index. It is also important when more than one index cross references the same master file. In such a design, a deletion of a record in the master file requires that the program look-up and delete all keys in the indicies that reference the master file. It can only do this if it knows what keys apply.

There are millions of ways to apply an index which points to a master file. Here are some examples:

- A check register index can contain check numbers. The index points to a master file that contains the date, payee, amount, and whether or not the check has cleared. Checks are maintained in sequence, and there is no problem with inserting or deleting.

- An index of account numbers can be used to access a general ledger, accounts payable or accounts receivable master file. An index of products can access an inventory file. An index of employee numbers or social security numbers can be used to access a payroll or personnel file.

- An index of radio station call letters can be used to access a radio listening log. An index of roll numbers can be used to access information for a carpet retailer. An index of ticker-tape codes can be used to keep information on stocks and bonds. An index of dates and times can keep track of appointments. An index of airport codes can access a file which maintains records of arrivals and departures or weather conditions.

Non-Unique Keys

The idea of having one or more keys in the same index accessing the same master file record is called "alternate" keys. You saw an example with one or more license plate numbers accessing the same apartment unit record. The idea of indexing on two or more data fields in a master file is called "multiple index accessing," "multiple key accessing" or "cross referencing." It is exemplified by the concept of being able to access an apartment by unit number as well as license plate number. The capability for non-unique keys is something entirely different.

A unique key is one that has no duplicates. You expect to have only one license number "324XLV". In an inventory system, you can expect to have only one product number "R3234A". A non-unique key is one that may have duplicates. Here are some examples:

- You want to use a person's last name to access records in a mailing list. More than one person might have the same surname.

- You want to index a record album collection by the names of the performers. You may have more than one record album by the same artist.

- You want to index your possessions for insurance purposes. You may have more than one of any particular item.

- You may want to keep a list of inventory item purchases orders by item number. There may be more than one order outstanding for a given item.

- Since it's easy to remember the street but not the full address, a real estate broker might want to index a file of listings by street name.

- A video tape rental service might want to keep track of an inventory of tapes by movie title. More than one copy may exist for any tape.

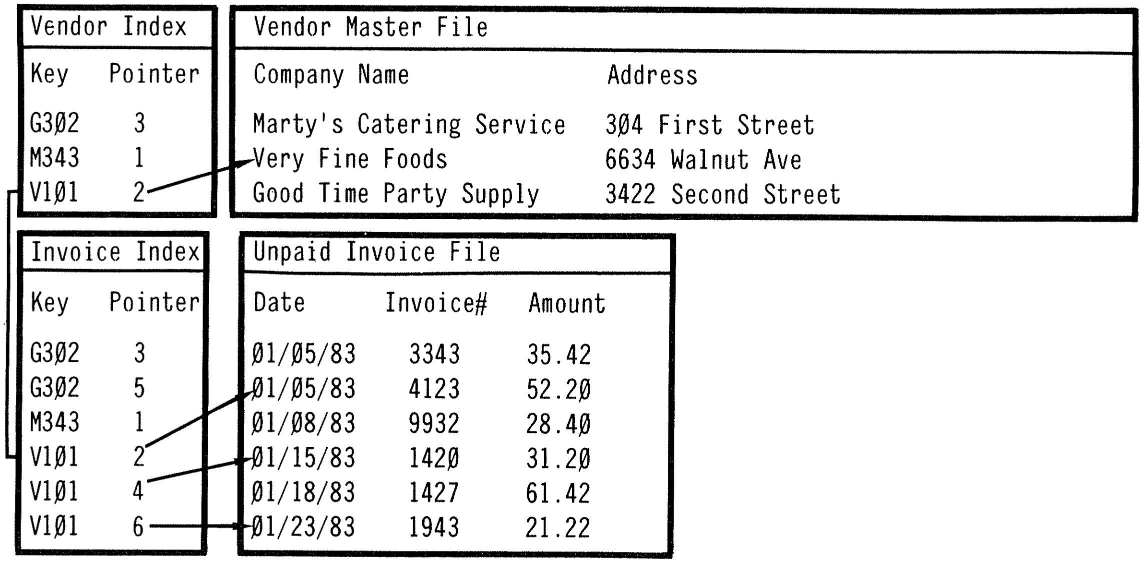
TREESAM allows you to specify whether non-unique keys are permitted in an index. It handles them on a FIFO basis. FIFO stands for "first-in first-out." The first time you add a key, it is just like a unique-key system. When you add a duplicate, the new key is inserted just after any other equal keys in the tree.

When you search for a key, the first one that was added is returned to you. Then if you perform a read-next, the next one added is returned. The last one you added is returned last. The same applies in a printout where you are reading through the whole file. If there are duplicates of a key, the first one added is the first one printed.

Programming for non-unique keys is a little more complicated. To provide for changes or deletions in the master file, you've got to allow the operator to go forward and backward through the list of duplicate keys until the right record related to a non-unique key is found. On a search, it is sometimes desirable to have the program check ahead to see if there are duplicates for a key that the operator has selected. Then it can display a warning.

We discussed the detail file handler and its purpose earlier in this book. Non-unique keys can also be used in situations similar to those that the detail file handler is used for. TREESAM is especially appropriate where there are a large quantity of additions and deletions in a detail file, and the individual records must be kept in sequence. Consider, for example, an accounts payable system. Suppose you have 100 vendors you pay your bills to. For any vendor, you may have more than one outstanding unpaid invoice. To handle it, construct an open invoice file that uses the vendor number as the key. A separate file can contain the vendor names and addresses. It can be accessed by the same vendor numbers, but they are unique for that file.

Let's say you have a unique number "V101" in one B-tree that points to a vendor named "Very Fine Foods" in the vendor master file. They've sent you 3 invoices you have not paid yet. A second B-tree allows non-unique keys and contains 3 "V101" keys. They point to 3 different records in the unpaid invoice file, where the amounts and due dates are maintained. The keys in the unpaid invoice index are maintained on a FIFO basis, so that's perfect for your needs. The oldest invoices show up first on the screen. As you pay any invoice, it can be easily deleted. The illustration below shows a simplification of the contents of each file:



Vendor Index

Key	Pointer
G3Ø2	3
M343	1
V1Ø1	2

Vendor Master File

Company Name	Address
Marty's Catering Service	3Ø4 First Street
Very Fine Foods	6634 Walnut Ave
Good Time Party Supply	3422 Second Street

Invoice Index

Key	Pointer
G3Ø2	3
G3Ø2	5
M343	1
V1Ø1	2
V1Ø1	4
V1Ø1	6

Unpaid Invoice File

Date	Invoice#	Amount
Ø1/Ø5/83	3343	35.42
Ø1/Ø5/83	4123	52.2Ø
Ø1/Ø8/83	9932	28.4Ø
Ø1/15/83	142Ø	31.2Ø
Ø1/18/83	1427	61.42
Ø1/23/83	1943	21.22

In designing the accounts payable application, there is an improvement that comes to mind. You may want to combine both B-trees into one. Your program could use the first occurrence of a vendor number to point to the vendor master file. Any subsequent occurrences could be considered to point to the open invoice file. The result is a savings in disk space, a savings in memory, and a speed improvement because only one search is required to access a vendor and the applicable open invoices.

The same idea can be applied to an open item accounts receivable file. The B-tree keeps all customer invoices in order by customer number. If there is more than one invoice for a customer, they are sequenced by transaction date. Additions and deletions are quick and easy.

Other Kinds Of Keys

A key can be nearly any string made up of letters, digits, and special characters. If you wish, you can use subroutines or functions to compress data for use as keys. The COMUNCOM USR subroutine from *BASIC Faster and Better and Other Mysteries*, for example, stores a string in two-thirds the space it would normally require. TREESAM doesn't insist that your key be displayable. When the operator inputs a key your program can translate it to the format that will be used to store it. When keys are to be displayed or printed, they can be translated back.

Your only limitation is that the keys must be sortable. That is, when a comparison of any two keys is made, the one that you intend to be greater than the other must come out that way. Suppose you want to use integer numbers as your keys, as you might in a check register system for the check numbers. You sense that you can save disk space by storing the check numbers in the B-tree as 2-byte integers in MKI$ format. The problem is that they won't sequence properly, unless you do a conversion before searching for the key. *BASIC Faster and Better* shows ways to get around these problems. In special applications, you can make modifications to TREESAM so the key comparisons will work properly for the data you intend to store.

In most of our discussions, I am assuming that the operator will be inputting a key from the keyboard, and TREESAM will act on it from that point. Many programs may be written, though, in which the index sequential accessing method is entirely invisible to the operator. A program may build its own keys, search, insert, and delete, without the operator even being aware of what's going on.

One example is the use of TREESAM to sort a file. Let's say, for instance that the operator tells the computer to sort the apartment unit file in ascending sequence by monthly rental rate and unit number. A program can read through the master file, extracting the rates and unit numbers. As it reads each record, it can convert the rate into a sortable format and concatenate it with the corresponding unit number to make a key. Each key can be inserted in a temporary TREESAM tree to make an index that is sequenced by rate and apartment number. After the desired printout is made, the temporary TREESAM index can be killed.

You can get quite creative when designing the keys that are to be used. Many special requirements can be satisfied by combining two or more pieces of data to build a key. Another approach is to save disk space in your index by using partial keys. An index that accesses members of a club for a mailing list, for example, might just use the first three letters of a member's name for keyed access.

How to Use TREESAM In Your Program

Once you've decided how your files will be organized, the indices that will be required, and the keys and data fields that they will hold, your next step is to start developing your program or programs. One of the best ways to get familiar with the TREESAM handler is to work with the TREESAM/BAS utility. It is a program that defines and builds a TREESAM index. It lets you try out the many capabilities of the TREESAM handler and examine how the various operations are controlled by a BASIC program.

On the next few pages we will see how to install TREESAM in a program, and go through the ways your program can interact with the TREESAM subroutines.

Installing TREESAM

Since TREESAM is a series of subroutines that occupy lines numbered between 59000 and 59600, the first step is to merge them into the program you are developing. Earlier we discussed the fact that, depending on how you divide up your application, certain capabilities may not be needed in a given program. The list of TREESAM subroutines in Chapter 12, with the accompanying discussions can help you decide just what lines you will need. If you have any doubts at this point, then merge all the routines into your program. You can delete the ones you don't need later.

If you are going to have other files in your program, such as master files and detail files, that part of your programming job can be done as you normally would. You can use your own methods, or you may want to use many of the methods discussed in this book. TREESAM as listed, does not require any external subroutines to perform its functions.

Housekeeping

There are a few things you should put in your BASIC program so TREESAM can function optimally.

The CLEAR command reserves space in upper memory for string storage. CLEAR 1000, for example, sets aside space for strings totaling 1000 bytes. Any data that is held in disk buffers does not use up string storage space. TREESAM uses, at most, about 500 bytes of string storage during the data manipulation it does. Add that to the string storage the rest of your program will require to arrive at a value for your CLEAR statement. Generally, the more string storage space you allow, the faster your program will run. Often it's a matter of trial and error. If you get an "Out of String Space" error, you know you've got to increase the number of bytes specified by your CLEAR statement. It should normally be one of the first commands executed in your program.

All variables within the TREESAM subroutines are of the integer type, unless otherwise explicitly defined. Since they all begin with "T", a "DEFINT T" statement should be executed before any of the TREESAM subroutines are called. If you don't redefine "T" variables, the DEFINT command only needs to be executed once in your program.

Line 59010 dimensions the 5 arrays that are used in TREESAM. If you are going to be using an especially large index, or if your key-plus-data length is rather long, your tree might eventually grow to more than 5 levels. If you expect more than 5 levels, the dimension of TS(5,1) in 59010 should be increased by changing the "5" to the number of levels you anticipate.

You can estimate the maximum number of levels that will be required if you know how many keys you'll be storing and the minimum number of keys per node. Use the following formula:

```
   Levels Required = LOG (Keys to be stored) / LOG ( Keys per node / 2 + 1 )
```

Round your answer up to the nearest whole number. Thus, if you want to store 2500 keys, and each node has a capacity of 30 keys, the maximum number of levels required is LOG (2500) / LOG (3Ø2 + 1) = 2.822 or 3, rounded up.

Optimizing TREESAM Search Speed

To optimize the TREESAM subroutines for searching speed, initialize the variables that will be used most frequently in your program before other less critical variables. Each time BASIC evaluates a variable in your program it must search a list in memory to find where the variable value is stored. Ideally you want the most-used variables to be at the beginning of that list. You can accomplish this by inserting the following program line early in a program that uses TREESAM:

```
   2 TH=Ø:TX=Ø:KY$="":TY=Ø:TW=Ø:TC=Ø:TM$="":TL=Ø:TF=Ø:TK=Ø:TW$="":T
   A$="":TS$="":TE$="":TN=Ø:TQ=Ø:TR=Ø:TZ=Ø
```

A line number other that 2 can be used, and if you wish, you can make it into a subroutine. Just call it early, and make sure that you've already executed your CLEAR statement, and variables starting with "T" have already been defined as integers. Though pre-initializing the most commonly used variables is not required in your program, I've found that it provides about a 28% improvement in searching speed, and a 20% improvement in search-add operations. (I concluded that the sequence shown is best by making a temporary modification to TREESAM that caused a count to be made of each use of each variable. Then I did several several test runs with hundreds of keys and examined the results.)

As you know, disk BASIC allows you the choice of any of 15 disk buffers. A TREESAM file can be opened in any buffer, provided, of course, that you answer the "How Many Files" question properly upon entering BASIC. Before calling any TREESAM subroutine, the integer variable TF must contain the number of the file you are using. For instance, if your program only uses two files, file 1 being the TREESAM index and file 2 being the master file it indexes, you can say "TF=1" sometime early in your program. TREESAM will then always know that file 1 contains the index. If you've got more that one TREESAM index in a given program, you need to reload TF each time you go from one index to another.

Creating a TREESAM Index

The first step in creating a TREESAM index is to build a file that's long enough to hold the number of nodes that will be needed in a worst-case situation. Though this step is not required, it is an important practice so unexpected disk-full errors can be avoided. A disk-full error encountered while TREESAM is splitting a node to add a new key could result in the loss of the keys that were stored in half of the old node. It can also cause some of the internode pointers to be left with invalid values.

To build the file, first decide whether you are going to use the variable length record feature (if your disk operating system allows it). You might calculate, for instance, that you will only require 195 bytes per node, rather than the 256 byte default. Open the file you intend to use, in random mode, on the disk drive that you want. If the index file name is to be "PRODINDX" on drive 1, and you want to opt for the default of 256 byte records, the command is:

```
OPEN"R",1,"PRODINDX:1"
```

If you are going to use variable length records, and you've allowed for the capability upon entering BASIC, the command might be something like:

```
OPEN"R",1,"PRODINDX:1",195
```

Now you can proceed to fill the file with records that contain nothing but hexadecimal zeros. If you've decided that 250 nodes will be required, and that each node will be 256 bytes, and 1 extra node will be needed for the statistics record, you can use the following:

```
FIELD 1, 128 AS A1$, 128 AS A2$
LSET A1$=STRING$(128,Ø) : LSET A2$=A1$
FOR X=1 TO 251
PUT 1, X
NEXT
```

It is optional, but recommended to build the whole file. If you don't wish to do so at this point, you must, at least, build the first two records.

Before TREESAM can use a file as an index, the file must have the desired parameters and beginning statistics stored in the first 20 bytes of record 1. The values required for an empty tree are shown below:

Bytes	Contents	Command
1-2	Half node capacity, TN	LSET TN$=MKI$(TN)
3-4	Key length, TK	LSET TK$=MKI$(TK)
5-6	Data length, TD	LSET TD$=MKI$(TD)
7-8	Item length, TI	LSET TI$=MKI$(TI)
9-1Ø	Options Code, TZ	LSET TZ$=MKI$(TZ)
11-12	Root node	LSET RN$=MKI$(2)
13-14	Next node, end of index	LSET T1$=MKI$(3)
15-16	Last node deleted	LSET T2$=MKI$(Ø)
17-18	Nodes active	LSET T3$=MKI$(1)
19-2Ø	Keys active	LSET T4$=MKI$(Ø)

TN, TK, TD, TI, and TZ are the parameters. Before recording record 1, they will have been loaded with the values you've decided on for the particular index you're creating. The others, in bytes 11-20, are the statistics. Notice that we set up an empty tree as if one node is already active. These beginning statistics are always the same for any tree that you may wish to create.

The LSET commands in the chart above assume you've already opened the file and fielded the statistics record. A "GOSUB 59005" can do the fielding for you if you've already opened the file, and you've already loaded TF with the file number. The command "PUT TF,1" records the buffer you've set up into the first record of the file.

Finally, you should close the file you've built. This locks in the end of file pointers so the file will have the length you intended the next time you open it.

TREESAM/BAS, which we will discuss later in this chapter, performs all the functions required for creating an empty index file. You can study lines 500 through 781 to see how it is done. For the final program that you develop, you may want to write a specialized creation routine so the operator doesn't have to key in all the information about key lengths, data lengths, and the other things that TREESAM/BAS requests. By hard-coding the parameters, you can make it a completely automatic process.

Initializing TREESAM

There are certain things your BASIC program must do before it can use a TREESAM index. First, it is your program's responsibility to open the file that is to contain the B-tree, as well as any other files that will be required. You can use your own routines or the applicable subroutines explained in this book. For instance, you might wish to use the standard random disk file handler open subroutine. If your index is to be in file 1 and its name is "ARINDEX:2", you can say:

```
FS$="ARINDEX:2":PF=1:GOSUB5825Ø
```

Once the index file is open, call the TREESAM initialization subroutines. Their function is to dimension the proper arrays, field the buffers, and load the statistics from record 1 into the required integer variables.

First you need to "GOSUB 59000". This fields the statistics record, gets it, and loads the integer variables, TN, TK, TD, TI, TZ, TR, T1, T2, T3 and T4. Remember that TF must contain the number of the file you are using before you make this call.

Next, you need to "GOSUB 59010". This dimensions the arrays that will be needed and fields the buffer to handle the node organization that will apply. A "GOSUB 59010" can only be executed once during the execution of a program because it dimensions arrays. A second call to 59010 will result in a redimensioned array error. If it becomes necessary to refield the buffer (it won't, unless you have multiple indices), you can use "GOSUB 59011" on subsequent calls.

We will discuss techniques for handling multiple indices later, but in the case of a single-index program, after your "GOSUB 59000" and your "GOSUB 59010", your program is ready to begin using the major TREESAM subroutines.

Searching for a Key

The search subroutine is the most important in the TREESAM handler. The obvious use of it is to look-up and access a key that is in the B-tree. It is also used though, when you want to add or delete a key.

To search the tree for a key, your program loads KY$ with the search key. Then execute a "GOSUB59100". Upon return, TQ is 0 if the key is not found, or –1 if the key is found in the tree. A simplified search call, where the operator inputs the key, can be programmed as follows:

```
LINEINPUT KY$
GOSUB 59100
IF TQ THEN PRINT "FOUND" ELSE PRINT "NOT FOUND"
```

The search subroutine relies on the contents of TZ. TZ tells it whether all the keys in the tree are unique or not. If TZ is 2 or 3 (more precisely, if bit 2 of TZ is set), the search subroutine assumes that the keys in the tree are non-unique, so it has to be sure to find the first one in case a key has duplicates. If TZ is 0 or 1 (bit 2 is not set), the search subroutine assumes that every key in the tree is unique, so the process is a little simpler. If you've built your index file properly, TZ is already specified by the statistics record. During run-time you don't have to worry about it.

Before starting the search, subroutine 59100 automatically truncates or pads the search key you supply. If you've provided a search key that is longer than the standard key length, it shortens KY$. If you've provided a search key that is too short, it adds blanks to KY$ to make it the standard key length, specified by TK.

Upon return from a successful search several variables contain information that is useful:

TK$(TX) contains the key that was found. It will be equal to KY$.

TD$(TX) contains the data string that is stored with the key that was found. If the data string is a pointer to a record in another file, it is up to your program to convert TD$(TX) into a record number and access the indicated record from the file being indexed. If your tree contains non-unique keys, TD$(TX), after a search, is the data field for the first key. The data fields for any subsequent keys that are equal to the first can be gotten by executing the read-next subroutine.

TC is the record number of the current node in the B-tree. If you wish to change the contents of the data field, you will need to know TC in order to do your PUT.

TL specifies the current level in the tree. The stack is already loaded with the information that calls the read-next, read-previous or delete subroutines.

If your search results in a not-found condition (TQ=0), certain variables are set up in case you want to add the key. The stack contains information about where you are in the tree. TL contains the level you are on (which will always be the lowest level of the tree after an unsuccessful search). TC specifies the current node, and TX specifies your position in the node. TREESAM handles all these variables automatically. The important thing is not to alter them if you want to follow up your unsuccessful search with an add.

Adding a Unique Key

In a system where all keys must be unique, a call to the add subroutine must follow an unsuccessful search. It is called by issuing a "GOSUB 59200". If an operator is sitting at the keyboard, the logic is most often handled as follows:

```
LINEINPUT"SEARCH KEY: ";KY$    'Operator enters the key
GOSUB 59100                    'Call the search subroutine
IF TQ THEN GOTO .....          'If found, go display the data
LINEINPUT"ADD IT? ";A$         'Allow for chance to escape
IF A$="N" THEN GOTO .....      'Abort if operator wishes
LINEINPUT"DATA: ";DA$          'Get the data to be stored
GOSUB 59200                    'Add the key and data
```

If a pointer to another file is being stored in the B-tree the above sequence would be modified slightly:

```
LINEINPUT"SEARCH KEY: ";KY$    'Operator enters the key
GOSUB 591ØØ                    'Call the search subroutine
IF TQ THEN GOTO .....          'If found, go display the data
LINEINPUT"ADD IT? ";A$         'Allow for chance to escape
IF A$="N" THEN GOTO .....      'Abort if operator wishes
```

At this point, call subroutines that select the record number in the master file. Allow data to be entered for the master file record, and write it to disk. Assuming the master file record was written as record "R". . .

```
DA$ = MKI$(R)                  'Load the master file pointer
GOSUB 592ØØ                    'Add the key and pointer
```

If keys are to be added to a non-unique tree by the operator in one step, or in a batch mode under program control, the logic can be arranged as follows:

```
LINEINPUT"KEY: ";KY$           'KY$ contains key to be added
LINEINPUT"DATA:";DA$           'DA$ contains data to be added
GOSUB 591ØØ                    'Search for the key
IF TQ=Ø THEN GOSUB 592ØØ
ELSE PRINT "DUPLICATE KEY,
NOT ADDED!"                    'Add if not already in file
```

It is important to realize that it is impossible to predict where you'll be in the tree immediately after an add. This is because node splitting may have occurred and you might not be anywhere near the node that received the new key. The only permissible TREESAM operations just after a unique key add are read-first, read-last and search. Following an add, a delete or a second add call without the intervening execution of a search will produce damaging results. It is up to your program to prevent these situations from occurring. (If you haven't made any logical errors, your program will flow so TREESAM's subroutines are called in a valid sequence. If your program gives a lot of control to the operator in selecting which subroutines are to be executed, as TREESAM/BAS does, you've got to take precautions. Notice how TREESAM/BAS prevents invalid commands.)

Updating the Statistics Record

After any add or delete operation, the statistics for your B-tree will change. The count of keys active will be different, and there's the possibility that the count of nodes active, the root node record number, and the location of the last in file and last deleted record will have changed. The TREESAM subroutines keep the memory variables up to date, but they don't automatically update the statistics in record 1 of the B-tree file.

Subroutine 59050 performs the function of writing the statistics to disk. To call the subroutine, your program simply issues a "GOSUB 59050". Your call to 59050 can be deferred to the end of a program. This will make add and delete operations a little faster, but it involves a risk in the event your program is improperly terminated because of a power failure or operator carelessness. I recommend that you GOSUB 59050 after every add or delete. The only exception I would consider is a program that automatically does a batch of many additions or deletions. In that case, the call to 59050 can be done after the batch operation has completed. Delaying the statistics update can make a significant time difference when many adds or deletes are involved, but for operator-paced data entry, the time it takes to record statistics is hardly noticeable.

Adding a Non-Unique Key

A special subroutine in the TREESAM handler controls the process of adding non-unique keys. Subroutine 59600 combines the search and add operations into one call. This subroutine is used whether you are adding a unique key in a file of non-unique keys or you are adding another key that is the same as one already in the tree.

The search preceding a non-unique add is a little different from a search before an add to a unique-key tree. Before a non-unique add, the search subroutine must find the position in the tree that is just beyond the last key in a set of (possibly many) duplicates already in the tree. The search subroutine does this by changing the binary search logic so comparisons are made on a greater than or equal basis, rather than just a greater than basis. The way the search subroutine performs is controlled by the variable TW. If TW is –1 and the search is entered via line 59110, it will perform in a "search before non-unique add" mode. As a programmer using the TREESAM routines, you don't really need to know this because subroutine 59600 handles it all. To add a key to a non-unique-key tree you load KY$ with the key, DA$ with the data or pointer to be stored, and you just issue a "GOSUB 59600".

To go back to the open-item accounts payable example from a few pages ago, the logic to handle the addition of a new unpaid invoice to the files might go something like this:

```
LINEINPUT"VENDOR NUMBER: ";V$  'Specify the Vendor
```

. . . here you might check against a unique-key index to make sure the vendor number is valid.

Now allow entry of the due date, invoice number and amount due, and store it in the next available record in the open-invoice file. Let's say that record is "R". . .

```
KY$=V$                        'The key is vendor number
DA$=MKI$(R)                   'Specify pointer to invoice file
GOSUB 59600                   'Add new key to non-unique index
```

As another example, you can use a B-tree to store a list of items that you've loaned to your friends. The key will be the borrower's first name. The data field will contain a description of the borrowed item. Since a person may have borrowed more than one item, you have got a non-unique key tree:

```
LINEINPUT "BORROWER'S NAME: ";KY$
LINEINPUT "WHAT IS BEING BORROWED?  ";DA$
GOSUB 59600
```

Later, if you wish, you can use the search subroutine for any name. Perhaps you key in "JACK", and the display shows that he borrowed your lawn mower. Upon finding the name, you can do repetitive read-next calls, displaying the contents of the data fields to see a list of every item your friend has borrowed and not returned yet. This may show that the same friend has borrowed some clippers and a book. The list will be in the sequence in which the items were put into the computer, so most likely that will be the sequence in which they were loaned. A sequential printout of the entire tree might give the following list:

```
BOB       $50
JACK      LAWNMOWER
JACK      CLIPPERS
JACK      BOOK
SALLY     MIXER
SALLY     BOWL
```

Read Next and Read Previous

We've mentioned read-next and read-previous a few times without explaining how they work. Let's say you've done a search for the key "LXR334" (a license plate number), your search was successful, and you want to find out what the next key in the file is. You can call subroutine 59300 to get the next key in the file. Perhaps it's "LXS100". From that point you might want to call subroutine 59400 to do a read-previous, which takes you back to "LXR334". Another read-previous may take you back to "LMO099". As you can see, read-next and read-previous moves you through the tree starting from any point.

You may perform a read-next immediately following a successful search, a prior read-next, or a read-previous. You simply "GOSUB 59300". Upon return:

TQ=–1 if a subsequent key has been found, and TL (the current level) is non-zero. TQ=0 and TL=0 if the read-next brought you to the end of the tree. When TQ is 0, consider it an unsuccessful read-next.

TK$(TX) contains the key after a successful read-next.

TD$(TX) contains the data related to the key.

TC contains the record number of the current node.

TL contains the current level, and the stack contains all the necessary information to identify to the program where we are in the tree.

Read-first is a special case handled by the read-next subroutine. To tell subroutine 59300 that you want it to return the first key in the tree, you preset the level indicator to 0 before your call. The command for read first is:

```
TL=Ø                        'Reset level to Ø
GOSUB 593ØØ                 'Read First
```

Here's a routine that prints, in sequence, all the keys in a TREESAM B-Tree:

```
1ØØ TL=Ø                    'Reset level to Ø
11Ø GOSUB 593ØØ             'Read next
12Ø IF TQ=Ø THEN 15Ø        'Handle end of tree
13Ø PRINT TK$(TX)           'Display the key
14Ø GOTO 11Ø                'Repeat
15Ø PRINT"END"              'Show we're done
```

Read-previous works just as you might expect it to (like read-next, but in reverse). The read-previous subroutine is accessed by "GOSUB 59400". The conditions upon entry and return are exactly the same except:

- If you set TL to 0 before a call to subroutine 59400, the last key in the file is returned.

- If TL or TQ equals 0 upon return from subroutine 59400, you've reached the beginning of the file.

Replacing line 110 in the program example above with. . .

```
11Ø GOSUB 594ØØ             'Read previous
```

. . . makes the program read all the keys in a TREESAM index in a descending sequence — from highest to lowest.

To make a printed report of the entire file, change line 130 so it calls a subroutine which prints the data fields as well as the keys. If the data field associated with each key, TD$(TX), is a pointer, the record in the related master file is accessed, and data from the master file is included in the printout.

The read-next and read-previous subroutines are very important when searching for a key in a non-unique file. The first key matching the search key is returned first. You must use read-next to access the duplicates. To print or display the all matches for the search key, you have to read-next until a non-matching key or the end of the tree is found.

Approximate Searches

One of the nice features of TREESAM is that it will do partial-key or approximate searches. A partial-key search is when the operator might key in "JA" to find the key, "JACKSON". Only enough characters need to be typed in to distinguish the search key from other keys that might be in the file. An approximate search is one in which the operator types as many characters as desired, and the system returns the first key in the tree that is equal to or greater than the key that was specified. (If the key specified is greater than any in the file, an end-of-file message is returned.) Once an approximate match is found, the operator may give commands to read-next or read-previous to get to the key that is actually wanted.

A subroutine for approximate search is not included in the TREESAM handler itself, but you can implement one with the following logic. Upon entry to the subroutine, KY$ has the approximate key to be searched:

```
'Approximate Key Search Subroutine

GOSUB591ØØ                   'Call the search subroutine
TQ=-1                        'Force found indicator to true
IF TX>ASC(TM$)
   THEN TX=TX-1 :GOSUB 593ØØ 'If end of node, read next key
RETURN                       'Return to main program
```

Upon return from this subroutine, TQ=0 if no key in the tree was greater than or equal to the search key. Otherwise, TQ=–1 and TK$(TX) contains the key that was found. TD$(TX) contains the related data field.

Once you have approximate-key search capability, partial-key capability is assumed. If you want to insist that a not-found condition is returned if the requested key doesn't exactly match the first few characters of the key found, use the following subroutine:

```
'Partial Key Search Subroutine

TA$=KY$                      'Store Partial Key in TA$
GOSUB591ØØ                   'Call the search subroutine
IF TX>ASC(TM$)
   THEN TX=TX-1 :GOSUB 593ØØ 'If end of node, read next key
IF TQ=Ø THEN RETURN          'Return if end of file
IF TA$<>LEFT$(TK$(TX),LEN(TA$))
   THEN TQ=Ø                 'Compare search key to key found
RETURN                       'Return to main program
```

Upon return from the partial-key subroutine, TQ is 0 if no key in the tree begins with the characters you requested as KY$. Again, if TQ is –1, TK$(TX) contains the key that was found, and TD$(TX) contains its data field. You'll want to provide a read-next capability immediately after the search to allow the operator to display any other records having a key that starts with the same characters.

Deleting A Key

The deletion subroutine, 59500, is the longest and most complex in the TREESAM handler. In many applications you may wish to put the deletion routines in a separate program to conserve memory in your main operating programs. This idea also gives you more control for preventing unauthorized deletions.

To delete a key and the data field associated with it, you first have to access it somehow. The usual way is a search. The search can be an exact or approximate search, or you can position to the key with read-next or read-previous routines. The important conditions are that you have got the key you want to delete, and that the stack level (TL), current node (TC), and current position (TX) variables are all valid. The level, TL, upon entry to a delete subroutine, must always be non-zero. The key to be deleted is given by TK$(TX). The data field that goes with it, TD$(TX) is also deleted.

Let's first look at the simple situation of a B-tree that has no master file associated with it. Here's how the logic might be organized:

```
LINEINPUT "KEY: ";KY$              'Request key to be deleted
GOSUB 591ØØ                        'Search for it
IF TQ=Ø THEN GOTO .....            'Abort if not found
PRINT TD$(TX)                      'Print data for verification
LINEINPUT"DELETE IT?";A$           'Let operator confirm
IF A$<>"Y" THEN GOTO .....         'Abort deletion if not yes
GOSUB 595ØØ                        'Call deletion subroutine
GOSUB 59Ø5Ø                        'Write new statistics to disk.
PRINT"DELETED!"                    'Report that deletion was done
```

When your TREESAM B-tree is an index to another file, the logic is the same, but you also have to perform a deletion on the record in the master file. This can be done before or after the key is deleted from the B-tree.

The case of two or more B-trees that access the same master file is more complex because you have to make your deletion in three or more places. Assuming that you've accessed the master file from one B-tree, you then have to position to, and delete the applicable keys in the other trees. Then you can delete the master file record and the primary key. We talked about multiple B-trees indexing the same master file earlier.

As with add commands, a delete command will leave you at an unpredictable position in the tree upon completion because node merging may have taken place. Read-next and read-previous operations are invalid immediately following a delete. (Read-first and read-last are okay.) This suggests a problem. How can you automatically delete a key and all its duplicates in a non-unique tree? The solution is, after the first key is deleted, to make the program repeat its search for the same key. If it is found again, delete it, and repeat until a search for the same key yields a not-found condition.

Changing a Data Field

There will be occasions when you'll want to leave a key as-is in the tree, but change the data field stored with it. If the data field is a pointer, perhaps you will want to make a change so it points to a different record, or to no record at all. If the data field contains other information, perhaps you will just want to update it or to make a correction.

To make a change to a data field, the first step is to find it. You can find it with a search, read-next or read-previous operation. Once you do, TD$(TX) will contain the data field to be changed, TF will be the file number, and TC will be the node. You simply LSET or RSET the new string into TD$(TX) and do a PUT of the current node to record it to disk.

As an example, let's say you've done a search for the key "X3453". When it's found, the data field TD$(TX) contains the string "IN STOCK", and you want to change it to "ON ORDER". The program commands are:

```
LSET TD$(TX)="ON ORDER"
PUT TF,TC
```

To illustrate another case, suppose the data field contains a two-byte pointer. The program sees that CVI(TD$(TX)) is 3453, and you want to change it to the number stored in the variable RN%. To make the change the program executes:

```
LSET TD$(TX)=MKI$(RN%)
PUT TF,TC
```

Remember, you can't change a key stored in TK$(TX) this way because there's a possibility that you'll alter the sequence of the keys in the tree. That could cause subsequent searches to fail. Also, you must be sure to use LSET or RSET. If you just say TD$(TX)=MKI$(RN%), you will destroy the fielding of the buffer, and your new data won't be recorded to disk.

Changing a Key

Changing a key is a bit more complex. Let's say, for example, that "X103A" is a new product number that replaces "M3424". As you can see, the new number is likely to put the key at an entirely different position in the tree. To make the change, you have to delete the old key and add the new one to the B-tree. If you select a variable to temporarily hold the data field associated with the key to be changed, you can make the operation entirely automatic.

If the tree is a unique-key tree, the program must first verify that the new key is not already in the file. It does this with the search subroutine. If the new key is not already in the tree, the program comes back to the old key, stores the data field in a temporary variable, deletes it, and then adds the new key, using the same data field. To make the operation a little faster, you can, for a short instant during the process, allow both the old and new keys be in the tree. After both are there, you can return to the old one and delete it.

After a key change, a call to subroutine 59050 should be done to write the new statistics to disk. There is a possiblity that the root node has been changed.

Using More Than One Index

Many program applications call for the use of more than one index. In an invoicing program, for example, you might want to access customers by account number as well as inventory product prices and descriptions by product number. In a carpet inventory system you might want to access rolls of carpet by roll number, as well as by description. There are several ways to handle these requirements with TREESAM.

One method is to have more than one "logical" index in the same "physical" index. In other words, the same B-tree might contain the keys for two different master files. The invoicing program application could probably use such an approach. There are two requirements. First, the keys for one logical index must be about the same length as the keys for the other. Second, the keys must be distinguishable by the program so it knows what group a particular key belongs to. This can be accomplished in a few different ways:

Perhaps the keys are already distinguishable and a program routine can tell the difference. Product numbers might start with a letter, while customer numbers start with a digit, for instance.

The program could make the keys for a particular logical index different by putting a code, invisible to the operator, in front of each key. It might put the letter "C" in front of each customer number, and the letter "P" in front of each product number. Any time a key is displayed, the program would strip the first letter off. This method takes an extra byte per key, but given 256 possible values for that byte, you could have 256 indices in one. If you are "disk rich" and "memory poor" this can be a good solution.

A variation on the same method, but without the need for an extra byte per key, is to have your program transform the first byte of each key for one of the logical indices in the same physical index by adding a value to it. If you know that the first character will always be a letter or a number, you can, for example, add 128 to the ASCII value of the first character for tree storage, and subtracting it out for display and printing. Here's how you do it, given a customer number and product number index stored as one:

Before search and add operations:

- If the key is a customer number, do nothing.
- If the key is a product number,

```
MID$(KY$,1,1)=CHR$(ASC(KY$)+128)
```

Before any operation that displays or prints a key from the tree:

- If ASCII(TK$(TX))<128 then it's a customer number. No special transformation is necessary. Otherwise it's a product number. To display or print it as A$,

```
A$=CHR$(ASC(TK$(TX))-128)+MID$(TK$(TX),2)
```

- Another way is to code the data field stored with each key so the program can tell which logical index you are working with. In the case of two logical indices, in which the data field is a pointer, the pointers for one set of keys could be stored negative. The pointers for the other index in the same B-tree would be stored positive.

I've highlighted the different ways to store more than one index in the same B-tree because these methods often result in a faster and more disk and memory-efficient program than the alternatives. These methods are limited, though to indices sharing similar key lengths, unique/non-unique characteristics, and so forth.

When you want to have two or more distinct B-trees in the same program, each B-tree uses a different file buffer. The B-tree for the customer number index might be file 3, while the tree for the product index might be file 5. Your program must open each one when a session starts. The operation of the TREESAM subroutine calls is changed slightly:

- As your program goes from one index to another, TF must be changed to indicate the file containing the B-tree you want to use. In the example, TF is set to 3 before each use of the customer index and is changed to TF=5 before each use of the product index.

- A program session must start by initializing the B-tree for the index that stores the most keys per node. Let's, as an illustration, say that your product index stores 30 keys per node, and your customer index stores 20. The program session uses the following startup sequence after both files are open:

```
TF=5                          'Select product index
GOSUB 59ØØØ                   'Get statistics
GOSUB 59Ø1Ø                   'Dimension and field TREESAM arrays
```

The reason for initializing the B-tree having the most keys per node first is that line 59010 does the dimensioning. You can only dimension an array once in a program session, so you want to be able to handle the maximum you'll require.

Now, anytime you want to use TREESAM in the program, you can use a simple subroutine that loads the statistics for the index you want to access. Consider the following, where the program loads TB with the number of the desired index before each TREESAM call:

```
IF TF=TB THEN RETURN          'No switch necessary if desired
                               index is already active.
TF=TB                         'Specify desired index.
GOSUB 59ØØØ                   'Reload statistics
GOSUB 59Ø11                   'Refield the node buffer
TL=Ø                          'Clear the stack
RETURN                        'Return to main program
```

Let's say that you've stored the subroutine listed above at line 59700. Now before a customer number search, you would say:

```
TB=3                          'Select customer index
GOSUB 597ØØ                   'Switch if necessary
GOSUB 591ØØ                   'Go ahead and do your search
```

If, on the other hand, you wanted to select the product index, you would use TB=5.

If read-next, read-previous, add, or delete operations are required after the search, no call to 59700 is necessary as long as you are working with the same index. If you are doing additions or deletions in a program that uses multiple trees, be sure you write the new statistics to disk (by doing a GOSUB 59050) after each addition or deletion, or at least, before you go from one B-tree to another.

This multiple index method is fairly simple, but a short pause of a second or so is noticeable as the new statistics are loaded and the file buffer is refielded when you switch from one index to another. The delay for refielding can be eliminated by modifying TREESAM so TM$ is a single dimension array, with the TP$, TK$ and TD$ arrays being doubly dimensioned.

The other limitation of the suggested method for handling more than one index in a program is that it doesn't let you keep your postion in one B-tree when you go to the other. To do that, you have to find some place to store the contents of the stack array, and the variables TX, TC and TL. Another array in memory is a possible answer, as a storage location for the current position statistics for all B-trees that are not currently selected.

Other ways to handle programs with multiple B-trees can be developed by modifying the TREESAM subroutines. You can, for example, modify TREESAM so each variable (other than the purely temporary working variables) is stored in an array which is dimensioned for the number of indices that will be required. (Several of the other handlers in this book use that approach.) This, of course, will make TREESAM more complex; it will require more memory, and a reduction in speed can be expected.

Deletions can be tricky in any program where more than one key indexes the same file. First, your program must have a way of knowing what the secondary keys are. The usual solution is to store them as fields in the master file. When a master file record is to be deleted, each key field is used to initiate a search and deletion in its related B-tree. If that B-tree is a non-unique key tree, the key to be deleted can be identified by comparing the pointer it stores with the record number of the master file record being deleted. After all secondary keys are deleted, your program can delete the master file record and the primary key.

Sometimes the best, and most easily programmed way to handle multiple index deletions is to periodically clear and rebuild the B-trees that contain the keys. In the interim, another step can be included in each search. Upon completion of a "successful" search using a secondary key, your program can compare the key against the appropriate field in the master file record index to make sure the search was really successful. This method lets you leave keys that are no longer applicable in the B-tree until it is re-built, or until a program is executed that goes through the tree looking for, and deleting obsolete keys.

TREESAM/BAS

TREESAM/BAS is a program that builds and accesses TREESAM files. As a training program, it lets you learn and test all the functions of TREESAM. As a utility program, it performs the following functions:

- It makes it easy for you to create TREESAM indices to be used with other application programs you may be developing.
- It allows you to automatically build a tree from pre-existing data that you may already have in one or more disk files.

- It can check the parameters and status of any TREESAM index.
- It provides ways to change parameters (key length, data length, unique or non-unique, etc.) for any TREESAM index.
- It gives ways to access all the functions (add, delete, change, etc.) provided by TREESAM.
- It shows you exactly how your TREESAM file is organized on disk (nodes, pointers, etc.). It provides ways to rebuild damaged TREESAM indices.
- It lists all keys and pointers from a TREESAM index in ascending or descending order.
- It can be used as a sorting program to help resequence disk files, or to create "sequence" files that tell another program what sequence it should print records from a file.

Loading TREESAM/BAS

TREESAM/BAS is a BASIC program. To use all its features, start from DOS Ready, specifying at least 2 files when you enter BASIC. Most of the functions of TREESAM/BAS, however, require only 1 file.

Creating an Index with TREESAM/BAS

Upon receiving the RUN command, TREESAM/BAS displays its initial menu:

```
TREESAM Utility Program
==============================================================

<C> CREATE A NEW TREESAM INDEX
<O> OPEN AN EXISTING TREESAM INDEX

==============================================================

PRESS THE KEY INDICATING YOUR SELECTION,
              OR PRESS <E> TO END THE PROGRAM...
```

Pressing <C> from the original menu takes you to the creation routine. You are asked a series of questions which you answer by entering the requested information. Any time you wish to go back to re-answer a previous question, you may press the up-arrow key and <ENTER>. (On the TRS-80 Model II, use <shift>-<left bracket>, and <ENTER>.)

Here's how the display might look after answering the questions:

```
INDEXING SYSTEM CREATION

<UP-ARROW> <ENTER> RETURNS YOU TO PREVIOUS QUESTION...

INDEX FILE NAME AND DRIVE:                  INDEX:Ø
MAXIMUM KEY LENGTH:                         8
DATA STRING LENGTH:                         2
IS DATA STRING A 2-BYTE POINTER? (Y=YES):   Y
MUST EACH KEY BE UNIQUE? (Y=YES):           Y
INDEX CAPACITY REQUIRED:                    999
KEYS PER NODE = 2Ø
```

```
BYTES PER NODE = 243
NODES REQUIRED = 1ØØ  + 1 FOR CONTROL INFORMATION = 1Ø1
WORST CASE SEARCH REQUIRES  3  DISK ACCESSES
CREATE THE INDEX NOW? (Y/N)  ..
```

In this example, you told the program you want to create an index file named "INDEX" on drive 0. The name can be anything, as long as you follow the rules for valid file name and drive specifications. You may want to be sure that the name you select doesn't already exist on the drive you specify. The creation process erases everything in the file so that a tree can be built. It may be necessary for you to go back to DOS Ready to check your directory. If the file already exists and you want to reuse it for an index, it's also a good idea to KILL it first. That insures that it will be created with the minimum length it will require.

The maximum key length can be anything from 1 to 124 bytes, depending on how you specify the data string length. In the example, you told the program that you wanted to be able to use 8-byte keys.

The data string length may be anything from 0 to 123 bytes, as long as the data string and key length combined don't exceed 124 bytes. You specified the data string length as 2 bytes. In this example, you intend to store an integer in each data string. Integers require 2 bytes in MKI$ format.

Next you were asked if the data string was to be a 2-byte pointer. This question is only asked when you specify a 2-byte data string length. The response tells the TREESAM/BAS program whether or not to use the MKI and CVI functions to handle the data in the data string associated with each key. For trees intended for use as indices to other random files, the answer to this question is usually "Y" for yes. In this case, you answered yes.

TREESAM/BAS then asks if you want to force each key in the tree to be unique. The alternative, of course, is to allow duplicate keys. On the sample screen, you answered <Y> for yes. You want the system to warn you if you attempt to add a key that already exists to the tree.

Next, you were asked to give the index capacity required. In other words, how many keys do you want the tree to hold. This question is asked so TREESAM/BAS can build a file big enough to hold all the keys you think you'll be needing. This is important to help avoid the possibility of getting an unexpected disk full error during an attempt to add a key to the tree. TREESAM and TREESAM/BAS do not limit you to the capacity entered here. You may, if you wish, enter a small number, such as 1, allowing the file to grow dynamically as required. In the sample display, you entered 999. You wanted to be sure that the disk could hold a tree with 999 keys in it.

After the creation questions have been answered, TREESAM/BAS displays some information. First it tells you the maximum number of keys that will be stored in each node. In this case it is 20. Knowing that each node will always be at least half full, you can conclude that each node, except the root, will always have between 10 and 20 keys.

For your convenience it also shows how many bytes will be used per node. In this case, 243 bytes will be used. This information is helpful if you want to make your TREESAM index a variable length record file. To handle the tree, you could indicate a record length of 243 upon opening the TREESAM file. It is important to note, though, that without modification, TREESAM/BAS only operates with fixed length record files. Each record is 256 bytes, whether or not all 256 are used.

The display then shows:

```
NODES REQUIRED = 1ØØ + 1 FOR CONTROL INFORMATION = 1Ø1
```

This computation is based on the key and data lengths you requested, along with the capacity requirement you specified. The 101 nodes is a worst-case requirement, occurring in the rare situation that the root would contain only one item, and all the other nodes would be only half full. The one node for "control information" is the statistics record, node 1 of the tree.

Finally, the display shows how many disk accesses a worst-case search will require, given the capacity requirement and the keys per node. The sample display shows that 3 disk accesses will be required to get down to the bottom level of the tree. This information is helpful because it tells you the minimum dimension that will be needed for the TS array. You can change line 59010 of the TREESAM handler to make the stack array big enough for the number of levels you'll require.

Once you've looked over the screen, you may elect to go ahead and create the tree you've defined. In that case you enter <Y> for yes. If you enter <N>, you are taken back to the first question on the screen. From that point you may re-enter your parameters, or you may enter <up-arrow> to abort the creation process. As you can see, the opportunity to enter <N> makes this part of the TREESAM/BAS program useful as a planning tool for doing computations about capacities and key and data lengths you are considering.

If <Y> is entered to create the index now, the screen shows the creation process as it is executed:

```
OPENING THE FILE...

WRITING RECORD NUMBER:   1Ø1

WRITING CONTROL INFORMATION

CLOSING THE FILE...

CREATION COMPLETE. PRESS <ENTER>...
```

During creation, TREESAM/BAS first opens the file you specified on the drive indicated. Then, starting from record 1, it creates the file by writing each record to disk. Each record is filled with 256 bytes of zeroes. As each is written, the record number flashes on the screen. Then it writes the parameters and initial statistics for the tree into record 1 of the file. Finally, the file is closed. This ensures that the length of file pointers are recorded into the disk directory. When you press <ENTER>, you are taken back to the initial menu.

It is possible to get a disk full error when TREESAM is creating the file, meaning that the disk you selected just doesn't have enough free space. If so, enter the following commands:

```
CLOSE
KILL FS$
RUN
```

This erases the file you were attempting to create from disk and restarts the program. You'll need to use a different diskette, or different creation parameters to build your tree.

Opening a TREESAM Index

To open a TREESAM index, you press <O> in response to the initial menu display. The display then shows:

```
INDEXING SYSTEM OPEN

UP-ARROW <ENTER> RETURNS YOU TO MENU...

NAME AND DRIVE OF INDEX TO BE OPENED: ..
```

At this point, enter the file specification for the TREESAM index you wish to open. If you've just done a create, enter the name of the file you created. To follow through the example, enter:

```
NAME AND DRIVE OF INDEX TO BE OPENED:  INDEX:Ø
```

The program displays:

```
LOOKING FOR IT...
LOADING INDEX STATISTICS...
FIELDING THE INDEX BUFFER...
```

Then, assuming it has opened the file successfully, it goes on to the main TREESAM/BAS menu. After displaying the message, "LOOKING FOR IT", the message, "NOT FOUND! PRESS <ENTER>", may be displayed. This means that the program can't find a file with the name you specified. Upon pressing <ENTER>, you are returned to the initial program menu.

It is important to note that TREESAM/BAS will use any file name you give it. It does no checking to see that the file it's opening is actually a TREESAM index. It expects you to know which files you've created as indices. If it opens a file that is not an index, the program is likely to abort with an error message.

The Main TREESAM/BAS Menu

Once you've opened the TREESAM file, the main menu is displayed. It is from this point that you may select all the main options in the program. To select an option, you just press the number indicated:

```
TREESAM UTILITY PROGRAM          "INDEX:Ø"
==============================================================

<1> DISPLAY INDEX STATISTICS
<2> SEARCH, ADD, READ, DELETE
<3> LIST ALL KEYS IN SEQUENCE
<4> LIST NODE CONTENTS
<5> OUTPUT TO SEQUENTIAL FILE
<6> INPUT FROM SEQUENTIAL FILE
<7> CLOSE AND END

==============================================================
PRESS THE KEY CORRESPONDING TO THE FUNCTION YOU WANT...
```

Notice that the main menu shows the name of the file you've opened. In the case of the example, it shows "INDEX:0" in the upper right corner of the display.

Displaying Statistics

Option 1 on the main menu lets you display the parameters and statistics for the TREESAM file you've opened. When you press <1>, the following display is shown:

```
CURRENT STATISTICS FOR "INDEX:Ø"

LENGTH OF FILE                  1Ø1
KEYS PER NODE / 2               1Ø
KEY LENGTH                      8
DATA LENGTH                     2
ITEM LENGTH                     12
ROOT NODE                       2
NEXT NODE, EOF                  3
LAST DELETED NODE               Ø
NODES ACTIVE                    1
KEYS ACTIVE                     Ø
OPTIONS                        UNIQUE KEYS WITH POINTERS

PRESS <ENTER>...
```

Pressing <ENTER> takes you back to the main menu, but let's look at what the display shows first. The one illustrated shows the paramaters and beginning statistics that the creation routine has loaded for the file "INDEX:0".

The "LENGTH OF FILE" just tells you how many physical records the TREESAM file is occupying on disk. The creation process gave this file a length of 101 records. Of course, all of them except the first are empty at this point.

"KEYS PER NODE / 2" shows you the half-node capacity in this tree. The tree you created stores a maximum of 20 keys per node. The half node capacity is 10. The TREESAM subroutines just find this number more convenient to work with in computations than the full node capacity of 20.

The "KEY LENGTH" and "DATA LENGTH" parameters are shown as you entered them when you created the file. The "ITEM LENGTH" parameter is equal to the key and data lengths, plus 2. The two extra bytes are for the internode pointers that TREESAM stores with each item.

Next you are shown the record number of the root node. In a new tree the root node always starts out as 2. As additions and deletions are made, the record number of the root node may change.

The "NEXT NODE, EOF" statistic tells you the record number of the next node that will be added, if the "LAST DELETED NODE" is 0. Regardless of the length of the file, TREESAM keeps its own "end of file" pointer. In the example, as for all new trees, the "NEXT NODE, EOF" statistic starts out as 3. Node 1 is the statistics record, node 2 is our empty root node, and node 3 is where the next node will be recorded when the tree expands.

The "LAST DELETED NODE" statistic always starts out as 0. If there are deletions that empty out a node so that it is no longer required, its number is stored here. When another node is deleted its number replaces the number of the last deleted node. TREESAM uses this information as a backward chain that can be followed to recover each deleted node, if any, before using the "NEXT NODE, EOF" statistic to extend the tree into new nodes.

The "NODES ACTIVE" statistic always starts out as 1. Each time a new node is linked into the tree, this number is incremented. In the special case of a new tree, there is 1 node active, but it contains no items.

The "KEYS ACTIVE" statistic tells you how many keys you've got in the tree. It always starts out as 0. It is incremented by 1 each time you add a key to the tree, and decremented by 1 each time you delete a key.

Finally, a description of the options is displayed. The options indicated will depend on the way you answered the questions when the tree was created. The possible options are:

```
UNIQUE KEYS WITH POINTERS
UNIQUE KEYS WITH DATA
NON-UNIQUE KEYS WITH POINTERS
NON-UNIQUE KEYS WITH DATA
```

You specified a data length of 2 and told the program that those 2 bytes were to be used to store a pointer in 2-byte MKI$ format. You also specified that each key had to be unique, so the display shows:

```
UNIQUE KEYS WITH POINTERS
```

Search, Add, Read, Delete

Option 2 from the main menu takes you into the part of the program that exercises the different capabilities of the TREESAM handler. The display is cleared and the following command menu is displayed:

```
SE, SA             SEARCH  - EXACT, APPROXIMATE
AU, AF, AC         ADD     - UNIQUE, FIFO, CURRENT
RR, RN, RP         READ    - RESET, NEXT, PREVIOUS
DC                 DELETE  - CURRENT
CC                 CHANGE  - CURRENT
------------------ UP-ARROW RETURNS TO MENU -----------------

COMMAND: ..
```

TREESAM/BAS lets you select from ten 2-letter commands to perform operations on the tree. "SE" is "Search Exact", "SA" is "Search Approximate", "AU" is "Add Unique", and so forth. You tell the program your command by typing the two letters and pressing <ENTER> when the "COMMAND:" prompt is shown. Simply pressing <ENTER> repeats the prior command you specified. Pressing <up-arrow> and <ENTER>, (or for the Model II, shift left bracket, <ENTER>), takes you back to the main menu. Any other entry, such as <?>, re-displays the command menu.

To show how it works, let's add a few keys to the tree. You'll recall that to add a key to a unique-key tree, you must first do a search. Then if it is not found, you can add it. Let's add the key, "JOE". First, do the search:

```
COMMAND: SE
SEARCH EXACT            KEY:     JOE
                        ** NOT FOUND **
```

In response to the "KEY:" prompt, you enter "JOE". The display shows "** NOT FOUND **". Since this is a new tree, the key obviously isn't in it, but the search has done the important task of finding the node and position where it should be added. The command to add the key you've just searched is "AC", for "Add Current". You want to add the new key at the current position.

```
COMMAND: AC
ADD CURRENT             KEY:     JOE
                        POINTER: 1
```

The "Add Current" command re-displays the key that you used for the search and requests the pointer to be stored with the key. Let's say you want the number "1" to be stored with the key, "JOE". We enter "1" and "JOE" is added to the tree.

Let's stop here for a moment to emphasize the purpose of TREESAM/BAS. If you are using TREESAM as an index to a file, the TREESAM handler subroutines normally will be imbedded within your program and the pointer will be handled automatically, invisible to the operator at the keyboard. In TREESAM/BAS, you do these functions "manually", just to see how the system handles the keys and numbers. Once you have your program written, you'll only use this function of TREESAM/BAS occasionally. Perhaps a few keys have gotten lost due to a power failure or some other malfunction. You can check out your tree and re-enter keys manually if necessary with TREESAM/BAS.

Now that you've added "JOE", try a search to see if it's in the tree:

```
COMMAND: SE
SEARCH EXACT            KEY:     JOE
                        POINTER: 1
```

There it is! TREESAM found the key, and reports that "1" is the pointer stored with it. Now add a few other keys. To do so, try another command, "AU", or "Add Unique".

"Add Unique" combines the "SE" and "AC" functions into one step. You supply the key and pointer you want to store, and if the key is not already in the tree, TREESAM/BAS adds it. The keys and pointers you want to add are:

```
MARY,2
ADAM,3
DON,4
```

Here's the display:

```
COMMAND: AU
ADD UNIQUE              KEY:     MARY
                        POINTER: 2
COMMAND: AU
ADD UNIQUE              KEY:     ADAM
                        POINTER: 3
COMMAND: AU
ADD UNIQUE              KEY:     DON
                        POINTER: 4
```

Now let's see what happens if you attempt to add "ADAM" again:

```
COMMAND: AU
ADD UNIQUE              KEY:     ADAM
                        POINTER: 5
                        ** DUPLICATE EXISTS - NOT ADDED **
```

Since you've set up the tree as a unique-key tree, "ADAM" is rejected the second time you try to add it. It is also rejected if you attempt to do a "SE" and an "AC" to add "ADAM". TREESAM/BAS doesn't permit an "Add Current" immediately following a successful search.

Now look at what you have got in the tree. The "RR" command, "Read Reset", resets the position to the beginning of the tree. (It does this by changing the level indicator to 0.) Immediately after a "Read Reset", a "Read Next" command is interpreted as "Read First". A "Read Previous" command is taken to mean "read the last key in the tree". To read through the tree in ascending sequence, you first enter "RR":

```
COMMAND: RR
READ RESET             * DONE *
```

The display shows "* DONE *" to indicate that it has reset to the beginning of the tree.

Now you can repeatedly enter "RN", or "Read Next" commands, to display each key and pointer in the tree. After the first "RN", though, it is easier to just press <ENTER>. Remember that <ENTER> tells TREESAM/BAS to repeat the last command you issued. Here's the display as you read through the tree:

```
COMMAND: RN
READ NEXT              KEY:     ADAM
                       POINTER: 3
COMMAND: RN
READ NEXT              KEY:     DON
                       POINTER: 4
COMMAND: RN
READ NEXT              KEY:     JOE
                       POINTER: 1
COMMAND: RN
READ NEXT              KEY:     MARY
                       POINTER: 2
COMMAND: RN
READ NEXT              * END OF FILE *
```

Notice that TREESAM/BAS returned the keys, not in the sequence you entered them, but in ascending sequence, from "ADAM" to "MARY". When you issued a "RN" after the last key was displayed, the "END OF FILE" message appeared. Now, if you enter "RN" again, the program starts over from the beginning of the file. "ADAM" and its pointer is displayed. Repeated "RN" commands read through the file in a circular fashion.

The "RP" command is "Read Previous". It returns the first key that is smaller than the current key. If you start with a "RR", you can read through the tree backwards:

```
COMMAND: RR
READ RESET             * DONE *
COMMAND: RP
READ PREVIOUS          KEY:     MARY
                       POINTER: 2
COMMAND: RP
READ PREVIOUS          KEY:     JOE
                       POINTER: 1
COMMAND: RP
READ PREVIOUS          KEY:     DON
                       POINTER: 4
COMMAND: RP
READ PREVIOUS          KEY:     ADAM
                       POINTER: 3
```

```
COMMAND: RP
READ PREVIOUS           * BEGINNING OF FILE *
```

You finally stop at the beginning of the file. The next "RP" command you give starts back at the end again. Whether you are going in ascending or descending sequence, TREESAM/BAS displays the keys in the tree as if they were on a continuous loop. The "end of file" or "beginning of file" message shows where the "loop" is connected. The message is "end of file" if you are going forward, or "beginning of file" if you are going back. Actually, the two points are the same.

You might also notice that you can "Read Next" or "Read Previous" from any point. Suppose the last key displayed was "DON". We can "RN" to "JOE", and then "RP" back to "DON":

```
COMMAND: RN
READ NEXT               KEY:     DON
                        POINTER: 4
COMMAND: RN
READ NEXT               KEY:     JOE
                        POINTER: 1
COMMAND: RP
READ PREVIOUS           KEY:     DON
                        POINTER: 4
```

Now try the "Search Approximate" command. "SA" let's you enter a partial or approximate key, and returns the next key in the tree that is greater than or equal to the key you entered. For example, to find "JOE", you could enter "J":

```
COMMAND: SA
SEARCH APPROXIMATE      KEY:     J
                        KEY:     JOE
                        POINTER: 1
```

TREESAM/BAS reports back the key and pointer that the approximate search has found. If you just press <ENTER> for an approximate search, the first key in the tree is returned:

```
COMMAND: SA
SEARCH APPROXIMATE      KEY:
                        KEY:     ADAM
                        POINTER: 3
```

If you do an approximate search using a search key that is greater than any in the tree, TREESAM/BAS returns an end of file message:

```
COMMAND: SA
SEARCH APPROXIMATE      KEY:    TED
                        * END OF FILE *
```

After any search, whether it has been successful or not in finding the search key, you can use the "read next" and "read previous" commands. An "RP" command after a "search approximate" like the one above, resulting in an "End of File" message, returns the last key in the file:

```
COMMAND: RP
READ PREVIOUS           KEY:     MARY
                        POINTER: 2
```

The "DC" command is used to delete a key and its pointer from the tree. "DC" stands for "Delete Current". That implies that you have got to use some means to position to the key you want to delete. The usual way is to do a "search exact" to find the key, and then to do the delete. Let's delete "DON". Here are the steps as they appear on the screen:

```
COMMAND: SE
SEARCH EXACT            KEY:     DON
                        POINTER: 4
COMMAND: DC
DELETE CURRENT          * DONE *
```

Now, if you wish, you can verify that "DON" and its pointer are gone:

```
COMMAND: SE
SEARCH EXACT            KEY:     DON
                        * NOT FOUND *
```

A "Delete Current" may also follow a "read next", "read previous" or "search approximate", as long as the resulting position is not the end (or beginning) of the file. TREESAM/BAS watches you carefully to make sure you don't enter commands when they are clearly invalid. A "DC" following a "SE" that has resulted in a "not found", for example, displays the message:

```
* NOT PERMITTED *
```

To keep you out of trouble, TREESAM/BAS gives the "NOT PERMITTED" message when you attempt any of the following:

```
AC after anything but SE, not found
DC after DC or RR
DC after SE, not found
DC after SA, end of file
DC after RN, end of file
DC after RP, beginning of file
RN after DC or any add
RP after DC or any add
```

There are two other commands on the menu that we have not discussed. "AF", or "Add FIFO", applies to non-unique key files only, so we will talk about it later. We can look at "CC", or "Change Current", now.

"Change Current" changes the data that is associated with any key in the tree. In the example, the data field holds a number. To perform a "Change Current", you move to the key whose data you want to change, just as if you were going to make a deletion. You can use "SE", "SA", "RN" or "RP" to get to the desired key. Let's say you want to change the pointer that is stored for "MARY" to 25. Do the search first; then the change:

```
COMMAND: SE
SEARCH EXACT            KEY:     MARY
                        POINTER: 2
COMMAND: CC
CHANGE CURRENT          KEY:     MARY
                        POINTER: 2
                        CHG TO:  25
```

Now, anytime you do a search for "MARY", 25 is shown as the pointer.

Every entry in this section of TREESAM/BAS provides the opportunity for an escape. Just press <up-arrow> and <ENTER> instead of typing the requested data. After using TREESAM/BAS to perform search, add, read, delete and change operations, you can get back to the main menu by entering <up-arrow> in response to the "COMMAND:" prompt.

Listing Keys

Option 3 of the TREESAM/BAS main menu lets you list the keys that are in a tree. When you press <3>, the following message is shown:

```
SEQUENTIAL READ:

<F> FORWARD  <R> REVERSE  <UP-ARROW> RETURNS TO MENU
```

To list the keys in ascending sequence, you simply press <F>. The keys and the pointers stored with each are shown:

```
FORWARD SEQUENTIAL READ
ADAM      3
JOE       1
MARY      25
DONE
PRESS <ENTER>...
```

In this case you only had 3 keys in the tree, "ADAM", "JOE", and "MARY". Upon pressing <ENTER>, you return to the point where you can select a forward or reverse read. When you press <R>, the following is shown:

```
REVERSE SEQUENTIAL READ
MARY      25
JOE       1
ADAM      3
DONE
PRESS <ENTER>...
```

After you press <ENTER>, you can again select forward or reverse, or you can press <up-arrow> to return to the main menu.

The automatic forward and reverse read capability is important when you want to see what keys are in your tree. It is also a very important tool to use when you want to verify that all the internal pointers are valid in the tree. If any of the pointers are invalid because of a hardware malfunction or a programming bug, this operation is likely to catch the problem. In a forward sequential read, as each key is displayed, the program verifies that it is greater than or equal to the prior key. In a reverse sequential read, each key is verified to be less than or equal to the prior key displayed.

Listing Node Contents

Option 4 from the main TREESAM/BAS menu makes a printout of the entire tree as it is stored on disk. This option lets you study the way that TREESAM handles its pointers and node assignments. It is also a good diagnostic tool if there is a problem in your tree.

Upon pressing <4>, the message on the screen is:

```
PRESS <ENTER> WHEN PRINTER IS READY,
OR PRESS <UP-ARROW> TO RETURN TO THE MENU...
```

The <up-arrow> option is provided just in case you didn't intend to request a printout, or perhaps you've suddenly realized that you don't have a line printer connected. (Again, Model II users may press shift right bracket instead of up-arrow.)

The printout starts with node 2 and goes to the last active node in the tree. Deleted nodes are noted, along with information about how they are linked to prior deletions. The root node is highlighted by a "** ROOT **" message.

For each node, the printout shows how many items are active. Then it shows each active internode pointer, key, and data field in the node, just as they are arranged from left to right. The internode pointers (to a key's left and right son) are shown between "(" and ")" symbols. The data fields are bracketed by "<" and ">" symbols.

The simple tree that you built with the foregoing examples has only three keys, so it is stored in only one node. The printout you get is:

```
NODE 2                       3  ITEMS ** ROOT **
( Ø)ADAM    < 3>( Ø)JOE     < 1>( Ø)MARY    < 25>( Ø)
```

As you can see, this is the root node. Since the tree has only one level, the internode pointers are all 0. TREESAM/BAS knows that each data field in this particular tree stores a 2-byte string in MKI$ format, so it uses the CVI function to make the data for each key readable.

Option 4 from the main menu is more illustrative and useful when you've built a large tree. With the resulting printout you can "play computer" to see how TREESAM starts from the root node to find any key in the tree.

Output to a Sequential File

Option 5 from the main menu copies all the keys from a TREESAM tree into a sequential file. This capability has two main purposes. First, it is useful when you want to change the parameters for a tree. Suppose you want to change the key length from 8 bytes to 10. You use option 5 to output all the keys and their pointers to a sequential file. Then you create a new empty TREESAM index with the new parameters you want. Finally you use option 6, which we'll discuss, to re-add the keys from the sequential file, but this time they are being automatically added to a tree having new parameters.

Option 5 is also helpful when there is a problem in your tree. Perhaps you've discovered that some of the internode pointers have gotten mixed up. Maybe a disk I/O error has made one or more of the nodes unreadable, and you want to recover as much as you can. You use option 5 to read through the TREESAM file, node-by-node, outputting all the keys and their pointers to a sequential file. Then you create a new empty tree with the same parameters, and you use option 6 to put the keys back in, using the sequential file for automatic input.

Upon selecting option 5, the display clears and the following message is shown:

```
OUTPUT FILE NAME AND DRIVE: ..
```

Here's where you supply the filespec for the sequential file you want to output to. Let's say you want to use a file named "INDEX/SEQ" on drive 1. Enter the name and drive, and the display shows:

```
OUTPUT FILE NAME AND DRIVE:  INDEX/SEQ:1
OUTPUT KEYS ONLY, OR INCLUDE POINTERS?        (K=KEYS ONLY): ..
```

Now it is asking whether you want to put the keys and their pointers, or just the keys, into the sequential file. If you enter <K>, the file will include keys only. If you answer with anything other than <K>, the system outputs to the sequential file, each key, followed by each corresponding pointer. That is, the file will contain key, pointer, key, pointer, . . ., until all the keys and pointers are written.

For recovery and parameter-change operations you'll usually want to output keys and pointers. For some other operations you may just want the keys. An example is when you are using TREESAM/BAS as a program for sorting a large sequential file.

Let's assume for now that you answered the question by entering <P>, meaning that you want to output keys and pointers. Now the display shows:

```
OUTPUT FILE NAME AND DRIVE:  INDEX/SEQ:1
OUTPUT KEYS ONLY, OR INCLUDE POINTERS?        (K=KEYS ONLY): P
OUTPUT IN ASCENDING OR NODE-BY-NODE SEQUENCE? (A=ASCENDING): ..
```

The next question gives an option. You can output to the file in ascending sequence. TREESAM/BAS will start at the root and follow the pointers so the lowest key alphabetically is outputted first, and the highest is outputted last. You can select "node-by-node" sequence. With node-by-node sequence the program starts at node 2, outputs all the keys from that node, goes on to node 3, and continues until the last node has been outputted. This method results in a sequential file that probably won't be in ascending sequence, but it is useful in a recovery operation when you suspect that some of the internode pointers in the tree are invalid. In that case, a node-by-node output is the only alternative that will work.

Let's say you selected ascending sequence for the output file. Enter <A>, and the program summarizes your responses:

```
FILE=INDEX/SEQ
KEYS AND POINTERS
ASCENDING SEQUENCE

PRESS <ENTER> TO BEGIN OR <UP-ARROW> TO CANCEL...
```

If everything is shown as intended, just press <ENTER>, and TREESAM/BAS outputs the data to the sequential file as you requested, closes it, and returns to the main menu.

Before you continue, a few notes should be made about node-by-node output. Node-by-node output is faster than ascending output, so that's one reason why you might want to use it. The main reason, though, is to offload keys and pointers from a damaged or invalid tree. In such a case, one or more of the nodes may be unreadable because of a disk I/O error. The TREESAM/BAS program may abort with:

```
Disk I/O Error in 522Ø
```

The sector that TREESAM/BAS is attempting to GET is unreadable. To salvage the rest, you can try to continue with the next node. To do so, enter the command:

```
GOTO 525Ø
```

Sometimes, more than one node will be unreadable, so you may need to jump to 5250 several times to salvage everything possible.

Of course, the best way to recover from disk I/O errors is to have recent backup copies of your data disks. The recovery procedure is to be used as a last resort.

Input from a Sequential File

Selection 6 from the main TREESAM/BAS menu adds keys and pointers to a tree from a sequential file. There are three main purposes for this capability. First, it allows you to build a tree automatically from data that you may already have on disk. Instead of writing a special program, or re-entering the keys and data manually, you can put them in a sequential file and have TREESAM/BAS do it for you. This is sometimes valuable when you are first converting from your old accessing system to TREESAM.

The second main reason for inputting data from a sequential file is to reload a tree after you've changed its parameters. Option 5 gives you a way to offload the data. Option 6 gives you a way to bring it back.

The third reason is recovery. You saw how option 5 provided a way to salvage as much data as possible from a damaged tree. Option 6 reloads the tree after you've re-created it. In the case of a unique-key tree, you can attempt to reload data from as many different sources as you wish because the input routine automatically rejects duplicates. Thus, if you think you've lost some keys, you can create a sequential file from a backup copy of your tree and use it as input to the current tree also.

Option 6 is also a good way to automatically load a large amount of data into a tree when you want to test it. During development and debugging of TREESAM, I used input from a sequential file quite extensively to make sure TREESAM was working properly.

Options 5 and 6 together also give you a way to sort and/or merge large amounts of data that may be contained in sequential files.

Upon selecting <6> from the main menu, the display shows:

```
INPUT FROM SEQUENTIAL FILE
<UP-ARROW><ENTER> RETURNS YOU TO PREVIOUS QUESTION...

INPUT FILE NAME AND DRIVE: ..
```

Here's where you supply the name of the sequential file you'd like to input from. Let's say for now that you want to input from the same file you outputted to with option 5. You'd enter "INDEX/SEQ:1" as the input file name and drive.

Next the program asks:

```
DOES INPUT FILE HAVE KEYS ONLY?    (Y=YES):  ..
```

It is your responsibility to know whether the file you are inputting from has been stored with keys only, or as in the case of the prior example, stored with alternating keys and pointers. Anything other than a <Y> answer tells the system to expect a file that has keys and data. In this case the data is pointers, so answer <N>.

Now the system summarizes your answers for you:

```
FILE=INDEX/SEQ:1
KEYS AND POINTERS
```

```
PRESS <ENTER> TO BEGIN OR <UP-ARROW> TO CANCEL...
```

You have got a chance to abort and make corrections here, or if everthing is okay, you can press <ENTER>. The program looks for the input file, tells you if it can't find it, or opens it if it is present on the disk you've indicated. Then it inputs each item. When it's done, the file is closed, and you can return to the main menu.

In the case of a unique-key tree, this routine does an "Add Unique" for each key and pointer in the file. If duplicates exist, they are rejected. The input file does not need to be in ascending sequence, but it is very important that it be in the right format. TREESAM/BAS doesn't know whether you've given it the name of a file that is actually a sequential file containing keys and pointers. It is up to you to know.

In the case of a non-unique-key tree, the input from sequential file does an "Add FIFO" for each key and pointer it finds. Duplicates are not rejected, so you have got to watch what you're doing. We'll discuss "Add FIFO" later.

Close and End

Selection 7 on the main TREESAM/BAS menu closes the TREESAM file you are using and returns you to the initial menu where you may open or create a different file. From that menu you may press <E> to indicate you are finished using the program. "E" returns you to BASIC's ready mode.

TREESAM/BAS and Non-Unique Keys

TREESAM/BAS will create and access trees in which the keys are not necessarily unique if you simply answer one question differently during the creation process.

```
MUST EACH KEY BE UNIQUE?  (Y=YES):                N
```

You answer <N> to say that each key need not be unique. This causes the creation routine to set a bit flag in the statistics record of the tree.

When you open a non-unique-key tree, TREESAM/BAS notes the fact that duplicate keys are permitted. The operation of the entire program is the same, except the "Add FIFO" command is enabled. "Add FIFO" means "Add First In First Out". Let's say you want to add the key, "SAM", twice, each "SAM" key having a different pointer:

```
COMMAND: AF
ADD FIFO                    KEY:     SAM
                            POINTER: 30
COMMAND: AF
ADD FIFO                    KEY:     SAM
                            POINTER: 31
```

Now "SAM" is in the tree two times. When you do a search for "SAM", the first "SAM" in the tree is returned:

```
COMMAND: SE
SEARCH EXACT                KEY:     SAM
                            POINTER: 30
```

As you can see, TREESAM/BAS retrieved the "SAM" having a pointer of 30. That was the first one you inputted, hence the term, FIFO — First In First Out. To see if there are any other "SAM" keys, you can use "Read Next" after the initial search:

```
COMMAND: RN
READ NEXT                    KEY:     SAM
                             POINTER: 31
```

TREESAM/BAS returned the next key in the tree, which also happened to be "SAM", but with a pointer of 31.

You have to take a little more care with non-unique keys. For deletions and changes make sure that you are positioned to the correct duplicate of the key you want.

As you can see, the "AF" command is a one-step operation. The system doesn't check to see if a duplicate for your key already exists. In a non-unique key tree you can still use "AU", and the "SE", "AC" combination if you want to make sure the key you are adding is unique. You can precede your "AF" with "SE" and one or more "RN" commands to see the duplicates that already exist in the tree before adding a new one.

Using Data Rather Than Pointers

The examples thus far have illustrated how to use TREESAM/BAS to handle trees with 2-byte integer "pointers" in the data fields associated with each key. Upon creation of the tree you can specify that a string of any valid length up to 123 bytes be stored with each key. This is specified by your response to:

```
DATA STRING LENGTH:
```

If you enter any number other than 2, the creation routine assumes that you are not using the data fields to store pointers. If you enter 2 for your data string length, it asks you:

```
IS DATA STRING A 2-BYTE POINTER? (Y=YES):
```

Enter <N> here if your data string is to be an ordinary 2-byte string, not in MKI$ format.

Once your tree has been created with these responses, the options flag in the statistics record will indicate that the data is not a pointer. Every place where the word "POINTER" is displayed in TREESAM/BAS now shows the word "DATA". Now, instead of being limited to numbers, you can type anything in response to the "DATA" prompt.

The data option makes TREESAM/BAS useful for an endless variety of stand-alone applications, with little or no modification. You can use it to store names and phone numbers, license numbers and names, check numbers and their amounts, program names and their descriptions, or just about anything else you can think of.

It is also important to notice that you can specify a data string length of zero. In such a case, the tree you build will contain keys only. In response to any prompts for data you just press <ENTER>. This idea can be very useful for sorting information. It can also be used when, for example, you just want to keep an alphabetical list of names to which you can make additions and deletions.

Programming TREESAM/BAS

The program listing for TREESAM/BAS is shown on the next few pages. Type it in exactly as shown. Then you need to type-in or merge the TREESAM handler subroutines that occupy lines 59000 through 59600. They were listed and explained earlier in Chapter 12.

Figure 13.1 — *TREESAM/BAS*

```
Ø 'TREESAM/BAS
1 CLEAR2ØØØ:DEFINTA-Z
2 TH=Ø:TX=Ø:KY$="":TY=Ø:TW=Ø:TC=Ø:TM$="":TL=Ø:TF=Ø:TK=Ø:TW$="":T
A$="":TS$="":TE$="":TN=Ø:TQ=Ø:TR=Ø:TZ=Ø

5Ø CLS:PRINT:PRINT"TREESAM UTILITY PROGRAM":PRINTSTRING$(63,"=")
51 PRINT
52 PRINT"<C> CREATE A NEW TREESAM INDEX"
53 PRINT"<O> OPEN AN EXISTING TREESAM INDEX"
59 PRINT:PRINTSTRING$(63,"=")
6Ø PRINT:PRINT"PRESS THE KEY INDICATING YOUR SELECTION,":PRINTTA
B(14)"OR PRESS <E> TO END THE PROGRAM..."
65 GOSUB4Ø5ØØ:A%=INSTR("ECO",A$):IFA%=ØTHEN65ELSEONA%GOTO7Ø,5ØØ,
8ØØ
7Ø CLS:END

1ØØ CLS:PRINT:PRINT"TREESAM UTILITY PROGRAM";TAB(32);CHR$(34);FS
$;CHR$(34):PRINTSTRING$(63,"=")
11Ø PRINT
111 PRINT"<1> DISPLAY INDEX STATISTICS"
112 PRINT"<2> SEARCH, ADD, READ, DELETE"
113 PRINT"<3> LIST ALL KEYS IN SEQUENCE"
114 PRINT"<4> LIST NODE CONTENTS"
115 PRINT"<5> OUTPUT TO SEQUENTIAL FILE"
116 PRINT"<6> INPUT FROM SEQUENTIAL FILE"
117 PRINT"<7> CLOSE AND END"
12Ø PRINT:PRINTSTRING$(63,"=")
13Ø PRINT:PRINT"PRESS THE KEY CORRESPONDING TO THE FUNCTION YOU
WANT..."
19Ø GOSUB4Ø5ØØ:A%=INSTR("1234567",A$):IFA%=ØTHEN19ØELSEONA%GOTO1
ØØØ,2ØØØ,3ØØØ,4ØØØ,5ØØØ,6ØØØ,7ØØØ

5ØØ 'CREATE A NEW INDEXING SYSTEM -----------------------------
51Ø CLS:PRINT"INDEXING SYSTEM CREATION":PRINT
511 PRINT"UP-ARROW <ENTER> RETURNS YOU TO PREVIOUS QUESTION...":
PRINT
52Ø PRINT"INDEX FILE NAME AND DRIVE: ";TAB(48):LINEINPUTA$:IFA$=
CHR$(91)THEN5ØELSEFS$=A$
53Ø PRINT"MAXIMUM KEY LENGTH:";TAB(48):LINEINPUTA$:IFA$=CHR$(91)
THEN52ØELSETK=VAL(A$)
54Ø PRINT"DATA STRING LENGTH:";TAB(48):LINEINPUTA$:IFA$=CHR$(91)
THEN53ØELSETD=VAL(A$):TI=TK+TD+2
541 TZ=Ø:IFTD=2THENPRINT"IS DATA STRING A 2-BYTE POINTER? (Y=YES
):";TAB(48):LINEINPUTA$:IFA$=CHR$(91)THEN54ØELSEIFA$="Y"THENTZ=1
542 PRINT"MUST EACH KEY BE UNIQUE? (Y=YES):";TAB(48):LINEINPUTA$
:IFA$=CHR$(91)THEN54ØELSEIFA$<>"Y"THENTZ=TZOR2
55Ø PRINT"INDEX CAPACITY REQUIRED:";TAB(48):LINEINPUTA$:IFA$=CHR
$(91)THEN54ØELSECT=VAL(A$)
```

```
6ØØ TN=INT(INT(253/TI)/2)
61Ø PRINT"KEYS PER NODE =";TN*2
63Ø PRINT"BYTES PER NODE =";TN*2*TI+3
64Ø NR=-INT(-((CT-(TN*2)/2)*2/(TN*2)+1))
65Ø PRINT"NODES REQUIRED =";NR;" + 1 FOR CONTROL INFORMATION =";
NR+1
66Ø IFTN>1THENPRINT"WORST CASE SEARCH REQUIRES ";-INT(-LOG(CT)/L
OG(TN+1));" DISK ACCESSES"

7ØØ LINEINPUT"CREATE THE INDEX NOW? (Y/N) ";A$:IFA$=CHR$(91)THEN
55ØELSEIFA$="N"THEN5ØØELSEIFA$<>"Y"THEN7ØØ
71Ø CLS:PRINT"OPENING THE FILE...":TF=1:OPEN"R",TF,FS$
72Ø FIELDTF,128ASA$:LSETA$=STRING$(128,Ø):FIELDTF,128ASA$,128ASA
$:LSETA$=STRING$(128,Ø)
73Ø PRINT@128,"WRITING RECORD NUMBER: ";
731 FORTY=1TONR+1:PRINT@128+24,TY
732 PUTTF,TY
733 NEXT
74Ø PRINT@256,"WRITING CONTROL INFORMATION"
741 GOSUB59ØØ5
75Ø LSETTN$=MKI$(TN):LSETTK$=MKI$(TK):LSETTD$=MKI$(TD):LSETTI$=M
KI$(TI):LSETTZ$=MKI$(TZ):LSETTR$=MKI$(2):LSETT1$=MKI$(3):LSETT2$
=MKI$(Ø):LSETT3$=MKI$(1):LSETT4$=MKI$(Ø)
76Ø TC=1:PUTTF,TC
77Ø PRINT@384,"CLOSING THE FILE...":CLOSETF
78Ø PRINT@512,"CREATION COMPLETE.  PRESS <ENTER>..."
781 LINEINPUTA$:GOTO5Ø

8ØØ 'OPEN AN EXISTING INDEXING SYSTEM -------------------------
81Ø ONERRORGOTOØ:CLS:PRINT"INDEXING SYSTEM OPEN":PRINT
811 PRINT"UP-ARROW <ENTER> RETURNS YOU TO MENU...":PRINT
82Ø LINEINPUT"NAME AND DRIVE OF INDEX TO BE OPENED:";A$:IFA$=CH
R$(91)THEN5ØELSEFS$=A$:TF=1
83Ø PRINT"LOOKING FOR IT..."
831 ONERRORGOTO833:OPEN"I",TF,FS$
832 ONERRORGOTOØ:CLOSETF:OPEN"R",TF,FS$:GOTO84Ø
833 IFERR/2+1=54THENPRINT"NOT FOUND!  PRESS <ENTER>...":LINEINPU
TA$:RESUME8ØØELSE8ØØ
84Ø PRINT"LOADING INDEX STATISTICS...":GOSUB59ØØØ
85Ø PRINT"FIELDING THE INDEX BUFFER...":GOSUB59Ø1Ø
86Ø GOTO1ØØ

1ØØØ 'DISPLAY INDEX FILE STATISTICS --------------------------
1Ø1Ø CLS:PRINT"CURRENT STATISTICS FOR ";CHR$(34);FS$;CHR$(34):PR
INT
1Ø2Ø PRINT"LENGTH OF FILE";TAB(32);LOF(TF)
1Ø3Ø PRINT"KEYS PER NODE / 2";TAB(32);TN
1Ø4Ø PRINT"KEY LENGTH";TAB(32);TK
1Ø5Ø PRINT"DATA LENGTH";TAB(32);TD
1Ø6Ø PRINT"ITEM LENGTH";TAB(32);TI
1Ø7Ø PRINT"ROOT NODE";TAB(32);TR
1Ø8Ø PRINT"NEXT NODE, EOF";TAB(32);T1
```

```
1Ø9Ø PRINT"LAST DELETED NODE";TAB(32);T2
11ØØ PRINT"NODES ACTIVE";TAB(32);T3
111Ø PRINT"KEYS ACTIVE";TAB(32);T4
112Ø PRINT"OPTIONS";TAB(32);
1121 IFTZAND2THENPRINT"NON-UNIQUE KEYS";ELSEPRINT"UNIQUE KEYS";
1122 PRINT" WITH ";:IFTZAND1THENPRINT"POINTERS"ELSEPRINT"DATA"
12ØØ PRINT:PRINT"PRESS <ENTER>...";:GOSUB4Ø5ØØ:GOTO1ØØ

2ØØØ 'SEARCH, ADD, READ DELETE --------------------------------
2Ø1Ø CLS:CD$="":TL=Ø:LC$="  "
2Ø2Ø PRINT"SE, SA           SEARCH - EXACT, APPROXIMATE"
2Ø21 PRINT"AU, AF, AC       ADD    - UNIQUE, FIFO, CURRENT"
2Ø22 PRINT"RR, RN, RP       READ   - RESET, NEXT, PREVIOUS"
2Ø23 PRINT"DC               DELETE - CURRENT"
2Ø24 PRINT"CC               CHANGE - CURRENT"
2Ø25 PRINT"--------------- UP-ARROW RETURNS TO MENU ---------"
2Ø3Ø LINEINPUT"COMMAND: ";A$:IFA$=CHR$(91)THEN1ØØELSEIFA$=""THEN
PRINTCD$:A$=CD$
2Ø31 A%=INSTR("SE,SA,AU,AF,AC,RR,RN,RP,DC,CC",A$):IF(A%=Ø)OR(LEN
(A$)<>2)THENPRINT"INVALID COMMAND!":GOTO2Ø2ØELSECD$=A$:ON(A%-1)/
3+1GOSUB21ØØ,211Ø,212Ø,213Ø,214Ø,215Ø,216Ø,217Ø,218Ø,219Ø
2Ø4Ø GOTO2Ø3Ø
2Ø99 '----------------------------------------------------

21ØØ PRINT"SEARCH EXACT";:PRINTTAB(25):LINEINPUT"KEY:      ";A$:I
FA$=CHR$(91)THENRETURNELSEKY$=A$:LC$="  "
21Ø1 GOSUB591ØØ
21Ø2 IFTQ=ØTHENPRINTTAB(25)"** NOT FOUND **":LC$="*N":RETURN
21Ø3 PRINTTAB(25):IFTZAND1THENPRINT"POINTER:";CVI(TD$(TX))ELSEPR
INT"DATA:    ";TD$(TX)
21Ø4 LC$="*F":RETURN
21Ø9 '----------------------------------------------------

211Ø PRINT"SEARCH APPROXIMATE";:PRINTTAB(25):LINEINPUT"KEY:
";A$:IFA$=CHR$(91)THENRETURNELSEKY$=A$:LC$="  "
2111 GOSUB591ØØ:IFTX>ASC(TM$)THENTX=TX-1:GOSUB593ØØ:IFTQ=ØTHENPR
INTTAB(25)"* END OF FILE *":RETURN
2112 PRINTTAB(25)"KEY:      ";TK$(TX)
2113 PRINTTAB(25):IFTZAND1THENPRINT"POINTER:";CVI(TD$(TX))ELSEPR
INT"DATA:    ";TD$(TX)
2114 LC$="*F":TQ=-1:RETURN
2119 '----------------------------------------------------

212Ø PRINT"ADD UNIQUE";:PRINTTAB(25):LINEINPUT"KEY:      ";A$:IFA
$=CHR$(91)THENRETURNELSEKY$=A$:LC$="  "
2121 PRINTTAB(25):IFTZAND1THENPRINT"POINTER: ";ELSEPRINT"DATA:
  ";
2122 LINEINPUTA$:IFA$=CHR$(91)THEN212ØELSEIFTZAND1THENDA$=MKI$(V
AL(A$))ELSEDA$=A$
2123 GOSUB591ØØ:IFTQ<>ØTHENPRINTTAB(25)"** DUPLICATE EXISTS - NO
T ADDED **":RETURN
2124 GOSUB592ØØ:GOSUB59Ø5Ø:LC$="AU":RETURN
2129 '----------------------------------------------------
```

```
213Ø PRINT"ADD FIFO";:PRINTTAB(25):IF(TZAND2)=ØTHENPRINT"* NOT P
ERMITTED *":RETURN
2131 LINEINPUT"KEY:      ";A$:IFA$=CHR$(91)THENRETURNELSEKY$=A$:L
C$="  "
2132 PRINTTAB(25):IFTZAND1THENPRINT"POINTER: ";ELSEPRINT"DATA:
  ";
2133 LINEINPUTA$:IFA$=CHR$(91)THEN213ØELSEIFTZAND1THENDA$=MKI$(V
AL(A$))ELSEDA$=A$
2134 GOSUB596ØØ:GOSUB59Ø5Ø:LC$="AF":RETURN
2139 '------------------------------------------------------

214Ø PRINT"ADD CURRENT";TAB(25):IFLC$="*N"THENPRINT"KEY:      ";K
Y$ELSEPRINT"* NOT PERMITTED *":RETURN
2141 PRINTTAB(25):IFTZAND1THENPRINT"POINTER: ";ELSEPRINT"DATA:
  ";
2142 LINEINPUTA$:IFA$=CHR$(91)THENRETURNELSEIFTZAND1THENDA$=MKI$
(VAL(A$))ELSEDA$=A$
2143 GOSUB592ØØ:GOSUB59Ø5Ø:LC$="AC":RETURN
2149 '------------------------------------------------------

215Ø PRINT"READ RESET";TAB(25):TL=Ø:PRINT"* DONE *":LC$="  ":RET
URN
2159 '------------------------------------------------------

216Ø PRINT"READ NEXT";TAB(25):IFINSTR("AD",LEFT$(LC$,1))ORLC$="*
N"THENPRINT"* NOT PERMITTED *":RETURN
2161 GOSUB593ØØ:LC$="RN":IFTQ=ØTHENPRINT"* END OF FILE *":RETURN
2162 PRINT"KEY:      ";TK$(TX)
2163 PRINTTAB(25):IFTZAND1THENPRINT"POINTER:";CVI(TD$(TX))ELSEPR
INT"DATA:     ";TD$(TX)
2164 RETURN
2169 '------------------------------------------------------

217Ø PRINT"READ PREVIOUS";TAB(25):IFINSTR("AD",LEFT$(LC$,1))ORLC
$="*N"THENPRINT"* NOT PERMITTED *":RETURN
2171 GOSUB594ØØ:LC$="RP":IFTQ=ØTHENPRINT"* BEGINNING OF FILE *":
RETURN
2172 PRINT"KEY:      ";TK$(TX)
2173 PRINTTAB(25):IFTZAND1THENPRINT"POINTER:";CVI(TD$(TX))ELSEPR
INT"DATA:     ";TD$(TX)
2174 RETURN
2179 '------------------------------------------------------

218Ø PRINT"DELETE CURRENT";TAB(25):IF(INSTR("RN,RP,*F",LC$)=Ø)OR
(TQ=Ø)THENPRINT"* NOT PERMITTED *":RETURN
2181 GOSUB595ØØ:GOSUB59Ø5Ø:PRINTTAB(25)"* DONE *":LC$="DC":RETUR
N
2189 '------------------------------------------------------

219Ø PRINT"CHANGE CURRENT";TAB(25):IF(INSTR("RN,RP,*F",LC$)=Ø)OR
(TQ=Ø)THENPRINT"* NOT PERMITTED *":RETURNELSEPRINT"KEY:      ";TK
$(TX)
```

```
2191 PRINTTAB(25):IFTZAND1THENPRINT"POINTER:";CVI(TD$(TX))ELSEPR
INT"DATA:    ";TD$(TX)
2192 PRINTTAB(25):LINEINPUT"CHG TO:  ";A$:IFA$=CHR$(91)THENRETUR
N
2193 IFTZAND1THENLSETTD$(TX)=MKI$(VAL(A$))ELSELSETTD$(TX)=A$
2194 PUTTF,TC:PRINTTAB(25)"* DONE *":RETURN
2199 '--------------------------------------------------------

3ØØØ 'READ ALL KEYS IN SEQUENCE
3Ø1Ø CLS:PRINT"SEQUENTIAL READ:":PRINT
3Ø11 PRINT"<F> FORWARD  <R> REVERSE  <UP-ARROW> RETURNS TO MENU"
3Ø2Ø GOSUB4Ø5ØØ:IFA$=CHR$(91)THEN1ØØELSEA%=INSTR("FR",A$):IFA%=Ø
THEN3Ø2ØELSEONA%GOSUB31ØØ,32ØØ
3Ø21 IFA$=CHR$(91)THENPRINT"ABORTED..."ELSEIFCT<>T4THENPRINT"COU
NT IS WRONG!"ELSEPRINT"DONE"
3Ø22 PRINT"PRESS <ENTER>...":GOSUB4Ø5ØØ:GOTO3ØØØ

31ØØ PRINT"FORWARD SEQUENTIAL READ":TL=Ø:LC$="":CT=Ø
311Ø GOSUB593ØØ:IFTQ=ØTHENA$="":RETURNELSECT=CT+1
3111 PRINTTK$(TX);"  ";:IFTZAND1THENPRINTCVI(TD$(TX))ELSEPRINTTD
$(TX)
3112 IFTK$(TX)<LC$THENPRINT"SEQUENCE ERROR!":A$=CHR$(91):RETURNE
LSELC$=TK$(TX)
3113 A$=INKEY$:IFA$=CHR$(91)THENRETURN
312Ø GOTO311Ø

32ØØ PRINT"REVERSE SEQUENTIAL READ":TL=Ø:LC$=STRING$(TK,255):CT=
Ø
321Ø GOSUB594ØØ:IFTQ=ØTHENA$="":RETURNELSECT=CT+1
3211 PRINTTK$(TX);"  ";:IFTZAND1THENPRINTCVI(TD$(TX))ELSEPRINTTD
$(TX)
3212 IFTK$(TX)>LC$THENPRINT"SEQUENCE ERROR!":A$=CHR$(91):RETURNE
LSELC$=TK$(TX)
3213 A$=INKEY$:IFA$=CHR$(91)THENRETURN
322Ø GOTO321Ø

4ØØØ 'LIST NODE CONTENTS ------------------------------------
4Ø1Ø CLS:PRINT"LIST NODE CONTENTS":IFT4=ØTHENPRINT"INDEX IS EMPT
Y...PRESS <ENTER>":GOSUB4Ø5ØØ:GOTO1ØØ
4Ø2Ø PRINT"PRESS <ENTER> WHEN PRINTER IS READY,":PRINT"OR PRESS
<UP-ARROW> TO RETURN TO THE MENU...":GOSUB4Ø5ØØ:IFA$=CHR$(91)THE
N1ØØ
4Ø3Ø FORTC=2TOT1-1:GETTF,TC:LPRINT"NODE";TC,,ASC(TM$);" ITEMS";:
IFASC(TM$)=ØTHENLPRINT" DELETED AND LINKED TO NODE";CVI(TP$(Ø)):
GOTO4Ø4Ø
4Ø31 IFTC=TRTHENLPRINT" ** ROOT **"ELSELPRINT" "
4Ø35 LPRINTUSING"(###)";CVI(TP$(Ø));
4Ø36 FORTX=1TOASC(TM$):LPRINTTK$(TX);:IFTZAND1THENLPRINTUSING"<#
##>";CVI(TD$(TX));ELSELPRINT"<";TD$(TX);">";
4Ø37 LPRINTUSING"(###)";CVI(TP$(TX));:NEXT:LPRINT" "
4Ø4Ø NEXT:GOTO1ØØ
```

```
5000 'OUTPUT TO SEQUENTIAL FILE ------------------------------
5010 CLS:PRINT"OUTPUT TO SEQUENTIAL FILE":PRINT"<UP-ARROW><ENTER
> RETURNS YOU TO PREVIOUS QUESTION...":PRINT
5020 LINEINPUT"OUTPUT FILE NAME AND DRIVE:  ";A$:IFA$=CHR$(91)TH
EN100ELSESF$=A$
5030 PRINT"OUTPUT KEYS ONLY, OR INCLUDE ";:IFTZAND1THENPRINT"POI
NTERS?";ELSEPRINT"DATA?    ";
5031 LINEINPUT"        (K=KEYS ONLY): ";A$:IFA$=CHR$(91)THEN5000
ELSECD$=A$
5050 LINEINPUT"OUTPUT IN ASCENDING OR NODE-BY-NODE SEQUENCE? (A=
ASCENDING): ";A$:IFA$=CHR$(91)THEN5030
5060 PRINT:PRINT"FILE=";SF$:IFCD$="K"THENPRINT"KEYS ONLY"ELSEPRI
NT"KEYS AND ";:IFTZAND1THENPRINT"POINTERS"ELSEPRINT"DATA"
5070 IFA$="A"THEN5100ELSE5200

5100 PRINT"ASCENDING SEQUENCE":PRINT:PRINT"PRESS <ENTER> TO BEGI
N OR <UP-ARROW> TO CANCEL...":GOSUB40500:IFA$=CHR$(91)THEN100
5110 PRINT"OPENING OUTPUT FILE...":OPEN"O",2,SF$
5120 TL=0
5130 GOSUB59300:IFTQ=0THEN5300
5140 PRINTTK$(TX);"  ";:PRINT#2,TK$(TX):IFCD$="K"THEN5150
5141 IFTZAND1THENA$=STR$(CVI(TD$(TX)))ELSEA$=TD$(TX)
5142 PRINTA$;:PRINT#2,A$
5150 PRINT:GOTO5130

5200 PRINT"NODE-BY-NODE SEQUENCE":PRINT:PRINT"PRESS <ENTER> TO B
EGIN OR <UP-ARROW> TO CANCEL...":GOSUB40500:IFA$=CHR$(91)THEN100
5210 PRINT"OPENING OUTPUT FILE...":OPEN"O",2,SF$
5220 FORTC=2TOT1-1:PRINT"** NODE";TC;" **":GETTF,TC
5230 IFASC(TM$)=0THEN5250
5240 FORTX=1TOASC(TM$):PRINTTK$(TX);"  ";:PRINT#2,TK$(TX):IFCD$=
"K"THEN5249
5241 IFTZAND1THENA$=STR$(CVI(TD$(TX)))ELSEA$=TD$(TX)
5242 PRINTA$;:PRINT#2,A$
5249 PRINT:NEXT
5250 NEXT

5300 PRINT"CLOSING OUTPUT FILE...":CLOSE2
5310 PRINT"DONE...PRESS <ENTER>...";:GOSUB40500:GOTO100

6000 'INPUT FROM SEQUENTIAL FILE ------------------------------
6010 ONERRORGOTO0:CLS:PRINT"INPUT FROM SEQUENTIAL FILE":PRINT"<U
P-ARROW><ENTER> RETURNS YOU TO PREVIOUS QUESTION...":PRINT
6020 LINEINPUT"INPUT FILE NAME AND DRIVE:  ";A$:IFA$=CHR$(91)THE
N100ELSESF$=A$
6030 LINEINPUT"DOES INPUT FILE HAVE KEYS ONLY?   (Y=YES): ";A$:I
FA$=CHR$(91)THEN6000ELSECD$=A$
6040 PRINT:PRINT"FILE=";SF$:IFCD$="Y"THENPRINT"KEYS ONLY"ELSEPRI
NT"KEYS AND ";:IFTZAND1THENPRINT"POINTERS"ELSEPRINT"DATA"
6050 PRINT:PRINT"PRESS <ENTER> TO BEGIN OR <UP-ARROW> TO CANCEL.
..":GOSUB40500:IFA$=CHR$(91)THEN100
```

```
6Ø6Ø PRINT"OPENING INPUT FILE...":ONERRORGOTO6Ø65:OPEN"I",2,SF$:
ONERRORGOTOØ:GOTO6Ø7Ø
6Ø65 IFERR/2+1=54THENPRINT"NOT FOUND!  PRESS <ENTER>...":GOSUB4Ø
5ØØ:RESUME6ØØØELSE6ØØØ
6Ø7Ø IFEOF(2)THEN61ØØELSELINE INPUT#2,KY$:PRINTKY$;"  ";
6Ø71 IFCD$<>"Y"THENLINE INPUT#2,A$:PRINTA$;ELSEA$=""
6Ø72 PRINT:IFTZAND1THENDA$=MKI$(VAL(A$))ELSEDA$=A$
6Ø8Ø IFTZAND2THENGOSUB596ØØ:GOTO6Ø9Ø
6Ø81 GOSUB591ØØ:IFTQ<>ØTHENPRINT"** DUPLICATE EXISTS - NOT ADDED
 **"ELSEGOSUB592ØØ
6Ø9Ø GOTO6Ø7Ø

61ØØ GOSUB59Ø5Ø:PRINT"CLOSING INPUT FILE...":CLOSE2
611Ø PRINT"DONE...PRESS <ENTER>...";:GOSUB4Ø5ØØ:GOTO1ØØ

7ØØØ 'CLOSE AND END ----------------------------------------
7Ø1Ø CLOSE:RUN

4Ø5ØØ A$=INKEY$:IFA$=""THEN4Ø5ØØELSERETURN

59ØØØ 'MERGE TREESAM SUBROUTINES 59ØØØ - 596ØØ HERE
```

Modifications to TREESAM/BAS

The modifications noted for the TREESAM handler subroutines can, of course, be applied to them as they exist in TREESAM/BAS. There are some other modifications you may want to make. I suggest you keep a copy of TREESAM/BAS as shown. Then you may want to make a few other versions, which you can name TREESAM1/BAS, TREESAM2/BAS, and so forth.

One modification that may be required is a change to handle a special key or data field format. Let's say, for example, that you are developing an application program in which you intend to use dates as keys. You have chosen to use the 2-byte compressed date format that is shown in *BASIC Faster and Better & Other Mysteries.* To handle this special type of key, you need to create a version of TREESAM/BAS that accepts the date keys from the operator in an 8-byte format, like "01/15/83", and then compresses them down to 2-bytes for storage. To make the change, you look for every line in TREESAM/BAS where a LINEINPUT statement requests the key from the operator. Once the key has been accepted, you need to call your functions or subroutines that compress it, before the TREESAM handler subroutines are called. Likewise, you need to insert logic to uncompress the key each time TREESAM/BAS prints or displays it. You'll also need to change sequential file input and output routines so that keys are stored in an uncompressed ASCII format when they are in the sequential file.

The same modifications apply to non-standard data string formats. Your program needs to make the conversion after the data is accepted from the operator and before it is put into the tree. When data is being displayed or printed, your program needs to convert back to a displayable or printable format.

If you wish, you can modify the create routine so that the options flag tells TREESAM/BAS what type of data you are working with. Notice how it determines whether the data is a 2-byte pointer or a normal string. "IF TZ AND 1" then it knows it is dealing with pointers. As shown, bits 1 and 2 of the options indicator, TZ, are used by TREESAM/BAS. You can use the other bits for your own purposes. *BASIC Faster and Better & Other Mysteries* gives you more information about how to use bit flags.

Another modification that you might find useful is to make the sequential file input routine perform other functions. As shown, it inputs keys and pointers to be added in "AF" or "AU" mode, depending on whether the tree holds unique or non-unique keys. You could modify the program so that you can use a sequential file to input a list of keys to be automatically searched and deleted. This would give you a "batch-mode" deletion capability.

Another change that is possible is to modify the sequential file output routine so that it can output a list of pointers, without their corresponding keys. This file containing pointers can be used as a "sequence" file which tells a report-printing program what order to print records from a random file.

TREESAM/BAS as shown is intended for use by readers of this book. Because of this, certain controls were left out that would make it error-proof for the normal computer operator. If you are developing programs that use TREESAM, and you want to provide TREESAM/BAS as a program for initial creation and emergency recoveries, you may want to delete some of the capabilities, and put more error handling safeguards in to make it safe for an inexperienced operator to use.

Model II Modifications to TREESAM/BAS

TREESAM/BAS was written to be Model II compatible. There are only a few changes required. First, the PRINT@ positions must be changed in lines 730 through 780:

```
Change PRINT@128 to PRINT@16Ø
Change PRINT@256 to PRINT@32Ø
Change PRINT@284 to PRINT@48Ø
Change PRINT@512 to PRINT@64Ø
```

The error handling must be changed in lines 833 and 6065.

```
Change "ERR/2+1=54" to "ERR=53"
```

You may also want to change "UP-ARROW" references to "SHIFT-LEFT-BRACKET". These are the keys that generate CHR$(91). If you'd like to substitute a more convenient key, you can take another approach. Change all references to CHR$(91).

KEYACCES — Faster and Better Accessing

KEYACCES is a Z-80 machine language subroutine for TRS-80 disk systems that your BASIC program can call for high performance keyed accessing of disk files. Here are some of the features of KEYACCES:

- Disk file logical records may be quickly accessed by one or more search keys. A key may be any string of from 1 to 255 bytes in length.
- The KEYACCES system is not memory dependent. You can create as many indices as you wish, each containing up to 21759 keys of up to 255 bytes each. The memory requirement is only 629 bytes for the machine language logic plus the space required for a few basic program lines. Your only practical limitation is disk capacity, but unlike other accessing systems, there is very little overhead required in disk storage. KEYACCES only requires a three-byte work area on disk for each key in the file.
- Accessing speed is very fast. For 1000 keys, using a TRS-80 Model I with standard Radio Shack minidisk drives, average access times are slightly over a half second. On a test of 5000 unique 10-byte keys, the average access time was 1 second, with a worst-case time of about 1.5 seconds. Access times on hard disk systems are much faster. The system is designed so that there is only a slight degradation in performance for very large files (due to the fact that the disk drive head must travel further).
- The file which you are indexing, the "master file", may be on a different disk drive from the index file used by the KEYACCES system. If you wish, the index file may be contained within the master file. The master file may be in any sequence, with logical records of any length up to 256 bytes. The keys may be at any position within each logical record of the master file, or you may set up a separate file to hold the keys.
- You may allow for unique or non-unique keys. With unique keys, each logical record in the master file has a key that is different from the keys used to access each other logical record. With non-unique keys, an unlimited number of logical records in the master file may share the same key. Each record sharing a key may be recalled in the sequence in which it was added to the file.
- Your program has complete control over the logical record numbers used in a master file, and it may be used independently from the indexing system. You may create indices "after-the-fact" for keyed access to master files that already exist.

- The KEYACCES system interactively handles additions and deletions of keys. No execution delays are required for "reorganization" of the file.
- KEYACCES is written as a relocatable Z-80 machine language subroutine. You can store at any location in protected memory.

Hashing

Hashing is a common approach to disk file organization that sets up a file so that it can be searched quickly, based on a search key. To understand how hashing works, pretend you have a disk file composed of logical records that have two fields each. The first field is a telephone number, stored as a double precision number in 8-byte MKD$ format.

The second field is a 24-byte string that stores the name that goes with the phone number. Suppose you want to store 1000 of these records in a random file. You want to be able to enter a phone number and have the system quickly find the name that goes with it.

Up until now, you've been accustomed to the idea of storing each new record in a file at the next available record position. The first record added goes in the first logical record, the second added goes in the second, and so forth. In hashing, a different idea is used. Before adding any records to the file, you decide on the capacity you'll allow for. As records are added, they are scattered into seemingly random locations throughout the file. The first record added might be stored as the 450th logical record of the file, the second might be stored in the 123rd, the third might be stored in the 842nd, and so forth.

How are these "random" locations selected? There are three steps:

A1. The key is converted to a number.

A2. That number is divided by the file capacity to find the remainder.

A3. The record containing the key and the other data that is associated with it is stored at the record number (plus 1) that equals the remainder gotten in step A2.

Suppose the first item to be added was:

```
Phone number:  (999) 3ØØ-3434
Name:          Jason Jones
```

The phone number is the key, and it is already a number, so for step A1, you get 9993003434. For step A2 you divide the number by the file capacity to find the remainder. You're allowing for 1000 records, so you divide 9993003434 by 1000. The remainder is 434. Now step A3 tells you to store the record at position 434 in the file, plus 1. Store the phone number and name for Jason Jones at record 435.

What have you accomplished with this? Now you have a way to quickly find whose name goes with (999) 300-3434. Let's say the operator sits down at the computer and types in that number. Here's how the search goes:

B1. The key is converted to a number (as it was in step A1 when you were adding).

B2. That number is divided by the file capacity to find the remainder.

B3. The remainder, plus 1, tells where the record is in the file. That record is gotten.

You can go on adding records with this method. Sooner or later you run into a problem. By chance, a new phone number that you wish to add, when divided by the file capacity, gives the same remainder as a previous phone number you added. The system attempts to store the new record and finds that the position is already occupied. Such a condition is called a "collision". Two different keys have "hashed" to the same location in the file. You have got to find a place to store the new record. To handle the collision, you need a method for computing an alternate location. Perhaps you might use the following steps.

C1. If a collision occurs, compute a new number based on the last number we used. That new number may be created in various ways. For a simple example, suppose you handle each collision by taking the last 9 digits of the prior number, adding 1, and multiplying the result by 7.

C2. Divide the new number you've created in step C1 by the file capacity to get the remainder. Add 1 to the remainder.

C3. Check the record location corresponding to the number gotten in step C2. If it is unoccupied, add the new record there. If it is already occupied, repeat step C1, until an unoccupied position is found where the record can be added.

Let's see how this might work. We're attempting to add the following item:

```
Phone Number:  (97Ø) 314-2434
Name:          David Daniels
```

Our program follows steps A1 through A3. Based on that, it attempts to add the new item at position 435, but it finds that phone number (999) 300-3434 is stored there. It goes to the collision handling routine. In step C1 it takes 703142434, adds 1, and multiplies by 7. The result is 4921997045. It divides by the capacity, 1000, takes the remainder and adds 1. The result is 45. The program checks record 45, finds that it's unoccupied, and adds David Daniels and his phone number there.

Suppose for a moment that a second collision occurred at record 45. Steps C1 through C2 would be repeated to find another position to attempt the add. The program would take the last 9 digits of 4921997045, add 1, and multiply by 7. The result would be 6453979322. Dividing by the capacity, taking the remainder and adding 1 would give record possition 323 as the next place to attempt the add. This process would be repeated until an empty record position is found.

Now you have got to modify the searching procedure to allow for the possibility of collisions:

D1. When searching, after a record number is computed, get the record and compare the field containing the key to the search key.

D2. If the field on disk matches the search key, the record has been found. If the record position is empty, the search has ended in a "not found" condition. But if the record position is occupied and it doesn't match, use the procedure in steps C1 and C2 to find the next position in the file to look.

D3. Repeat steps D1 and D2 until a match or a "not found" condition results.

This example has shown the basic ideas used to organize hash files, and to search and add in them. Several points are worth noting.

As the file approaches its maximum capacity, you tend to get more and more collisions. To minimize the effect of this, hash files are usually allocated to have more record positions than the anticipated requirement. In this example, if you need to store 1000 phone numbers and names, you might give the file a capacity of 1500 to reduce the number of collisions.

The procedure for computing a new hash number in the event of a collision is quite important. The goal is to keep the active records randomly disbursed throughout the file. The computation in step C1 is just an example to give you the idea. Computer science texts show better ways to re-generate hash numbers. You have to be concerned that the search doesn't degenerate into an endless loop caused by a repeating number pattern.

One possible way to be sure of avoiding an endless loop would be to eliminate the formula in C1, and just add 1 to the number each time step C1 is executed. But suppose that, by chance, a disproportionate number of your phone number keys end in, say "1234". You would have a "clustering" effect starting at position 235 in the hash file. The first key added that ends in "1234" would be added at 235, the second ending in "1234" would be added at 236, and so forth. If it happens that 30 keys have "1234" as their last 4 digits, the 30th one added will not be found until 29 collisions have occurred.

As you can see, in hashing it helps to have search keys that are random so that collisions and clustering are minimized. Since you happened to be dividing by 1000 in the example, it worked out that the first 7 digits weren't even considered in computing a location for the first record access. The randomness of the computation in steps A1 through A3 could have been improved if every digit was used. One way to accomplish this could have been to add the first 3 digits, the second 3 digits, the third 3 digits, and the final digit all together before doing the division.

What happens when the search key is not a number? Suppose you want to search based on the name field. You need a way to come up with a number from a key like "Jason Jones". One way to get a number from a string like this is to have your program add the ASCII values of all the letters. ASC("J") is 74, ASC("a") is 97, ASC("s") is 115, and so forth. In step A1 you can use the number that you get when you add up all the letters.

Another problem with an organization like we've discussed is that the file can be accessed by only one key. If you organize it for access by telephone number, you can't access it by name. If you organize it for access by name, you can't use the telephone number as a key.

Our discussion to this point has focused on the traditional approach to accessing disk files with hashing methods. Though hash file accessing is often done successfully with this approach, we've noted several problems:

- A large amount of unused space must be left in the file to minimize collisions.
- You can't read records back in the sequence in which they were added to the file.
- Access speeds can be hampered because the disk drive head has to jump around in the file to find the desired key. While this may be okay on a large computer with very fast disk drives, it can be slow on a microcomputer with relatively slow disk drives.
- Much depends on the randomness of the keys you use.
- One collision handling procedure produces the risk that an endless loop of repeating numbers will result from a key you might use. The other procedure, and possibly both, may result in clustering, which decreases the performance of the search and add operations.
- You are limited to only one key per record. You cannot easily access the file with alternate keys.

KEYACCES is based on hashing, but it uses various ways to get around all these problems. On the next few pages we will discuss the approach that KEYACCES uses, making it one of the most efficient, flexible, and speedy accessing methods you'll ever find on a microcomputer.

How KEYACCES Searches

Here is an overview of what happens when you wish to search for a key using the KEYACCES USR subroutine:

1. The operator enters a search key, or your BASIC program provides a key by other methods. This key could be a name, an alphanumeric item or account number, an integer compressed into MKI$ format or any other string that corresponds to a field in each logical record of the master file.

2. The KEYACCES routine is called from BASIC in "search" mode. It examines each byte of the the string that was provided as a search key, and it computes a 2-byte integer number based on the contents and position of each byte in the key string. Every character in the string enters into the computation, but blanks and binary zeros are ignored.

   This integer is the "hash code". The object of the computation has been to generate a number, such that, whenever you enter the same key, the same number is generated, but whenever you enter a different key, a drastically different number is likely to be computed. KEYACCES attempts to randomly scatter the hash codes generated across a range from 1 to 65535, and to minimize the chances that two different keys will generate the same hash code.

3. Once it's got a 2-byte integer hash code to represent the key, the next step is to translate that code into a disk location in a hash index file. Here's where KEYACCES departs from the usual approach to hash file searching. The hash index file acts as an intermediary between the search key and the master file that is being searched. The hash index file contains the logical record numbers for each active record in the master file. It is within this index that the scattering is done.

   To compute the position in the hash index file, KEYACCES first divides the hash code it generated by the number of physical records in the hash index. The remainder tells it which physical record to access in the hash index. Then a second division is done. The original hash code is divided by 85, and the remainder is taken to find a position within the hash index physical record that has been gotten. The number 85 is used because each hash index physical record stores up to 85 "pointers". Each pointer in the hash file index is 3 bytes long. Two of those bytes store a master file record number. We'll get to the purpose of the other byte in a minute.

4. Once KEYACCES has accessed a 3-byte pointer from the hash index file, it first checks to see if the most significant bit of the first byte in the pointer has been set. If it has been set, it knows that there was previously a pointer at the current position, but the record it once indexed in the master file has been deleted. In this case it gets the next 3-byte pointer, going back to the first record in the index if necessary, and it repeats step 4.

   Next it checks to see if the first two bytes of the pointer it has located are zero. If they are zeros, there is no record in the master file corresponding to the key you entered as the search key. KEYACCES reports a "not found" condition to the BASIC program.

   If the first two bytes of the pointer contain a non-zero number, it is the logical record number of one of the records in the master file. But KEYACCES, at this point, doesn't know for sure whether the record in the master file is an exact match for the search key, because a different search key could have "hashed" to the same location in the hash index. Here is where the third byte of the 3-byte pointer comes in. It is compared to the least significant byte of the 2-byte hash code generated in step 2. If the two bytes are not equal, step 4 is repeated with the next pointer in the hash index.

If the third byte of the 3-byte pointer and the least significant byte of the 2-byte hash code are equal, KEYACCES uses the pointer to specify a record in the master file being indexed. It does a disk file read, and the key in the master file logical record is compared to the search key. If they match, KEYACCES has found the logical record you are looking for, and it returns control back to your BASIC program, reporting a "found" condition. If they don't match, it repeats step 4, using the next pointer in the hash index.

It sounds like a rather complex procedure, but since everything is in machine language, it's very fast. Part of the complexity arises from the fact that from any two different search keys, KEYACCES can arrive at the same position in the hash index. To resolve the "collision", it ultimately needs to go to the master file to compare the search key with the key in the master file logical record. This access to the master file is fruitless if they don't match. So, to avoid, on average, 254 out of every 255 possible unsuccessful master file accesses, it first compares to the one-byte hash representation of the search key that is stored as the third byte of each pointer in the hash index.

After its first collision, or when it encounters a pointer marked as a previous deletion, KEYACCES resumes with the next pointer in the hash index. If it reaches the last position in the last physical record of the index, it continues with the first position in the first physical record. To avoid the possibility of an endless search through the hash index, at least one pointer position in the hash index must be empty. For best performance, at least 1 percent of the pointer positions in the index should be unused. With 1 percent of the pointers unused it will be very rare that KEYACCES will need to search more than 2 contiguous physical records in the hash index.

Figure 14.1 (see next page) shows how a successful KEYACCES search flows from your BASIC program, to the hash index file, and finally to the master file.

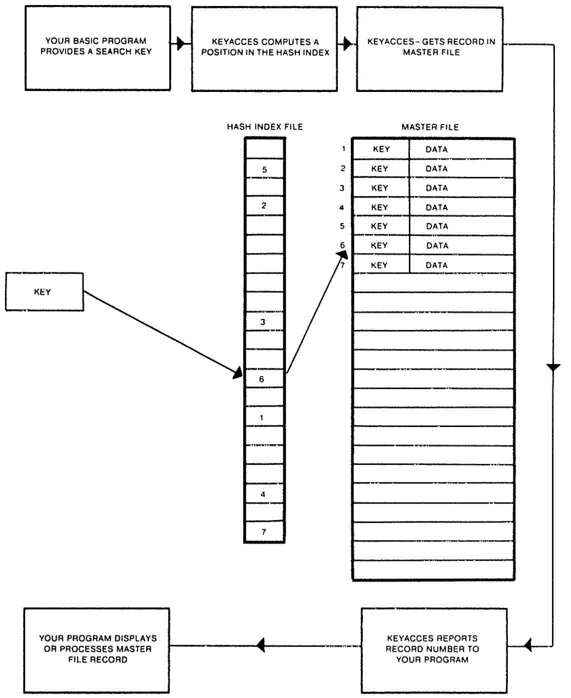
Resuming a Search

In the event you are allowing non-unique keys, KEYACCES lets you resume a search after the first matching key in the master file has been found. It does this by remembering where it left off in the hash index, and remembering the hash code that it computed. With a "resume-search" command, it starts at the next pointer in the hash index. It can repeatedly resume the search until a "not-found" condition is encountered. The effect of this is to give your BASIC program all the duplicates for any key in the master file.

Adding New Keys

The KEYACCES system provides two methods for adding new keys. If it has just completed a search with a "not-found" condition, or (in the case of files having non-unique keys) it has repeatedly resumed a search until a "not-found" condition was encountered, your BASIC program can use the "add" command, specifying the master file logical record number it wants recorded into the hash index. In most cases, KEYACCES simply writes the 2-byte pointer and 1-byte hash into the current position of the hash index. In the case that it encountered a deletion during its search to the current position, KEYACCES goes back to that position to add the new pointer. Recording the new record in the master file is up to your BASIC program.

Figure 14.1 — *A KEYACCES Search*


YOUR BASIC PROGRAM PROVIDES A SEARCH KEY
KEYACCES COMPUTES A POSITION IN THE HASH INDEX
KEYACCES - GETS RECORD IN MASTER FILE
HASH INDEX FILE
MASTER FILE
KEY
KEY DATA
YOUR PROGRAM DISPLAYS OR PROCESSES MASTER FILE RECORD
KEYACCES REPORTS RECORD NUMBER TO YOUR PROGRAM

When you want to insure that every key in the master file is unique, you do a search, and then you add only if KEYACCES returns a "not-found" condition.

The second method for adding new keys is the "search/add" method. This method combines the "search", "resume search" and "add" commands. The effect is, in one step, to compute a hash code, convert the code to a position in the hash index, search the hash index until a deletion or empty pointer is found, and record the new logical record pointer your BASIC program has provided. The "search/add" method is very convenient in programs where you are allowing non-unique keys.

Deleting Keys

KEYACCES can only do a delete from the hash index if it has just received a "search" or "resume search" command that has resulted in a succesful match. That insures that it knows which position in the hash index to mark as deleted. To mark a position in the hash index as deleted, KEYACCES simply changes all three bytes to zeros. Then it changes bit 7 of the second byte to a binary 1.

As you may have noted, hash index positions that have been marked as deleted are skipped during searches, but reused when you add. During a search, the KEYACCES routine remembers the location of the first deleted hash index pointer that is encountered. If an "add" follows the search, that location is used.

During deletions, KEYACCES takes care of its index for you. As for removing the record from the master file, that is up to your BASIC program.

How Hash Codes Are Generated

One of the most critical elements in the performance of hash file accessing is the way in which the program converts a search key into a number or "hash code". KEYACCES generates a number between 1 and 65535 for any key string that you give it. Though it is hardly scientific, much experimentation was done to come up with the method. Several problems had to be avoided:

- Keys having the same characters, such as "123" and "321" should generate different hash codes. It is not sufficient to simply add the ASCII values because only a limited range of codes can be generated that way.
- Two keys that are adjacent, such as "A101" and "A102", should not generate adjacent hash codes. Otherwise a clustering effect could result.
- Blanks and hexadecimal zeros should be eliminated from entering into the hash code computation so that BASIC doesn't have to make sure they are truncated before calling KEYACCES. It's also possible that a large number of blanks in each search key could bias the distribution and cause clustering.
- Significance should be given to the position of the characters in a key string. This compensates for the fact that zeros and blanks within the key string are not considered in the hash code computation.

To implement these ideas, KEYACCES does the following with a search key:

1. The hash code starts out as 0.
2. For each non-zero and non-blank character, the 8 bits are rotated. (This makes the values of common characters, such as "A" through "Z" and "0" through "9", non-adjacent.)
3. A count of the position in the string is maintained. The count is added to the rotated value of each character.
4. The 16 bits in the hash code are rotated 4 times to make sure the full range of numbers from 1 to 65535 is generated.
5. The 8 bits generated for the current character and the 8 bits for the previous character are added into the hash code. If the code exceeds 65535, it wraps around to a small number again.
6. If the final code is 0, it is changed to 1.

Once a 2-byte hash code that stores a number from 1 to 65535 is generated, it is used in three ways that help insure randomness. First, it is divided by the hash file index length, and the remainder is used to select the physical record number. Then the original hash code is divided by 85, and that remainder is used to select a position in the physical record. Finally, the remainder of the hash code divided by 256 is used as a one-byte hash for preliminary comparisons.

As a user of KEYACCES, you don't need to understand the foregoing discussions completely. A hazy awareness of what's going on is all that's required to write programs that implement it. The next sections of this chapter will show you the steps that are required.

How to Use KEYACCES

KEYACCES is a machine language USR subroutine, but you don't need to understand assembly language programming to use it. If you've never implemented USR calls in your programming, it would be a good idea to start with something short and simple, rather than KEYACCES. Appendix 2 in this book gives you a sample program that is easy to try out. *BASIC Faster and Better and Other Mysteries* gives many examples of USR subroutines with detailed explanations of the many ways you can work them into your BASIC programs.

The KEYACCES USR routine is fully relocatable, so you can POKE it or load it into any protected area of memory. The usual way is to reserve memory when you enter BASIC. KEYACCES is 629 bytes, so if you put it right at the top of memory, you'll need to specify a memory size of no more than 64907. If you use the magic array technique from *BASIC Faster and Better and Other Mysteries*, you don't need to protect any memory upon entering BASIC.

KALOADER/BAS

KALOADER/BAS is a BASIC program that you can use to POKE KEYACCES into memory. In a session where some of your programs will be using KEYACCES, you can run KALOADER/BAS first, and KEYACCES will be in memory when your other programs need it. KALOADER puts KEYACCES at memory locations **F000** through **F274**. (In decimal that's 61440 through 62068. In integer notation it's –4096 through–3864.) You can change line 20 to put it at any other location you wish, as long as you've reserved enough memory.

KALOADER/BAS first checks the memory size so it can stop you if you haven't reserved enough. Then it reads the 629 bytes to be POKEd from data statements. Finally, it asks you an important question:

```
DOES YOUR DISK OPERATING SYSTEM USE:

    A> A 5Ø-BYTE DCB, LIKE TRSDOS 1.3 FOR THE MODEL III
    B> A 32-BYTE DCB, LIKE TRSDOS 2.3 FOR THE MODEL I
```

DCB stands for data control block. Each file you allocate when entering BASIC has space in memory for a buffer, where your data is transferred to and from disk, and a DCB. The DCB is where the disk operating system keeps certain statistics about each file that is open. Depending on your operating system, the buffer is usually either 50 or 32 bytes beyond the DCB's starting address. Appendix 1 tells you how to find out what applies for the DOS you are using, and it tells you how to get some other numbers you'll be needing. If you are interested in getting more information on the DCB and what it contains, see your *Disk System Owner's Manual.*

KALOADER is requesting the information on your DCB size so that it can make a required modification to the 293rd byte of KEYACCES. That byte contains the offset from the beginning of the DCB to the beginning of the buffer. The modification is done by line 121 or 122 of KALOADER/BAS.

When KALOADER/BAS has finished, you are ready to use KEYACCESS. As a convenience, it notes for you the address that you will need in the DEFUSR statement of the program that you will be writing or using. As shown, assuming a starting address of **F000**, the DEFUSR parameter is –4096.

One other note: if you are typing in KALOADER, remember that the listing in this book is shown exactly as it will appear on your Model I or III screen. Watch out for some of the numbers in the data statements that are continued from one video line to the next.

Figure 14.2 — *KALOADER/BAS*

```
Ø 'KALOADER/BAS
1Ø CLS:PRINT"KEYACCES USR SUBROUTINE LOADER"
2Ø BA%=&HFØØØ:        'KEYACCES ADDRESS - MODIFY AS DESIRED
21 BA!=ASC(MKI$(BA%))+ASC(MID$(MKI$(BA%),2))*256
3Ø PRINT:PRINT"THIS PROGRAM POKES KEYACCESS TO MEMORY,":PRINT"ST
ARTING AT ADDRESS:";BA!
4Ø MS!=PEEK(16561)+PEEK(16562)*256+1
41 PRINT:PRINT"CURRENT MEMORY SIZE SETTING IS";MS!
42 IFMS!>BA!THENPRINT"YOU'VE NOT RESERVED ENOUGH MEMORY! PLEASE
RESTART BASIC.":END
5Ø PRINT:PRINT"PRESS <ENTER> TO BEGIN...":LINEINPUTA$
51 PRINT"POKING IN KEYACCES..."
6Ø A%=BA%
1ØØ FORX=1TO629                 'FOR EACH BYTE OF KEYACCESS
1Ø1 READ B%                     'READ A BYTE TO BE POKED
1Ø2 POKE A%, B%                 'POKE THE BYTE INTO THE ADDRESS
1Ø3 A%=A%+1                     'POINT TO NEXT ADDRESS
1Ø4 NEXT                        'REPEAT UNTIL DONE
1Ø5 PRINT
11Ø PRINT"DOES YOUR DISK OPERATING SYSTEM USE:"
111 PRINT"  A> A 5Ø-BYTE DCB, LIKE TRSDOS 1.3 FOR THE MODEL 3"
112 PRINT"  B> A 32-BYTE DCB, LIKE TRSDOS 2.3 FOR THE MODEL 1"
12Ø LINEINPUTA$:IFA$<>"A"ANDA$<>"B"THEN11Ø
121 IFA$="A"THENPOKEBA%+292,5Ø  'POKE BUFFER OFFSET FROM DCB
122 IFA$="B"THENPOKEBA%+292,32  'POKE BUFFER OFFSET FROM DCB
2ØØ PRINT:PRINT"YOU MAY NOW USE KEYACCES"
21Ø PRINT"YOUR DEFUSR ADDRESS IS  ";BA%
22Ø END
6Ø5ØØ 'KEYACCES  629 BYTES
6Ø5Ø1 DATA 2Ø5,127,1Ø,229,221,225,62,1,221,19Ø,Ø,4Ø,7,62,3,221,1
9Ø,Ø,32,68,221,11Ø,4,221,1Ø2,5,7Ø,35,94,35,86,213,253,225,17
6Ø5Ø2 DATA Ø,Ø,213,225,4,5,4Ø,41,62,32,183,14,1,245,253,94,Ø,187
,32,3,241,24,18,2Ø3,3,121,131,95,12,241,237,1Ø6,237,1Ø6,237
6Ø5Ø3 DATA 1Ø6,237,1Ø6,237,9Ø,83,253,35,16,224,125,18Ø,32,6,46,1
,24,2,24,77,221,117,22,229,221,86,2Ø,6,Ø,197,125,1Ø8,38,Ø,3Ø
6Ø5Ø4 DATA Ø,6,16,253,33,Ø,Ø,41,23,48,1,44,253,41,253,35,183,237
,82,48,3,25,253,43,16,237,193,2Ø3,64,32,69,2Ø3,72,32,1Ø,221
6Ø5Ø5 DATA 116;24,225,6,2,22,85,24,2Ø5,221,116,25,62,3,221,19Ø,Ø
,4Ø,7,175,221,119,26,221,119,27,62,2,221,19Ø,Ø,32,37,221,11Ø
6Ø5Ø6 DATA 24,221,1Ø2,25,36,62,85,188,32,3,44,38,Ø,221,126,2Ø,18
9,32,2,46,Ø,221,117,24,221,116,25,24,33,24,56,24,146,24,219
6Ø5Ø7 DATA 62,4,221,19Ø,Ø,32,2Ø,221,2Ø3,27,126,4Ø,14,221,126,26,
221,119,24,221,126,27,2Ø3,191,221,119,25,221,11Ø,18,221,1Ø2
6Ø5Ø8 DATA 19,221,78,24,6,Ø,221,94,16,221,86,17,213,253,225,197,
3Ø,Ø,24,6,24,82,24,198,24,194,213,9,229,193,2Ø3,67,32,1Ø,253
6Ø5Ø9 DATA 11Ø,1Ø,253,1Ø2,11,43,183,237,66,245,253,229,225,17,32
,Ø,25,241,4Ø,23,253,229,2Ø9,2Ø5,66,68,32,3,2Ø5,54,68,2Ø9,4Ø
6Ø51Ø DATA 1Ø,193,38,255,47,111,35,195,154,1Ø,2Ø9,2Ø3,67,32,25,1
93,221,94,25,22,Ø,213,235,41,25,2Ø9,22,Ø,25,229,24,1Ø,24,176
```

```
6Ø511 DATA 24,176,24,1Ø2,24,174,24,121,62,3,221,19Ø,Ø,56,61,94,3
5,86,35,7Ø,2Ø3,122,4Ø,3Ø,221,2Ø3,27,126,32,21,62,3,221,19Ø,Ø
6Ø512 DATA 4Ø,39,221,126,24,221,119,26,221,126,25,2Ø3,255,221,11
9,27,225,24,2ØØ,235,124,181,32,19,62,3,221,19Ø,Ø,4Ø,1Ø,221
6Ø513 DATA 117,2,221,116,3,195,154,1Ø,24,88,2Ø9,62,3,221,19Ø,Ø,4
Ø,222,12Ø,221,19Ø,22,32,216,221,117,2,221,116,3,221,86,1Ø,43
6Ø514 DATA 6,1,24,15Ø,229,253,229,225,221,94,6,221,86,7,213,253,
225,221,78,8,221,7Ø,9,2Ø3,195,24,131,227,221,116,3Ø,76,221,94
6Ø515 DATA 12,221,86,13,33,Ø,Ø,2Ø3,57,48,1,25,4Ø,5,235,41,235,24
,244,2Ø9,25,221,94,14,221,86,15,25,24,4,24,44,24,138,229,221
6Ø516 DATA 11Ø,4,221,1Ø2,5,7Ø,35,94,35,86,225,26,19Ø,32,237,35,1
9,16,248,253,11Ø,1Ø,253,1Ø2,11,221,117,28,221,116,29,221,11Ø
6Ø517 DATA 2,221,1Ø2,3,195,154,1Ø,225,62,4,221,19Ø,Ø,56,16,221,1
26,2,119,35,221,126,3,119,35,221,126,22,119,24,11,62,Ø,119,35
6Ø518 DATA 2Ø3,255,119,35,62,Ø,119,253,229,2Ø9,253,11Ø,1Ø,253,1Ø
2,11,43,253,117,1Ø,253,116,11,221,229,253,229,2Ø5,57,68,253
6Ø519 DATA 225,221,225,4Ø,183,38,255,47,111,35,195,154,1Ø
```

Note: Model II users must use modifications pages 370-373.

Allocating Space for the Hash File Index

One of the first steps in setting up the KEYACCES system is to decide how many keys you will allow for. This number will normally correspond to the number of logical records you are allowing for in the master file to be indexed. The hash file index requires 1 physical record for every 85 keys to be handled. Thus, if you want to allow for 1000 keys, the hash file index will be at least 12 physical records in length. If you want to allow for 5000 keys, the index must be at least 59 physical records in length. The computation to figure the minimum hash file index size is:

```
HASH INDEX SIZE IN PHYSICAL RECORDS = INT ( NUMBER OF KEYS / 85 ) + 1
```

For good performance, you should add 1 percent to the number you specify as the number of keys. For even better performance, you can add more than 1 percent. (Unlike most other hash accessing systems, you don't need to leave your file 10, 20, or 30 percent unused.) As a minimum for the above computation, you must add 1 to the number you specify as the number of keys.

The maximum size of any hash index is 256 physical records. This limitation is due to the division routines used in the machine language logic. Now you can see how we computed our limitation for the number of keys that can be handled. (256 * 85 - 1 = 21759 keys.)

The hash file index can be a separate file, and if you wish, on a separate disk drive from the master file to be indexed. You can reserve a block of contiguous physical records within your master file for the hash index. The hash file index may start at any physical record number within a file, so you can have more than one index in a file. Your only restriction is that the hash file index and the master file to be indexed must both be "on-line" during any call to the KEYACCES USR routine. That is, they must both be present in the disk drives and the files must be open.

Initializing the Hash File Index

To initialize a hash file index to an empty condition, you simply record zeros into each physical record to be used for the index. Thus, if your hash index is to start at physical record 1 of a file named "INDEX", and it is to be 12 physical records in length, your command is:

```
OPEN "R",1,"INDEX"          'OPEN THE FILE IN RANDOM MODE
FIELD 1,255 AS FD$          'FIELD THE FILE
LSET FD$ = STRING$(255,Ø)   'LOAD ZEROS TO THE FILE BUFFER
FOR X = 1 TO 12             'FOR EACH HASH INDEX PHYSICAL RECORD
PUT 1, X                    'RECORD FILE BUFFER TO DISKETTE
NEXT                        'REPEAT
CLOSE 1                     'CLOSE THE FILE
```

In most cases you won't need to have this routine (or a similar routine that performs the same function) in your main program, because you can handle the initialization job separately at original setup time.

The Control Array

Communication between your BASIC program and the KEYACCES USR routine is done with an integer control array. (You'll find more information about control arrays in *BASIC Faster and Better.*) The control array must have 16 elements for each hash index to be used in your program. Your USR argument when calling the KEYACCES routine will be the VARPTR of the first element in the control array. Here are the elements of the array and how they are used:

Element 0 is loaded by your BASIC program before any call to the KEYACCES USR routine to specify the command you want to execute. The command codes are:

```
1=SEARCH, 2=RESUME SEARCH, 3=SEARCH/ADD, 4=ADD, 5=DELETE
```

Element 1 is used for the master file logical record number. After a search, element 1 will contain the record number where the key was found in the master file. Before an add, your BASIC program must load element 1 with the logical record number of the record in the master file that you are adding. Logical record numbers may range from 1 to 32767.

Element 2 contains the VARPTR of the string you are using for your search key. Before a search, you load the string with the key you want to search for.

Element 3 is the memory address of the master file Data Control Block. This number will depend on what file buffer you are using for your master file, and the disk operating system you are using. Refer to Appendix 1 to see how you can find the proper number to load into element 3.

Element 4 is the is the master file offset. This tells the system the number of physical records that precede the first physical record in your master file. If logical record 1 of your master file starts in the first physical record, the offset will be zero. If logical record 1 of your master file starts in the 13th physical record, the offset will be 12. (Element 4 is most useful when you want to store your hash index at the beginning of your master file.)

Element 5 is the master file blocking factor. If, for example, you are blocking 4 logical records per physical record, element 5 would be specified as 4. If you are using the variable length record feature of your disk operating system, and your master file has been opened as a variable length record file, element 5 must be specified as 1.

Element 6 is the master file logical record length. If each logical record in the master file is 64 bytes, element 6 is specified as 64. (Element 5 multiplied by element 6 must be less than or equal to 256.)

Element 7 is the key offset within each master file logical record. This is how you specify where the key begins. Element 7 is the number of bytes that precede the key, so if, for example, your key starts in the first byte of each logical record, element 7 is specified as 0.

Element 8 is the memory address of the hash file index Data Control Block. This will depend on the file number that you are using for your hash file index, and the disk operating system you are using. Again, Appendix 1 tells you how to find the information you need. If the master file and the hash index are to share the same file, element 8 will be equal to element 3.

Element 9 is the hash index offset. If the hash index starts at physical record 1 of the file, your offset is 0. The purpose of element 9 is to let you store more than one hash index in a single file, or to use files that may contain other data in records preceding the hash index.

Element 10 is the hash index size in physical records. This is the number you computed earlier when deciding how much space to allocate for the hash file.

Element 11 is used by the KEYACCES USR routine to store the hash code that it has computed when doing a search. It should not be modified by your program.

Element 12 is used by the KEYACCES USR routine to store the current hash file index physical record number, and the subrecord number within that physical record. It should not be modified by your mainline program.

Element 13 is used by the KEYACCES USR routine to save the physical record and subrecord number of the first hash file index deletion that it encounters during a search, if any. If you want automatic recovery of deleted space in the hash index file, your program shouldn't modify element 13.

Element 14 returns the physical record number in the master file when a key is found after a search. It is provided so that your BASIC program won't need to compute the physical record number from the logical record number. If you are using the random disk file handler subroutines, you may wish to load PR%(PF%) and PP%(PF%) with this number after a search.

Element 15 returns the number of preceding subrecords within the master file physical record that contains your key after a successful search. It is provided to simplify the fielding computations for your BASIC program after a search. If you are using the random disk file handler subroutines, you may wish to load LS% with the contents of element 15 after a search. Then you can go directly to your logical record fielding subroutine, (58010, 58020, 58030, . . . depending on the file number).

The DEFUSR Statement

A DEFUSR statement tells BASIC the memory address where a machine language subroutine is to be executed. BASIC can remember up to 10 different USR subroutine addresses at a time. DEFUSR0 through DEFUSR9 define the addresses. To execute KEYACCES you can select any one of these. We will use USR8 in all the examples.

The command, which should be executed early in the program is:

```
DEFUSR8=&HFØØØ
```

or,

```
DEFUSR8=-4Ø96
```

Either of these will work if you loaded the 629-byte KEYACCESS routine at **FOOO**. If you executed KALOADER/BAS as shown, that's were KEYACCES will be.

Dimensioning the Control Array

You can load most of the values that will be required for the control array early in your program. It will be necessary for you to dimension the array with 16 elements. Thus, if you are going to be using just one hash index, you might use this dimension statement:

```
DIM KA%(15)                 'DIMENSION KA% AS CONTROL ARRAY
```

Now, with each call to KEYACCES, you'll use the VARPTR of the KA% as your USR argument. If you've done a DEFUSR so that USR8 calls KEYACCES, here is the statement you'll be using:

```
J% = USR8(VARPTR(KA%(Ø)))
```

If you are going to be using more than one hash index in a given program, you can double dimension your control array. The first dimension will be 15, and the second dimension will correspond to the number of hash indices to be used in the program. If, for example, you will have 3 hash indices in a program, the control array can be dimensioned with the following command:

```
DIM KA%(15,2)
```

Then, depending on the hash index you want to use, your call will be one of the following:

```
J% = USR8(VARPTR(KA%(Ø,Ø)))   'TO USE FIRST HASH INDEX
J% = USR8(VARPTR(KA%(Ø,1)))   'TO USE SECOND HASH INDEX
J% = USR8(VARPTR(KA%(Ø,2)))   'TO USE THIRD HASH INDEX
```

Remember that the use of KA% and USR8 is totally arbitrary. You may use any integer array for your control array, and any USR call, 0 through 9, to define the address of the KEYACCES routine. In our examples, J% receives the results of each call to KEYACCES. The use of J% is also arbitrary. It is important to know that J% (or whatever variable name you substitute for it) must have been used elsewhere in your program before the first call to KEYACCES to avoid a shift of the control array. This requirement can be satisfied by a statement such as "J%=0".

Loading the Control Array

Once you've dimensioned your control array, you can pre-load the values that will remain constant throughout your program. Normally, control array elements 2 through 10 will be constant. This can all be done in one program line, or you may wish to do it in several lines, with remarks to identify what you are loading.

The following example sets up the control array that will be used for most of the illustrations that follow. It assumes that you are using the TRSDOS 2.3 operating system, with file 1 as the master file, and file 2 as the hash file index:

```
DIM KA%(15)           'DIMENSION THE CONTROL ARRAY
KA$=""                'INITIALIZE A SEARCH KEY STRING
KA%(2)=VARPTR(KY$)    'PRE-LOAD SEARCH KEY VARPTR
KA%(3)=263Ø3          'MASTER FILE: FILE 1 DCB ADDRESS
A%(4)=Ø               'MASTER FILE BEGINS AT RECORD 1
KA%(5)=3              'BLOCKING: 3 LOGICAL RECS / PHYSICAL
KA%(6)=85             'MASTER FILE LOGICAL RECORD LENGTH
KA%(7)=Ø              'KEY POSITION: BYTE 1 IN EACH RECORD
KA%(8)=26593          'HASH INDEX: FILE 2 DCB ADDRESS
KA%(9)=Ø              'HASH INDEX BEGINS AT RECORD 1
KA%(1Ø)=12            'HASH INDEX CONTAINS 12 PHYSICAL RECS
```

Opening the Master File and Hash Index

Before you can use the KEYACCES you must open your master file and hash index file. For now, let's say you are using the random disk file handler in the program. The master file is to be a labeled file, "DATA:0", in buffer 1, and the index file is to be an unlabeled file, "INDX:0", in buffer 2. Here is the logic that can be used:

```
PF=1: FS$="DATA:Ø"          'LOAD MASTER FILE NUMBER AND NAME
GOSUB5825Ø :GOSUB584Ø5      'OPEN MASTER FILE, LOAD STATISTICS
PF=2: FS$="INDX:Ø"          'LOAD INDEX FILE NUMBER AND NAME
GOSUB5825Ø                  'OPEN INDEX FILE
```

If you've already become familiar with the labeled files handler, a question might come to mind. You set up KA%(4) to specify that the master file begins at record 1, but you use the first subrecord in physical record 1 to store the file statistics. Won't KEYACCES get confused and possibly overwrite the statistics? The answer is no. First of all, KEYACCES never writes to the master file. It just reads from it. Secondly, it will never read the first subrecord if you never tell it that there is a logical record 1 in the master file.

Adding a Record — Non-unique Keys

To add a record to a file in which you are allowing non-unique keys is simple. You provide the string to be used as the key, and the logical record number to be used as the pointer. Then you call KEYACCES in "search/add" mode. Here's an example that shows how your program logic might be organized if you are using the random disk file handler, and your master file is a labeled file.

```
1ØØ LINEINPUT KY$           'ALLOW ENTRY OF THE KEY
11Ø PF=1:GOSUB5842Ø         'GET & FIELD NEXT AVAILABLE RECORD

12Ø 'Logic to LSET data into the new record's fields goes here.

13Ø GOSUB5843Ø              'ADD THE NEW RECORD, UPDATE STATS
14Ø KA%(Ø)=3                'LOAD KEYACCES COMMAND - SEARCH/ADD
15Ø KA%(1)=LR(1)            'SPECIFY NUMBER USED FOR NEW RECORD
16Ø J%=USR8(VARPTR(KA%(Ø))) 'CALL THE KEYACCES USR ROUTINE
17Ø IF J% < Ø THEN GOTO ... 'HANDLE POSSIBLE DISK ERROR
18Ø PRINT "ADDED!"          'OTHERWISE, NEW REC & KEY ARE ADDED
```

The GOSUB 58420 in line 110 of the example above used the subroutine from the labeled file handler to get and field the next avaialable record in the master file. Upon return from 58420, LR%(1) contained the logical record number to be added. You can see that in line 150 we loaded it into element 1 of the control array.

Notice that, in line 130, you called subroutine 58430 to put the new record and update the file statistics before calling the KEYACCES routine. This is the usual sequence for doing it, but you could have called 58430 after calling KEYACCES. As long as the master file and the index are separate files there is no problem. On the other hand, when the master and the index share the same file buffer, you must write your master record before you do your KEYACCES call to avoid losing data that you've LSET into the master file record.

In line 170 you tested the contents of J% immediately after calling KEYACCES to handle the possibility that a disk error may have been encountered. If J% is less than zero upon return, it contains the minus value of the disk error code. The error codes returned correspond to those given in the TRSDOS section of your *Disk System Owner's Manual.* To illustrate, a –8 error code returned from KEYACCES is TRSDOS error code 8, "Device not available". When using any of the KEYACCES commands, no harm is done if you wish to do a retry after an error condition is encountered. In this situation, for example, the error handling logic could let the operator make a decision on whether to retry, repeating from line 160 if a second attempt is requested.

To summarize, once your master file logical record contains the key, the only essential steps of the "search/add" command are to:

1. Load the string you've specified as your key string.
2. Load element 0 of your control array with 3.
3. Load element 1 of your control array with the logical record number of the master file to be indexed.
4. Call KEYACCES.

Sometimes you will want to add pointers into your hash index for master file records that already exist. As you convert your old files to KEYACCES you may be doing "after the fact" indexing. You may be adding a pointer to a secondary hash index so that the file can be accessed on a different key field. Another situation in which you'll be doing a KEYACCES add, but not a master file add, is when you are making a change to a key field. In all of these cases, you don't need the "GOSUB 58420" and the "GOSUB 58430" that were used in the example.

Adding a Record — Unique Keys

To add a record to a file when you wish to prevent any two records from having the same accessing key, you must precede your "add" command with a "search" command. The object of your search is to insure that a "not found" condition is encountered. The steps required are to:

1. Load the string you've specified as your key string.
2. Load element 0 of your control array with 1 for "search".
3. Call KEYACCES.
4. Verify that 0 is returned, indicating "not found".
5. Add the new record to your master file.
6. Load element 0 of your control array with 4 to "add".
7. Load element 1 of with the logical record number of the record you are adding.
8. Call KEYACCES to record the new pointer in its index.

Here is an example that shows how you might wish to organize your program logic:

```
1ØØ LINEINPUT KY$            'ALLOW ENTRY OF THE KEY
11Ø KA%(Ø)=1                 'LOAD KEYACCES COMMAND - "SEARCH"
12Ø J%=USR8(VARPTR(KA%(Ø))) 'CALL KEYACCES ROUTINE
13Ø IF J%< Ø THEN GOTO ...  'HANDLE POSSIBLE ERROR CONDITION
131 IF J%> Ø THEN GOTO ...  'HANDLE IF KEY'S ALREADY IN FILE
14Ø PF=1 : GOSUB5842Ø       'GET AND FIELD RECORD TO BE ADDED

15Ø 'Logic to LSET data into the new record goes here...
```

```
160 GOSUB 58430             'ADD RECORD TO MASTER FILE
170 KA%(0)=4                'LOAD KEYACCES COMMAND - "ADD"
180 KA%(1)=LR(1)            'SPECIFY NUMBER USED FOR NEW RECORD
190 J%=USR8(VARPTR(KA%(0))) 'CALL THE KEYACCES USR ROUTINE
200 IF J% < 0 THEN GOTO ... 'HANDLE POSSIBLE DISK ERROR
210 PRINT "ADDED!"          'OTHERWISE, IT'S DONE
```

It is important to note that KEYACCES command 4, "add", must always follow a "search" or "resume search", and the search must have resulted in a "not found" condition.

Searching for a Record

KEYACCES command 1 lets you search for a record in the master file by key. Your program simply loads the key into the string you've specified, and loads element 0 of the control array with 1.

If a record is found whose key field matches the search key, KEYACCES returns the logical record number to your program. The buffer you are using for the master file contains the record. If you wish to field the record, element 15 of the control array will simplify things for your BASIC program by providing the number of preceding subrecords in the buffer. (Element 15 multiplied by your logical record length is the length that can be used for your "dummy" field.) Element 14 returns the physical record number in which the logical record resides.

If the key is not found, your USR call returns 0. If an error condition was encountered, the error code is returned as a negative number.

The following program lines illustrate the BASIC program logic that is required for a search:

```
300 LINEINPUT KY$           'ALLOW ENTRY OF THE SEARCH KEY
310 KA%(0)=1                'LOAD KEYACCES COMMAND - "SEARCH"
320 J%=USR8(VARPTR(KA%(0))) 'CALL KEYACCES USR ROUTINE
330 IF J%<0 THEN GOTO ....  'HANDLE POSSIBLE ERROR CONDITION
340 IF J%=0 THEN GOTO ....  'HANDLE "NOT FOUND" CONDITION
350 PRINT"FOUND, REC #";J%  'HANDLE "FOUND" CONDITION
```

Once you've found a record, you will usually want to field it so that you can LPRINT, display, or update its contents. The following lines could be used after line 350 of the previous illustration:

```
370 PF=1                    'SPECIFY FILE BUFFER NUMBER
371 LR(PF)=J%               'GET LOGICAL RECORD # FROM J%
372 PR(PF)=KA%(14)          'BRING PHYSICAL REC # UP TO DATE
373 PP(PF)=PR(PF)           'MAKE PREVIOUS PHYSICAL = CURRENT
374 LS=KA%(15)              'LOAD PRECEDING SUBRECORDS COUNT
380 GOSUB 58210             'GO FIELD FILE 1
390 'Logic to print or update the data goes here...
```

As you can see, each of the BASIC variables required by the random disk file handler was brought up to date in lines 370 through 374. In line 380 you called subroutine 58210 to do the fielding. Let's assume that the 85-byte record has 2 fields. The first, DK$, contains a 15-byte key. The second, DD$, contains 70 bytes of other data. Here's how subroutine 58010 might have been set up for fielding the master file:

```
58010 FIELD PF, LS * LL(PF) AS FD$(PF), 15 AS DK$, 70 AS DD$
58011 RETURN
```

Note that since the master file logical record is 85 bytes long, LL(1) and KA%(6) must have both been specified as 85. KA%(5) was specified as 3 because INT(256/85) = 3.

To field the master file logical record, you may, as an easier but slower alternative, use the following logic to replace lines 370 through 380:

```
37Ø PF=1 : LR(PF)=J% : GOSUB582ØØ
```

Resuming a Search

When you are allowing non-unique keys, you need the "resume search" command to find the remaining records in the master file, if any, that share the same key. In most cases, a "resume search" command will be used following a "search" command, but it can be used after any of the KEYACCES commands.

You specify a "resume search" by loading element 0 of your control array with 2. The information returned by a "resume search" command is exactly the same as that returned by a "search" command.

In most cases, to access with non-unique keys you will want to start with a "search" command, and then use the "resume search" command until a "not found" condition is encountered. The previous example could be extended to handle non-unique keys by adding the following line:

```
4ØØ KA%(Ø)=2 : GOTO 32Ø     'REQUEST "RESUME SEARCH": GO DO IT
```

The "resume search" command can be used in conjunction with the "add" command in situations where you wish to display all previous records that contain the same key before the operator makes the decision to add a new record to the master file.

You may also use the "resume search" when you want to insure that each new record with a non-unique key is added at the end of the "chain" of records with the same key. If you simply use the "search/add" command, the new record will be added at the position of the first deletion in the chain. If, on the other hand, you "search" and "resume search" until a "not found" condition is encountered, and then you load control array element 13 with 0 before the "add", you can defeat the feature that re-uses deleted pointers in the hash index. The effect will be to permit a re-display of the records that share a key, in the sequence in which they were added.

Deleting a Key

To delete a key, you must first access it with the "search" or "resume search" KEYACCES command. Then you load element 0 of your control array with 5, and you make your USR call. To make the previous illustration into a deletion routine, you could insert the following logic at line 390:

```
39Ø LINEINPUT "DELETE?";A$  'ALLOW OPERATOR TO DELETE
391 IF A$<>"Y" THEN 4ØØ     'SKIP IF NOT "Y" FOR YES
392 KA%(Ø)=5                'LOAD KEYACCES COMMAND - "DELETE"
393 J%=USR8(VARPTR(KA%(Ø))) 'DELETE POINTER FROM INDEX FILE
394 IF J%<Ø THEN ....       'HANDLE A POSSIBLE ERROR CONDITION
394 GOSUB 5844Ø             'NOW DELETE RECORD FROM MASTER FILE
```

Notice that you must handle the deletion in the hash file index before you delete the record from the master file. Otherwise, KEYACCES won't be able to find the appropriate position in its index to delete the pointer.

Changing a Key

To change a key, you must first access the master file record using the old key. Then you may use command 5 to delete the old key from the index. Next you must do a "search" and "add", or a "search/add" for the new key. Finally, you LSET the new key into the proper field of your master file logical record, and you PUT it.

If you are using labeled files, note that the procedure for changing a key does not involve subroutines 58020, 58030 or 58040. You are not deleting or re-adding the master file logical record. You are just deleting and re-adding the pointer to the master file record in the hash file index.

More About Error Conditions

With any call to the KEYACCES USR routine, there is the possibility that a disk error will be encountered. During program development, errors can be caused by such things as using incorrect DCB addresses, failing to open the required files, or failing to pre-create your hash index. An invalid DCB address will usually cause error 38, "I/O Attempt to Unopen File". Other incorrect parameters in your control array can lead to error 28, "Attempt to read past end of file".

Once your program is written and debugged, disk errors are more likely to occur because of a faulty diskette or disk drive. It is important to understand that BASIC's error routines are not activated by KEYACCES. In other words, you can't set up an "ON ERROR GOTO" and a "RESUME". Instead, your program must act upon the codes that are returned. This is best done by making your USR call into a subroutine, such as the one shown below:

```
58600 J%=USR8(VARPTR(KA$(0))) 'CALL THE KEYACCES USR ROUTINE
58601 IF J%>=0 THEN RETURN    'RETURN TO MAIN PROG. IF NO ERROR
58610 PRINT"DISK ERROR, CODE";ABS(J%)
58611 LINE INPUT "PRESS <ENTER> IF YOU WISH TO TRY AGAIN...";A$
58620 IF A$ <>"" THEN END ELSE 58600
```

The error codes that are returned by the KEYACCES routine are explained in Radio Shack's *TRSDOS & Disk BASIC* reference manual in the section on TRSDOS error messages. Most of the other disk operating systems available for the TRS-80 use the same codes.

KEYACCES and DOS Calls

It is up to your BASIC program to open and close the files that are involved in a KEYACCES system. The disk accesses required to maintain the index are performed by the DOS you are using. KEYACCES calls the required system routines directly. The following DOS calls are used:

```
CALL 04442H   Position
CALL 04436H   Read
CALL 04439H   Write
```

They are explained in the technical section of the TRSDOS *Disk System Owner's Manual*. I can't guarantee that this will always be the case, but every popular disk operating system I've seen for the TRS-80 Models I and III uses the same call addresses. It seems that, for compatibility, the various DOS vendors have found it worthwhile to recognize TRSDOS as the standard, at least in this respect. In any case, be sure to test your program thoroughly before you rely on it.

KEYACCES Flow

A study of the accompanying flow chart and assembly listing should answer any further technical questions you may have about how the various KEYACCES commands are executed.

KEYACCES was written to be fully relocatable without modification. To make a Z-80 machine language program relocatable it is necessary to make all jumps relative and to forgo the convenience of subroutine calls. Because of this, where long jumps are required to lead the flow from one part of the program to another, they are done as two or three short jumps. Where the same routines serve different purposes, they are treated like subroutines, but everything is done with relative jumps. If you have a particular reason to, you can reorganize the logic to make it non-relocatable. That will allow you to make it more neatly organized, and it will give you more flexibility if you want to expand or modify its capabilities.

Magic Array and Poke Formats

BASIC Faster and Better and Other Mysteries discusses the magic array technique. KEYACCES is listed in magic array and poke format so that you can use whatever technique you prefer to implement it.

Remember that you must make a change, depending on whether your version of Disk BASIC uses a 32-byte or a 50-byte DCB. If it's a 50-byte DCB, as with Model III Disk BASIC, change the 147th element from 32 to 50 in the magic array chart, and the 292nd byte value from 32 to 50 in the poke chart.

Magic Array Format, 315 elements

```
  32717  -69Ø2  -7715    318 -16675  1Ø24Ø  15879  -8957    19Ø
  1744Ø  28381  -8956   1382   9Ø3Ø   9Ø54 -1Ø922  -7683     17
 -11ØØ8   1249  1Ø245  15913 -18656    27Ø   -523     94   8379
  -3837   4632    971 -31879   3167  -4623  -4758  -4758  -4758
  -4758  21338   9213  -8176 -19331   1568    3Ø2    536  19736
  3Ø173  -689Ø  22237   1556 -151Ø4  27773     38     3Ø   41Ø2
   87Ø1      Ø   5929    3Ø4   -724   -727 -18653  21229    816
   -743   4139 -15891  16587  17696  18635   2592  29917  -7912
    518  21782 -13Ø32  29917  15897  -8957    19Ø   1832  -8785
   6775  3Ø685  15899  -8958    19Ø   95Ø4  28381  -8936   65Ø2
  159Ø8 -17323    8ØØ   9772  -896Ø   5246   8381  11778  -896Ø
   6261  29917   6169   6177   62ØØ   629Ø  16Ø91  -8956    19Ø
   5152 -13347  32283   3624  32477  -8934   6263  32477 -13541
  -8769   6519  28381  -8942   4966  2Ø189   156Ø  -896Ø   419Ø
  22237 -1Ø991  -7683   7877   6144   615Ø   6226   6342 -1Ø814
  -69Ø3 -13375   8259   -758   267Ø  26365  11Ø19  -4681  -275Ø
  -6659   4577     32  -3815   5928  -6659 -12847  17474    8ØØ
  14Ø29 -11964   26ØØ   9921  12287   9Ø71 -25917 -12Ø22  17355
   6432  -8767   6494     22  -5163   6441   5841   64ØØ   6373
   6154   632Ø   632Ø   6246   6318  15993  -8957    19Ø  15672
   9Ø54   9Ø46 -13498  1Ø362  -893Ø   7115   8318  15893  -8957
    19Ø  1ØØ24  32477  -8936   6775  32477 -13543  -87Ø5   7Ø31
   6369  -5176 -19Ø76   4896    83Ø -16675  1Ø24Ø -12Ø22  3Ø173
  -8958    884 -25917   6154 -11944    83Ø -16675  1Ø24Ø  3Ø942
 -16675   8214  -8744    629  29917  -8957   2646   1579   6145
  -6762  -6659  -8735   163Ø  22237 -11ØØ1  -7683  2Ø189  -8952
  2374 -15413 -31976  -8733   7796  -8884   3166  22237   8461
     Ø  14795    3Ø4  1Ø265  -5371  -5335  -3048   66Ø9  24285
 -8946   3926   6169   6148   6188  -6774  28381  -8956   1382
  9Ø3Ø   9Ø54  -785Ø -1687Ø  -4832   4899  -2Ø32  28413   -758
  2918  3Ø173  -8932   754Ø  28381  -8958    87Ø -25917  -7926
  1Ø86 -16675  14336  -8944    638   9Ø79  32477  3Ø467  -8925
  5758   6263  15883  3Ø464 -13533  3Ø719  159Ø7  3Ø464  -6659
  -559   267Ø  26365  11Ø19  3Ø2Ø5   -758   2932  -6691  -6659
 14797   -7ØØ  -8735  1Ø465   9911  12287   9Ø71 -25917     1Ø
```

Poke Format, 629 bytes

```
2Ø5 127  1Ø 229 221 225  62   1 221 19Ø   Ø  4Ø   7  62   3 221
19Ø   Ø  32  68 221 11Ø   4 221 1Ø2   5  7Ø  35  94  35  86 213
253 225  17   Ø   Ø 213 225   4   5  4Ø  41  62  32 183  14   1
245 253  94   Ø 187  32   3 241  24  18 2Ø3   3 121 131  95  12
241 237 1Ø6 237 1Ø6 237 1Ø6 237 1Ø6 237  9Ø  83 253  35  16 224
125 18Ø  32   6  46   1  24   2  24  77 221 117  22 229 221  86
 2Ø   6   Ø 197 125 1Ø8  38   Ø  3Ø   Ø   6  16 253  33   Ø   Ø
 41  23  48   1  44 253  41 253  35 183 237  82  48   3  25 253
 43  16 237 193 2Ø3  64  32  69 2Ø3  72  32  1Ø 221 116  24 225
  6   2  22  85  24 2Ø5 221 116  25  62   3 221 19Ø   Ø  4Ø   7
175 221 119  26 221 119  27  62   2 221 19Ø   Ø  32  37 221 11Ø
 24 221 1Ø2  25  36  62  85 188  32   3  44  38   Ø 221 126  2Ø
189  32   2  46   Ø 221 117  24 221 116  25  24  33  24  56  24
146  24 219  62   4 221 19Ø   Ø  32  2Ø 221 2Ø3  27 126  4Ø  14
221 126  26 221 119  24 221 126  27 2Ø3 191 221 119  25 221 11Ø
 18 221 1Ø2  19 221  78  24   6   Ø 221  94  16 221  86  17 213
253 225 197  3Ø   Ø  24   6  24  82  24 198  24 194 213   9 229
193 2Ø3  67  32  1Ø 253 11Ø  1Ø 253 1Ø2  11  43 183 237  66 245
253 229 225  17  32   Ø  25 241  4Ø  23 253 229 2Ø9 2Ø5  66  68
 32   3 2Ø5  54  68 2Ø9  4Ø  1Ø 193  38 255  47 111  35 195 154
 1Ø 2Ø9 2Ø3  67  32  25 193 221  94  25  22   Ø 213 235  41  25
2Ø9  22   Ø  25 229  24  1Ø  24 176  24 176  24 1Ø2  24 174  24
121  62   3 221 19Ø   Ø  56  61  94  35  86  35  7Ø 2Ø3 122  4Ø
 3Ø 221 2Ø3  27 126  32  21  62   3 221 19Ø   Ø  4Ø  39 221 126
 24 221 119  26 221 126  25 2Ø3 255 221 119  27 225  24 2ØØ 235
124 181  32  19  62   3 221 19Ø   Ø  4Ø  1Ø 2Ø9 221 117   2 221
116   3 195 154  1Ø  24  88 2Ø9  62   3 221 19Ø   Ø  4Ø 222 12Ø
221 19Ø  22  32 216 221 117   2 221 116   3 221  86  1Ø  43   6
  1  24 15Ø 229 253 229 225 221  94   6 221  86   7 213 253 225
221  78   8 221  7Ø   9 2Ø3 195  24 131 227 221 116  3Ø  76 221
 94  12 221  86  13  33   Ø   Ø 2Ø3  57  48   1  25  4Ø   5 235
 41 235  24 244 2Ø9  25 221  94  14 221  86  15  25  24   4  24
 44  24 138 229 221 11Ø   4 221 1Ø2   5  7Ø  35  94  35  86 225
 26 19Ø  32 237  35  19  16 248 253 11Ø  1Ø 253 1Ø2  11 221 117
 28 221 116  29 221 11Ø   2 221 1Ø2   3 195 154  1Ø 225  62   4
221 19Ø   Ø  56  16 221 126   2 119  35 221 126   3 119  35 221
126  22 119  24  11  62   Ø 119  35 2Ø3 255 119  35  62   Ø 119
253 229 2Ø9 253 11Ø  1Ø 253 1Ø2  11  43 253 117  1Ø 253 116  11
221 229 253 229 2Ø5  57  68 253 225 221 225  4Ø 183  38 255  47
111  35 195 154  1Ø
```

Figure 14.3 — *KEYACCES Assembly Listing*

```
                ØØØØØ ;KEYACCES
                ØØØØ1 ;
FØØØ            ØØ3ØØ         ORG     ØFØØØH         ;ORIGIN - (RELOCATABLE)
FØØØ CD7FØA     ØØ31Ø         CALL    ØA7FH          ;PUT USR ARGUMENT IN HL
FØØ3 E5         ØØ32Ø         PUSH    HL             ;
FØØ4 DDE1       ØØ33Ø         POP     IX             ;IX POINTS TO CONTROL ARRAY
                ØØ34Ø ;THE FOLLOWING LOGIC TESTS ON COMMAND FROM BASIC
FØØ6 3EØ1       ØØ35Ø         LD      A,Ø1H          ;TEST ON SEARCH COMMAND
FØØ8 DDBEØØ     ØØ36Ø         CP      (IX+Ø)         ;COMPARE WITH COMMAND FROM BASIC
FØØB 28Ø7       ØØ37Ø         JR      Z,J2           ;COMPUTE HASH IF COMMAND 1
FØØD 3EØ3       ØØ38Ø         LD      A,Ø3H          ;TEST ON SEARCH/ADD
FØØF DDBEØØ     ØØ39Ø         CP      (IX+Ø)         ;COMPARE WITH COMMAND FROM BASIC
FØ12 2Ø44       ØØ4ØØ         JR      NZ,J1AH        ;JUMP IF NOT SEARCH/ADD
                ØØ41Ø ;
                ØØ42Ø ;THE FOLLOWING LOGIC COMPUTES THE ORIGINAL HASH CODE
FØ14 DD6EØ4     ØØ43Ø J2      LD      L,(IX+4)       ;
FØ17 DD66Ø5     ØØ44Ø         LD      H,(IX+5)       ;HL=SKEY VARPTR
FØ1A 46         ØØ45Ø         LD      B,(HL)         ;B=SKEY LENGTH
FØ1B 23         ØØ46Ø         INC     HL             ;HL POINTS TO SKEY POINTERS
FØ1C 5E         ØØ47Ø         LD      E,(HL)         ;
FØ1D 23         ØØ48Ø         INC     HL             ;
FØ1E 56         ØØ49Ø         LD      D,(HL)         ;DE POINTS TO SKEY
FØ1F D5         ØØ5ØØ         PUSH    DE             ;
FØ2Ø FDE1       ØØ51Ø         POP     IY             ;IY POINTS TO SKEY
FØ22 11ØØØØ     ØØ52Ø         LD      DE,Ø           ;DE = Ø FOR HASH COMP
FØ25 D5         ØØ53Ø         PUSH    DE             ;
FØ26 E1         ØØ54Ø         POP     HL             ;HL = Ø FOR HASH COMP
FØ27 Ø4         ØØ55Ø         INC     B              ;
FØ28 Ø5         ØØ56Ø         DEC     B              ;Z FLAG IS SET IF B=Ø
FØ29 2829       ØØ57Ø         JR      Z,J2D          ;JUMP IF ZERO LENGTH
FØ2B 3E2Ø       ØØ58Ø         LD      A,Ø2ØH         ;PUT SPACE CODE IN ACCUM
FØ2D B7         ØØ59Ø         OR      A              ;CLEAR CARRY FLAG
FØ2E ØEØ1       ØØ6ØØ         LD      C,Ø1H          ;
FØ3Ø F5         ØØ61Ø J2A     PUSH    AF             ;SAVE FLAGS
FØ31 FD5EØØ     ØØ62Ø         LD      E,(IY)         ;PREP TO ADD NEXT BYTE
FØ34 BB         ØØ63Ø         CP      E              ;COMPARE TO SPACE CODE
FØ35 2ØØ3       ØØ64Ø         JR      NZ,J2B         ;JUMP IF NOT A SPACE
FØ37 F1         ØØ65Ø         POP     AF             ;RESTORE FLAGS
FØ38 1812       ØØ66Ø         JR      J2C            ;IGNORE SPACE
FØ3A CBØ3       ØØ67Ø J2B     RLC     E              ;ROTATE THE CHARACTER
FØ3C 79         ØØ68Ø         LD      A,C            ;LOAD COUNT TO ACCUM
FØ3D 83         ØØ69Ø         ADD     A,E            ;ADD ROTATED CHARACTER TO COUNT
FØ3E 5F         ØØ7ØØ         LD      E,A            ;PUT IT BACK IN E REGISTER
FØ3F ØC         ØØ71Ø         INC     C              ;ADD TO COUNT
FØ4Ø F1         ØØ72Ø         POP     AF             ;RESTORE FLAGS
FØ41 ED6A       ØØ73Ø         ADC     HL,HL          ;ROTATE HASH ONCE
FØ43 ED6A       ØØ74Ø         ADC     HL,HL          ;ROTATE HASH AGAIN
FØ45 ED6A       ØØ75Ø         ADC     HL,HL          ;ROTATE HASH AGAIN
```

```
FØ47 ED6A      ØØ76Ø        ADC     HL,HL          ;HL IS NOW ROTATED LEFT 4
FØ49 ED5A      ØØ77Ø        ADC     HL,DE          ;ADD PREV AND CURR BYTE
FØ4B 53        ØØ78Ø        LD      D,E            ;ROLL CURRENT BYTE TO PREV
FØ4C FD23      ØØ79Ø J2C    INC     IY             ;POINT TO NEXT BYTE IN SKEY
FØ4E 1ØEØ      ØØ8ØØ        DJNZ    J2A            ;REPEAT IF NOT ALL PROCESSED
FØ5Ø 7D        ØØ81Ø        LD      A,L            ;TEST FOR ZERO HASH
FØ51 B4        ØØ82Ø        OR      H              ;
FØ52 2ØØ6      ØØ83Ø        JR      NZ,J3A         ;HASH IS OK IF NON-ZERO
FØ54 2EØ1      ØØ84Ø J2D    LD      L,Ø1H          ;OTHERWISE MAKE IT A 1
FØ56 18Ø2      ØØ85Ø        JR      J3A            ;
               ØØ86Ø ;
               ØØ87Ø ;THE FOLLOWING LOGIC HANDLES A HALF-JUMP FOR RELOCATABILITY
FØ58 184D      ØØ88Ø J1AH   JR      J1A            ;
               ØØ89Ø ;
               ØØ9ØØ ;THE FOLLOWING LOGIC CONVERTS CURRENT HASH CODE TO A
               ØØ91Ø ;PHYSICAL RECORD AND SUBRECORD WITHIN HASH INDEX
FØ5A DD7516    ØØ92Ø J3A    LD      (IX+22),L      ;FIRST, SAVE 1-BYTE HASH
FØ5D E5        ØØ93Ø        PUSH    HL             ;SAVE HASH CODE
FØ5E DD5614    ØØ94Ø        LD      D,(IX+2Ø)      ;DIVISOR IS NUMBER OF PHYS RECS
FØ61 Ø6ØØ      ØØ95Ø        LD      B,ØØH          ;NO BRANCH AFTER DIVIDE
FØ63 C5        ØØ96Ø J3B    PUSH    BC             ;SAVE BRANCH INDICATOR
FØ64 7D        ØØ97Ø        LD      A,L            ;DIVIDE HL BY D
FØ65 6C        ØØ98Ø        LD      L,H            ;CONTINUE DIVISION
FØ66 26ØØ      ØØ99Ø        LD      H,ØH           ;CONTINUE DIVISION
FØ68 1EØØ      Ø1ØØØ        LD      E,ØH           ;CONTINUE DIVISION
FØ6A Ø61Ø      Ø1Ø1Ø        LD      B,1ØH          ;CONTINUE DIVISION
FØ6C FD21ØØØØ  Ø1Ø2Ø        LD      IY,ØH          ;CONTINUE DIVISION
FØ7Ø 29        Ø1Ø3Ø J3C    ADD     HL,HL          ;CONTINUE DIVISION
FØ71 17        Ø1Ø4Ø        RLA                    ;CONTINUE DIVISION
FØ72 3ØØ1      Ø1Ø5Ø        JR      NC,J3D         ;CONTINUE DIVISION
FØ74 2C        Ø1Ø6Ø        INC     L              ;CONTINUE DIVISION
FØ75 FD29      Ø1Ø7Ø J3D    ADD     IY,IY          ;CONTINUE DIVISION
FØ77 FD23      Ø1Ø8Ø        INC     IY             ;CONTINUE DIVISION
FØ79 B7        Ø1Ø9Ø        OR      A              ;CONTINUE DIVISION
FØ7A ED52      Ø11ØØ        SBC     HL,DE          ;CONTINUE DIVISION
FØ7C 3ØØ3      Ø111Ø        JR      NC,J3E         ;CONTINUE DIVISION
FØ7E 19        Ø112Ø        ADD     HL,DE          ;CONTINUE DIVISION
FØ7F FD2B      Ø113Ø        DEC     IY             ;CONTINUE DIVISION
FØ81 1ØED      Ø114Ø J3E    DJNZ    J3C            ;CONTINUE DIVISION
FØ83 C1        Ø115Ø        POP     BC             ;RESTORE RETURN INDICATOR
FØ84 CB4Ø      Ø116Ø        BIT     Ø,B            ;TEST FOR RETURN TO J6D
FØ86 2Ø45      Ø117Ø        JR      NZ,J6DH        ;JUMP IF IT IS
FØ88 CB48      Ø118Ø        BIT     1,B            ;TEST FOR RETURN TO J3F
FØ8A 2ØØA      Ø119Ø        JR      NZ,J3F         ;JUMP IF IT IS
FØ8C DD7418    Ø12ØØ        LD      (IX+24),H      ;IX+24 HAS HASH PHYS REC
FØ8F E1        Ø121Ø        POP     HL             ;HL HAS HASH CODE
FØ9Ø Ø6Ø2      Ø122Ø        LD      B,Ø2H          ;LOAD RETURN INDICATOR
FØ92 1655      Ø123Ø        LD      D,85           ;LOAD HASH BLOCKING FACTOR
FØ94 18CD      Ø124Ø        JR      J3B            ;GO DIVIDE
FØ96 DD7419    Ø125Ø J3F    LD      (IX+25),H      ;IX+25 HAS HASH SUB RECORD
FØ99 3EØ3      Ø126Ø        LD      A,Ø3H          ;LOAD SEARCH/ADD COMMAN
FØ9B DDBEØØ    Ø127Ø        CP      (IX+Ø)         ;COMPARE TO COMMAND FROM BASIC
```

```
FØ9E 28Ø7      Ø128Ø        JR      Z,J1A           ;JUMP IF IT IS
FØAØ AF        Ø129Ø        XOR     A               ;CLEAR ACCUM
FØA1 DD771A    Ø13ØØ        LD      (IX+26),A       ;
FØA4 DD771B    Ø131Ø        LD      (IX+27),A       ;"FIRST-DELETED" POINTER CLEARED
               Ø132Ø ;
               Ø133Ø ;THE FOLLOWING LOGIC TESTS FOR RESUME-SEARCH COMMAND FROM BASIC
FØA7 3EØ2      Ø134Ø J1A    LD      A,Ø2H           ;RESUME-SEARCH COMMAND
FØA9 DDBEØØ    Ø135Ø        CP      (IX+Ø)          ;COMPARE WITH COMMAND FROM BASIC
FØAC 2Ø25      Ø136Ø        JR      NZ,J1B          ;JUMP IF NOT RESUME-SEARCH
               Ø137Ø ;
               Ø138Ø ;THE FOLLOWING LOGIC INCREMENTS TO NEXT HASH INDEX RECORD
FØAE DD6E18    Ø139Ø J4A    LD      L,(IX+24)       ;L = OLD PHYSICAL RECORD
FØB1 DD6619    Ø14ØØ        LD      H,(IX+25)       ;H = OLD SUB RECORD
FØB4 24        Ø141Ø        INC     H               ;GO TO NEXT SUB RECORD
FØB5 3E55      Ø142Ø        LD      A,85            ;LOAD HASH BLOCKING FACTOR
FØB7 BC        Ø143Ø        CP      H               ;COMPARE IT
FØB8 2ØØ3      Ø144Ø        JR      NZ,J4B          ;JUMP IF IT'S NOT AT LIMIT
FØBA 2C        Ø145Ø        INC     L               ;ADD 1 TO PHYSICAL REC
FØBB 26ØØ      Ø146Ø        LD      H,ØØH           ;RESTORE SUB RECORD TO ZERO
FØBD DD7E14    Ø147Ø J4B    LD      A,(IX+2Ø)       ;ACCUM HAS PHYS REC LIMIT
FØCØ BD        Ø148Ø        CP      L               ;COMPARE PHYS REC TO LIMIT
FØC1 2ØØ2      Ø149Ø        JR      NZ,J4C          ;JUMP IF NOT AT LIMIT
FØC3 2EØØ      Ø15ØØ        LD      L,ØØH           ;OTHERWISE RESTORE TO ZERO
FØC5 DD7518    Ø151Ø J4C    LD      (IX+24),L       ;STORE NEW PHYS RECORD
FØC8 DD7419    Ø152Ø        LD      (IX+25),H       ;STORE NEW SUB RECORD
FØCB 1821      Ø153Ø        JR      J1C             ;
               Ø154Ø ;
               Ø155Ø ;THE FOLLOWING LOGIC HANDLES SOME HALF-JUMPS FOR RELOCATABILITY
FØCD 1838      Ø156Ø J6DH   JR      J6DH1           ;
FØCF 1892      Ø157Ø J3BH2  JR      J3B             ;
FØD1 18DB      Ø158Ø J4AH2  JR      J4A             ;
               Ø159Ø ;
               Ø16ØØ ;THE FOLLOWING LOGIC PREPARES TO GET HASH INDEX RECORD FROM DISK
FØD3 3EØ4      Ø161Ø J1B    LD      A,Ø4H           ;LOAD "ADD" COMMAND CODE
FØD5 DDBEØØ    Ø162Ø        CP      (IX+Ø)          ;COMPARE WITH COMMAND FROM BASIC
FØD8 2Ø14      Ø163Ø        JR      NZ,J1C          ;JUMP IF NOT AN "ADD" COMMAND
FØDA DDCB1B7E  Ø164Ø        BIT     7,(IX+27)       ;WAS THERE A DELETION IN CHAIN?
FØDE 28ØE      Ø165Ø        JR      Z,J1C           ;JUMP IF THERE WASN'T
FØEØ DD7E1A    Ø166Ø        LD      A,(IX+26)       ;A=PHYS REC# OF DELETION
FØE3 DD7718    Ø167Ø        LD      (IX+24),A       ;REPLACE DESIRED PHYS REC
FØE6 DD7E1B    Ø168Ø        LD      A,(IX+27)       ;A=SUB REC# OF DELETION
FØE9 CBBF      Ø169Ø        RES     7,A             ;RESET INDICATOR BIT
FØEB DD7719    Ø17ØØ        LD      (IX+25),A       ;REPLACE DESIRED SUB RECORD
FØEE DD6E12    Ø171Ø J1C    LD      L,(IX+18)       ;
FØF1 DD6613    Ø172Ø        LD      H,(IX+19)       ;HL HAS HASH INDEX OFFSET
FØF4 DD4E18    Ø173Ø        LD      C,(IX+24)       ;
FØF7 Ø6ØØ      Ø174Ø        LD      B,ØØH           ;BC HAS PHYS RECORD DESIRED
FØF9 DD5E1Ø    Ø175Ø        LD      E,(IX+16)       ;
FØFC DD5611    Ø176Ø        LD      D,(IX+17)       ;DE POINTS TO HASH INDEX DCB
FØFF D5        Ø177Ø        PUSH    DE              ;
F1ØØ FDE1      Ø178Ø        POP     IY              ;IY POINTS TO HASH INDEX DCB
F1Ø2 C5        Ø179Ø        PUSH    BC              ;PUT DESIRED RECORD ON STACK
```

```
F1Ø3 1EØØ     Ø18ØØ        LD     E,ØØH          ;SET RETURN INDICATOR
F1Ø5 18Ø6     Ø181Ø        JR     J5A            ;
              Ø182Ø ;
              Ø183Ø ;THE FOLLOWING LOGIC HANDLES SOME HALF-JUMPS FOR RELOCATABILITY
F1Ø7 1852     Ø184Ø J6DH1  JR     J6DH2          ;
F1Ø9 18C6     Ø185Ø J4AH1  JR     J4AH2          ;
F1ØB 18C2     Ø186Ø J3BH1  JR     J3BH2          ;
              Ø187Ø ;
              Ø188Ø ;THE FOLLOWING LOGIC GETS A RECORD FROM DISK AND POINTS HL TO
                    ;BUFFER
F1ØD D5       Ø189Ø J5A    PUSH   DE             ;SAVE RETURN INDICATOR
F1ØE Ø9       Ø19ØØ        ADD    HL,BC          ;ADD OFFSET TO DESIRED REC
F1ØF E5       Ø191Ø        PUSH   HL             ;
F11Ø C1       Ø192Ø        POP    BC             ;BC HAS ACTUAL PHYS REC
F111 CB43     Ø193Ø        BIT    Ø,E            ;ARE WE GETTING MASTER REC?
F113 2ØØA     Ø194Ø        JR     NZ,J5F         ;FORCE "HARD-GET" IF WE ARE
F115 FD6EØA   Ø195Ø        LD     L,(IY+1Ø)      ;
F118 FD66ØB   Ø196Ø        LD     H,(IY+11)      ;HL HAS NRN FOR DCB
F11B 2B       Ø197Ø        DEC    HL             ;HL = CURRENT REC # IN BUFFER
F11C B7       Ø198Ø        OR     A              ;CLEAR CARRY
F11D ED42     Ø199Ø        SBC    HL,BC          ;COMPARE DESIRED WITH CURRENT
F11F F5       Ø2ØØØ J5F    PUSH   AF             ;SAVE FLAGS
F12Ø FDE5     Ø2Ø1Ø        PUSH   IY             ;
F122 E1       Ø2Ø2Ø        POP    HL             ;HL POINTS TO DCB
F123 112ØØØ   Ø2Ø3Ø        LD     DE,2ØH         ;ADD 32 TO DCB
              Ø2Ø31 ;NOTE: CHANGE ABOVE TO "LD DE,32H" IF DCB IS 5Ø BYTES
              Ø2Ø32 ;      AS IT IS FOR TRSDOS MODEL 3 DISK BASIC
F126 19       Ø2Ø4Ø        ADD    HL,DE          ;HL POINTS TO BUFFER
F127 F1       Ø2Ø5Ø        POP    AF             ;RESTORE FLAGS
F128 2817     Ø2Ø6Ø        JR     Z,J5D          ;NO GET REQUIRED IF EQUAL
F12A FDE5     Ø2Ø7Ø        PUSH   IY             ;
F12C D1       Ø2Ø8Ø        POP    DE             ;DE POINTS TO FILE DCB
F12D CD4244   Ø2Ø9Ø        CALL   Ø4442H         ;POSITION
F13Ø 2ØØ3     Ø21ØØ        JR     NZ,J5C         ;JUMP IF DISK ERROR
F132 CD3644   Ø211Ø        CALL   Ø4436H         ;READ
F135 D1       Ø212Ø J5C    POP    DE             ;RESTORE RETURN INDICATOR
F136 28ØA     Ø213Ø        JR     Z,J5B          ;SKIP FOLLOWING IF NO ERROR
F138 C1       Ø214Ø        POP    BC             ;RESTORE STACK
F139 26FF     Ø215Ø        LD     H,ØFFH         ;
F13B 2F       Ø216Ø        CPL                   ;
F13C 6F       Ø217Ø        LD     L,A            ;
F13D 23       Ø218Ø        INC    HL             ;HL HAS MINUS OF ERROR CODE
F13E C39AØA   Ø219Ø        JP     ØA9AH          ;RETURN ERROR CODE TO BASIC
F141 D1       Ø22ØØ J5D    POP    DE             ;RESTORE RETURN INDICATOR
F142 CB43     Ø221Ø J5B    BIT    Ø,E            ;TEST FOR RETURN TO J6E
F144 2Ø19     Ø222Ø        JR     NZ,J6EH        ;JUMP IF IT IS
              Ø223Ø ;
              Ø224Ø ;THE FOLLOWING LOGIC POINTS HL TO HASH INDEX SUB RECORD
F146 C1       Ø225Ø        POP    BC             ;RESTORE STACK
F147 DD5E19   Ø226Ø        LD     E,(IX+25)      ;E HAS SUB RECORD NUMBER
F14A 16ØØ     Ø227Ø        LD     D,ØØH          ;DE HAS SUB RECORD NUMBER
F14C D5       Ø228Ø        PUSH   DE             ;SAVE SUB REC NUMBER
F14D EB       Ø229Ø        EX     DE,HL          ;PREPARE TO MULTIPLY SUB REC
                                                 ;BY 3
```

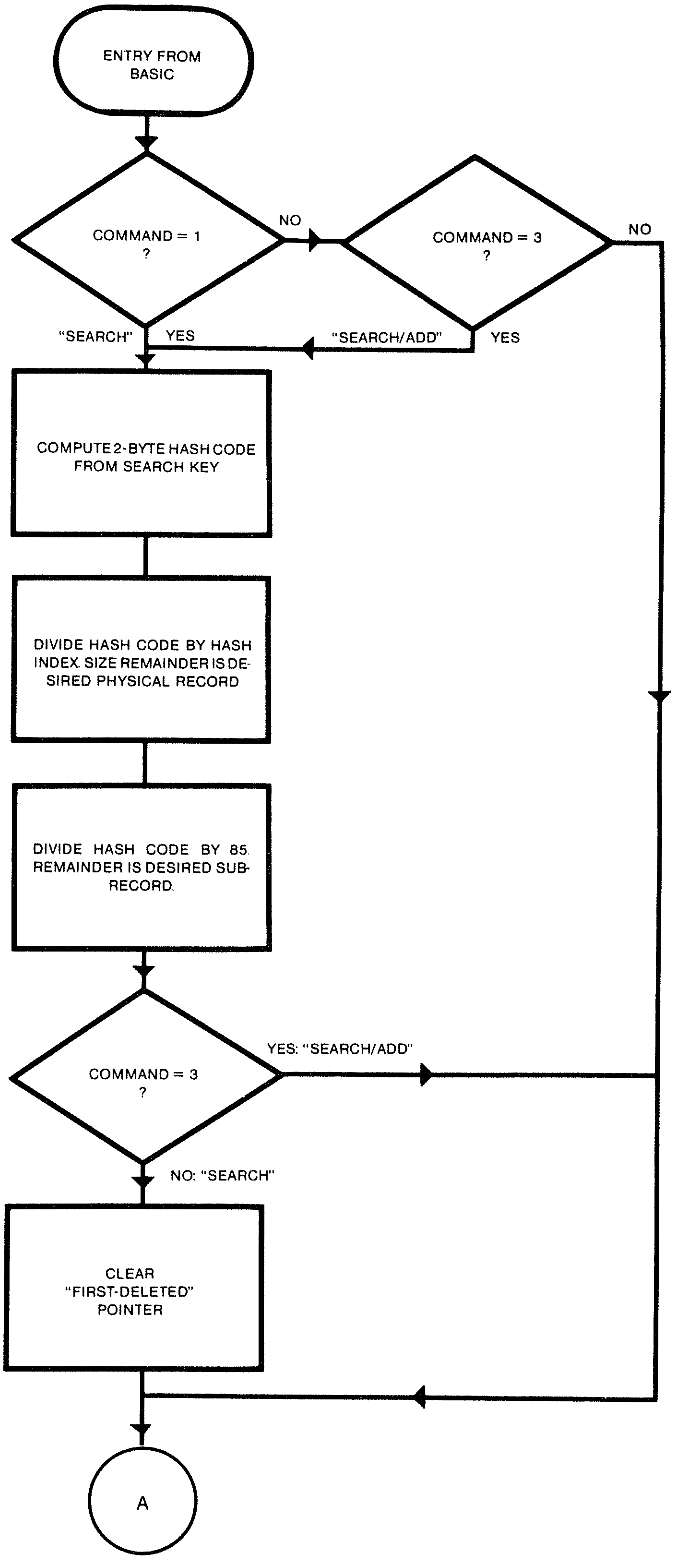
```
F14E 29       Ø23ØØ         ADD     HL,HL           ;MULT SUB REC BY 2
F14F 19       Ø231Ø         ADD     HL,DE           ;
F15Ø D1       Ø232Ø         POP     DE              ;CONTINUE MULTIPLY BY 3
F151 16ØØ     Ø233Ø         LD      D,Ø             ;
F153 19       Ø234Ø         ADD     HL,DE           ;HL POINTS TO SUB RECORD
F154 E5       Ø235Ø         PUSH    HL              ;SAVE POINTER ON STACK
F155 18ØA     Ø236Ø         JR      J1D             ;
              Ø237Ø ;
              Ø238Ø ;THE FOLLOWING LOGIC HANDLES SOME HALF-JUMPS FOR RELOCATABLITY
F157 18BØ     Ø239Ø J4AH    JR      J4AH1           ;
F159 18BØ     Ø24ØØ J3BH    JR      J3BH1           ;
F15B 1866     Ø241Ø J6DH2   JR      J6D             ;
F15D 18AE     Ø242Ø J5AH    JR      J5A             ;
F15F 1879     Ø243Ø J6EH    JR      J6E             ;
              Ø244Ø ;
              Ø245Ø ;THE FOLLOWING LOGIC BRANCHES FOR ADD OR DELETE COMMANDS
F161 3EØ3     Ø246Ø J1D     LD      A,Ø3H           ;COMPARE TO COMMAND 3
F163 DDBEØØ   Ø247Ø         CP      (IX+Ø)          ;COMPARE TO COMMAND FROM BASIC
F166 383D     Ø248Ø         JR      C,J8AH          ;GOTO ADD/DELETE ROUTINES
              Ø249Ø ;
              Ø25ØØ ;THE FOLLOWING LOGIC CONTINUES WITH THE SEARCH COMMANDS
F168 5E       Ø251Ø         LD      E,(HL)          ;
F169 23       Ø252Ø         INC     HL              ;
F16A 56       Ø253Ø         LD      D,(HL)          ;DE HAS REC POINTER FROM DISK
F16B 23       Ø254Ø         INC     HL              ;
F16C 46       Ø255Ø         LD      B,(HL)          ;GET 1-BYTE HASH FROM DISK
F16D CB7A     Ø256Ø         BIT     7,D             ;TEST IF IT IS A DELETED REC
F16F 281E     Ø257Ø         JR      Z,J6A           ;JUMP IT IT'S NOT
F171 DDCB1B7E Ø258Ø         BIT     7,(IX+27)       ;WAS IT FIRST DELETION IN CHAIN
F175 2Ø15     Ø259Ø         JR      NZ,J6I          ;JUMP IF IT WASN'T
F177 3EØ3     Ø26ØØ         LD      A,Ø3H           ;LOAD SEARCH/ADD COMMAND
F179 DDBEØØ   Ø261Ø         CP      (IX+Ø)          ;COMPARE TO COMMAND FROM BASIC
F17C 2827     Ø262Ø         JR      Z,J8AH          ;JUMP IF IT IS
F17E DD7E18   Ø263Ø         LD      A,(IX+24)       ;A=CURRENT PHYSICAL REC
F181 DD771A   Ø264Ø         LD      (IX+26),A       ;SAVE IT IN CASE WE WANT TO ADD
F184 DD7E19   Ø265Ø         LD      A,(IX+25)       ;A=CURRENT SUB REC
F187 CBFF     Ø266Ø         SET     7,A             ;INDICATE WE'VE FOUND FIRST DELETE
F189 DD771B   Ø267Ø         LD      (IX+27),A       ;SAVE IT IN CASE WE WANT TO ADD
F18C E1       Ø268Ø J6I     POP     HL              ;RESTORE STACK
F18D 18C8     Ø269Ø J6B     JR      J4AH            ;GET NEXT HASH REC
F18F EB       Ø27ØØ J6A     EX      DE,HL           ;HL HAS REC POINTER
F19Ø 7C       Ø271Ø         LD      A,H             ;
F191 B5       Ø272Ø         OR      L               ;POINTER = ZERO ?
F192 2Ø13     Ø273Ø         JR      NZ,J6C          ;JUMP IF IT'S NOT
F194 3EØ3     Ø274Ø         LD      A,Ø3H           ;LOAD SEARCH/ADD CODE
F196 DDBEØØ   Ø275Ø         CP      (IX+Ø)          ;COMPARE TO COMMAND FROM BASIC
F199 28ØA     Ø276Ø         JR      Z,J8AH          ;GO TO ADD ROUTINE
F19B D1       Ø277Ø         POP     DE              ;OTHERWISE, RESTORE STACK
F19C DD75Ø2   Ø278Ø         LD      (IX+2),L        ;
F19F DD74Ø3   Ø279Ø         LD      (IX+3),H        ;LOAD ZERO TO RECORD NUMBER
F1A2 C39AØA   Ø28ØØ         JP      ØA9AH           ;AND RETURN "NOT-FOUND" TO BASIC
              Ø281Ø ;
```

```
                  Ø282Ø ;THE FOLLOWING LOGIC HANDLES SOME HALF-JUMPS FOR RELOCATABILITY
F1A5 1858         Ø283Ø J8AH    JR      J8AH1           ;
                  Ø284Ø ;
                  Ø285Ø ;THE FOLLOWING LOGIC GETS KEY FROM MASTER FILE
F1A7 D1           Ø286Ø J6C     POP     DE              ;RESTORE STACK
F1A8 3EØ3         Ø287Ø         LD      A,Ø3H           ;LOAD SEARCH/ADD COMMAND
F1AA DDBEØØ       Ø288Ø         CP      (IX+Ø)          ;COMPARE TO COMMAND FROM BASIC
F1AD 28DE         Ø289Ø         JR      Z,J6B           ;GET ANOTHER HASH INDEX IF IT IS
F1AF 78           Ø29ØØ         LD      A,B             ;1-BYTE HASH TO ACCUM
F1BØ DDBE16       Ø291Ø         CP      (IX+22)         ;COMPARE TO HASH FOR SKEY
F1B3 2ØD8         Ø292Ø         JR      NZ,J6B          ;GET NEXT HASH INDEX IF NOT EQUAL
F1B5 DD75Ø2       Ø293Ø         LD      (IX+2),L        ;
F1B8 DD74Ø3       Ø294Ø         LD      (IX+3),H        ;SAVE RECORD NUMBER
F1BB DD56ØA       Ø295Ø         LD      D,(IX+1Ø)       ;DIVISOR IS BLOCKING FACTOR
F1BE 2B           Ø296Ø         DEC     HL              ;DIVIDEND IS REC# - 1
F1BF Ø6Ø1         Ø297Ø         LD      B,Ø1H           ;LOAD RETURN INDICATOR
F1C1 1896         Ø298Ø         JR      J3BH            ;GO DIVIDE REC#/BLOCK FACTOR
F1C3 E5           Ø299Ø J6D     PUSH    HL              ;SAVE REMAINDER
F1C4 FDE5         Ø3ØØØ         PUSH    IY              ;
F1C6 E1           Ø3Ø1Ø         POP     HL              ;HL HAS QUOTIENT
F1C7 DD5EØ6       Ø3Ø2Ø         LD      E,(IX+6)        ;
F1CA DD56Ø7       Ø3Ø3Ø         LD      D,(IX+7)        ;DE HAS MASTER FILE DCB ADRESS
F1CD D5           Ø3Ø4Ø         PUSH    DE              ;
F1CE FDE1         Ø3Ø5Ø         POP     IY              ;IY HAS MASTER FILE DCB ADDRESS
F1DØ DD4EØ8       Ø3Ø6Ø         LD      C,(IX+8)        ;
F1D3 DD46Ø9       Ø3Ø7Ø         LD      B,(IX+9)        ;BC HAS MASTER FILE OFFSET
F1D6 CBC3         Ø3Ø8Ø         SET     Ø,E             ;SET RETURN INDICATOR
F1D8 1883         Ø3Ø9Ø         JR      J5AH            ;GO GET THE RECORD
F1DA E3           Ø31ØØ J6E     EX      (SP),HL         ;HL=REMAINDER, BUFFER ADDR ON STACK
F1DB DD741E       Ø311Ø         LD      (IX+3Ø),H       ;LOAD PRECEDING SUBRECORDS
F1DE 4C           Ø312Ø         LD      C,H             ;REMAINDER WILL BE MULTIPLICAND
F1DF DD5EØC       Ø313Ø         LD      E,(IX+12)       ;
F1E2 DD56ØD       Ø314Ø         LD      D,(IX+13)       ;DE HAS LOGICAL REC LENGTH
F1E5 21ØØØØ       Ø315Ø         LD      HL,ØØH          ;MULTIPLY DE BY C
F1E8 CB39         Ø316Ø J6F     SRL     C               ;CONTINUE...
F1EA 3ØØ1         Ø317Ø         JR      NC,J6G          ;CONTINUE...
F1EC 19           Ø318Ø         ADD     HL,DE           ;CONTINUE...
F1ED 28Ø5         Ø319Ø J6G     JR      Z,J6H           ;CONTINUE...
F1EF EB           Ø32ØØ         EX      DE,HL           ;CONTINUE...
F1FØ 29           Ø321Ø         ADD     HL,HL           ;CONTINUE...
F1F1 EB           Ø322Ø         EX      DE,HL           ;CONTINUE...
F1F2 18F4         Ø323Ø         JR      J6F             ;CONTINUE...
F1F4 D1           Ø324Ø J6H     POP     DE              ;GET BUFFER ADDRESS
F1F5 19           Ø325Ø         ADD     HL,DE           ;HL POINTS TO SUB RECORD
F1F6 DD5EØE       Ø326Ø         LD      E,(IX+14)       ;
F1F9 DD56ØF       Ø327Ø         LD      D,(IX+15)       ;DE HAS KEY OFFSET
F1FC 19           Ø328Ø         ADD     HL,DE           ;HL POINTS TO KEY IN SUB RECORD
F1FD 18Ø4         Ø329Ø         JR      J7              ;
                  Ø33ØØ ;
                  Ø331Ø ;THE FOLLOWING LOGIC HANDLES SOME HALF-JUMPS FOR RELOCATABILITY
F1FF 182C         Ø332Ø J8AH1   JR      J8A             ;
F2Ø1 188A         Ø333Ø J6BH    JR      J6B             ;
```

```
                Ø334Ø ;
                Ø335Ø ;THE FOLLOWING LOGIC COMPARES SEARCH KEY.TO KEY IN MASTER FILE
F2Ø3 E5         Ø336Ø J7      PUSH    HL              ;SAVE POINTER TO KEY IN MF
F2Ø4 DD6EØ4     Ø337Ø         LD      L,(IX+4)        ;
F2Ø7 DD66Ø5     Ø338Ø         LD      H,(IX+5)        ;HL HAS SKEY VARPTR
F2ØA 46         Ø339Ø         LD      B,(HL)          ;B HAS SKEY LENGTH
F2ØB 23         Ø34ØØ         INC     HL              ;
F2ØC 5E         Ø341Ø         LD      E,(HL)          ;
F2ØD 23         Ø342Ø         INC     HL              ;
F2ØE 56         Ø343Ø         LD      D,(HL)          ;DE POINTS TO SKEY
F2ØF E1         Ø344Ø         POP     HL              ;HL POINTS TO KEY ON DISK
F21Ø 1A         Ø345Ø J7A     LD      A,(DE)          ;SKEY CHARACTER TO ACCUM
F211 BE         Ø346Ø         CP      (HL)            ;COMPARE TO CHARACTER ON DISK
F212 2ØED       Ø347Ø         JR      NZ,J6BH         ;IF NOT EQUAL, NEXT HASH INDEX
F214 23         Ø348Ø         INC     HL              ;POINT TO NEXT ON DISK
F215 13         Ø349Ø         INC     DE              ;POINT TO NEXT IN SKEY
F216 1ØF8       Ø35ØØ         DJNZ    J7A             ;REPEAT FOR NEXT PAIR
F218 FD6EØA     Ø351Ø         LD      L,(IY+1Ø)       ;
F21B FD66ØB     Ø352Ø         LD      H,(IY+11)       ;HL=PHYSICAL REC#
F21E DD751C     Ø353Ø         LD      (IX+28),L       ;
F221 DD741D     Ø354Ø         LD      (IX+29),H       ;RET IT TO BASIC IN CNTRL 14
F224 DD6EØ2     Ø355Ø J8C     LD      L,(IX+2)        ;
F227 DD66Ø3     Ø356Ø         LD      H,(IX+3)        ;HL HAS REC NUMBER OF MATCH
F22A C39AØA     Ø357Ø         JP      ØA9AH           ;RETURN RECORD# TO BASIC
                Ø358Ø ;
                Ø359Ø ;THE FOLLOWING LOGIC HANDLES ADDS AND DELETES TO HASH INDEX
F22D E1         Ø36ØØ J8A     POP     HL              ;RSTORE PNTR TO HASH SUB REC
F22E 3EØ4       Ø361Ø         LD      A,Ø4H           ;LOAD CODE FOR "ADD"
F23Ø DDBEØØ     Ø362Ø         CP      (IX+Ø)          ;COMPARE TO CMD FROM BASIC
F233 381Ø       Ø363Ø         JR      C,J8B           ;JUMP IF DELETE COMMAND
F235 DD7EØ2     Ø364Ø         LD      A,(IX+2)        ;LOAD LSB OF RECORD POINTER
F238 77         Ø365Ø         LD      (HL),A          ;WRITE TO BUFFER
F239 23         Ø366Ø         INC     HL              ;
F23A DD7EØ3     Ø367Ø         LD      A,(IX+3)        ;LOAD MSB OF RECORD POINTER
F23D 77         Ø368Ø         LD      (HL),A          ;WRITE IT TO BUFFER
F23E 23         Ø369Ø         INC     HL              ;
F23F DD7E16     Ø37ØØ         LD      A,(IX+22)       ;LOAD 1-BYTE HASH
F242 77         Ø371Ø         LD      (HL),A          ;WRITE IT TO BUFFER
F243 18ØB       Ø372Ø         JR      J9A             ;
F245 3EØØ       Ø373Ø J8B     LD      A,ØØH           ;ZERO POINTER MEANS DELETION
F247 77         Ø374Ø         LD      (HL),A          ;WRITE IT TO BUFFER
F248 23         Ø375Ø         INC     HL              ;
F249 CBFF       Ø376Ø         SET     7,A             ;SET BIT 7 TO FLAG DELETION
F24B 77         Ø377Ø         LD      (HL),A          ;WRITE IT TO BUFFER
F24C 23         Ø378Ø         INC     HL              ;SET ZERO FOR HASH
F24D 3EØØ       Ø379Ø         LD      A,ØØH           ;
F24F 77         Ø38ØØ         LD      (HL),A          ;WRITE TO BUFFER
                Ø381Ø ;
                Ø382Ø ;THE FOLLOWING LOGIC WRITES RECORD TO DISK
F25Ø FDE5       Ø383Ø J9A     PUSH    IY              ;
F252 D1         Ø384Ø         POP     DE              ;DE HAS DCB ADDRESS
F253 FD6EØA     Ø385Ø         LD      L,(IY+1Ø)       ;
```

```
F256 FD66ØB   Ø386Ø        LD      H,(IY+11)       ;HL HAS NRN (NXT REC NUMBER)
F259 2B       Ø387Ø        DEC     HL              ;PUT IT BACK TO CURRENT
F25A FD75ØA   Ø388Ø        LD      (IY+1Ø),L       ;
F25D FD74ØB   Ø389Ø        LD      (IY+11),H       ;NRN IS ADJUSTED
F26Ø DDE5     Ø39ØØ        PUSH    IX              ;SAVE DURING DOS CALL
F262 FDE5     Ø391Ø        PUSH    IY              ;SAVE DURING DOS CALL
F264 CD3944   Ø392Ø        CALL    4439H           ;WRITE THE RECORD
F267 FDE1     Ø393Ø        POP     IY              ;RESTORE IY
F269 DDE1     Ø394Ø        POP     IX              ;RESTORE IX
F26B 28B7     Ø395Ø        JR      Z,J8C           ;RET TO BASIC IF NO ERROR
F26D 26FF     Ø396Ø        LD      H,ØFFH          ;
F26F 2F       Ø397Ø        CPL                     ;
F27Ø 6F       Ø398Ø        LD      L,A             ;
F271 23       Ø399Ø        INC     HL              ;HL HAS MINUS OF ERROR CODE
F272 C39AØA   Ø4ØØØ        JP      ØA9AH           ;RETURN ERROR CODE TO BASIC
              Ø4Ø1Ø        END                     ;
ØØØØØ TOTAL ERRORS
```

Figure 14.4a — *KEYACCES Flowchart*


ENTRY FROM BASIC
COMMAND = 1 ?
NO
COMMAND = 3 ?
NO
"SEARCH"
YES
"SEARCH/ADD"
YES
COMPUTE 2-BYTE HASH CODE FROM SEARCH KEY
DIVIDE HASH CODE BY HASH INDEX SIZE REMAINDER IS DESIRED PHYSICAL RECORD
DIVIDE HASH CODE BY 85 REMAINDER IS DESIRED SUB-RECORD.
COMMAND = 3 ?
YES: "SEARCH/ADD"
NO: "SEARCH"
CLEAR "FIRST-DELETED" POINTER
A

COMMANDS

1 = SEARCH
2 = RESUME SEARCH
3 = SEARCH/ADD
4 = ADD
5 = DELETE

Figure 14.4b — *KEYACCES Flowchart*

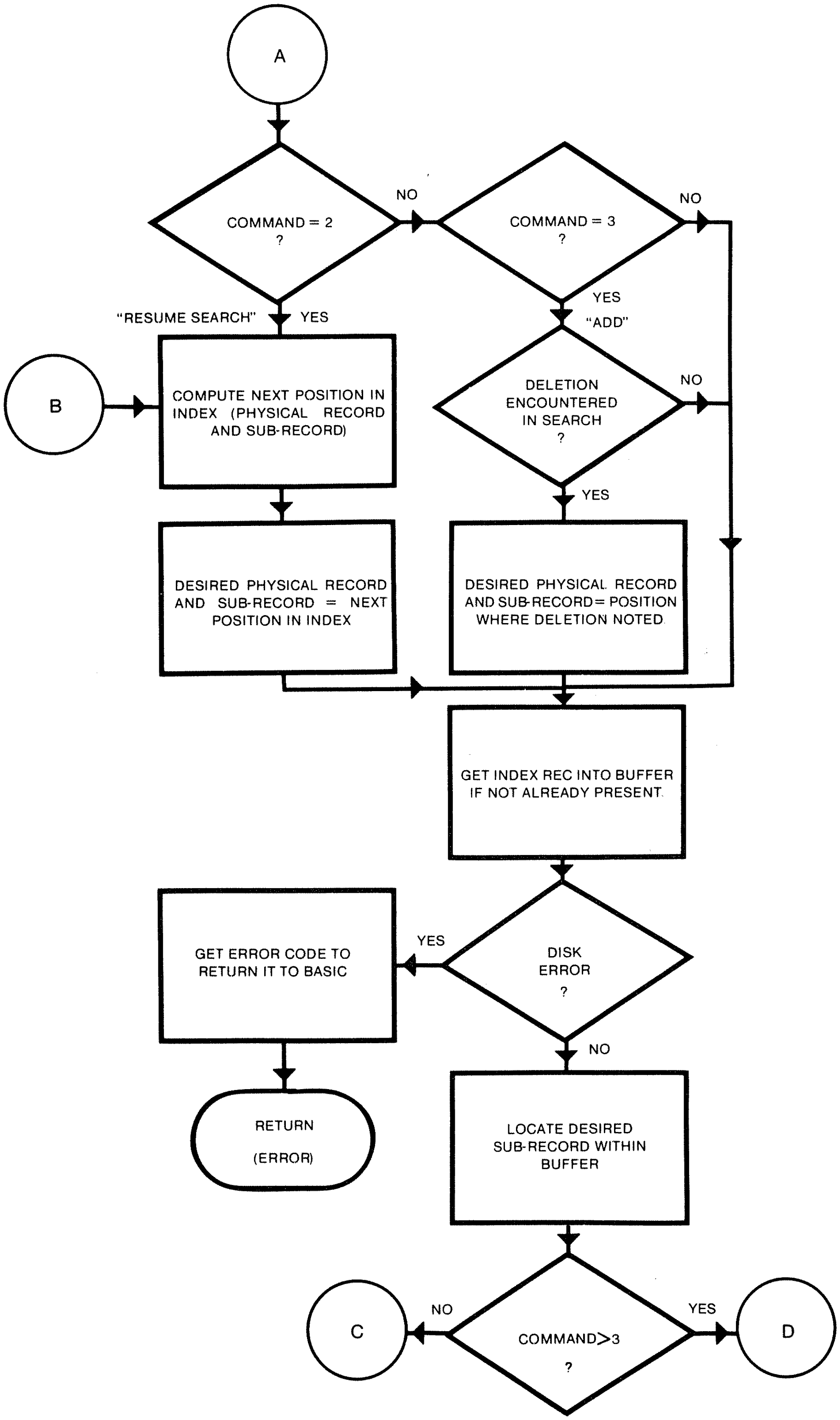

A
COMMAND = 2 ?
NO
COMMAND = 3 ?
NO
"RESUME SEARCH"
YES
YES
"ADD"
B
COMPUTE NEXT POSITION IN INDEX (PHYSICAL RECORD AND SUB-RECORD)
DELETION ENCOUNTERED IN SEARCH ?
NO
YES
DESIRED PHYSICAL RECORD AND SUB-RECORD = NEXT POSITION IN INDEX
DESIRED PHYSICAL RECORD AND SUB-RECORD= POSITION WHERE DELETION NOTED
GET INDEX REC INTO BUFFER IF NOT ALREADY PRESENT
GET ERROR CODE TO RETURN IT TO BASIC
YES
DISK ERROR ?
NO
RETURN (ERROR)
LOCATE DESIRED SUB-RECORD WITHIN BUFFER
C
NO
COMMAND>3 ?
YES
D

Figure 14.4c — *KEYACCES Flowchart*

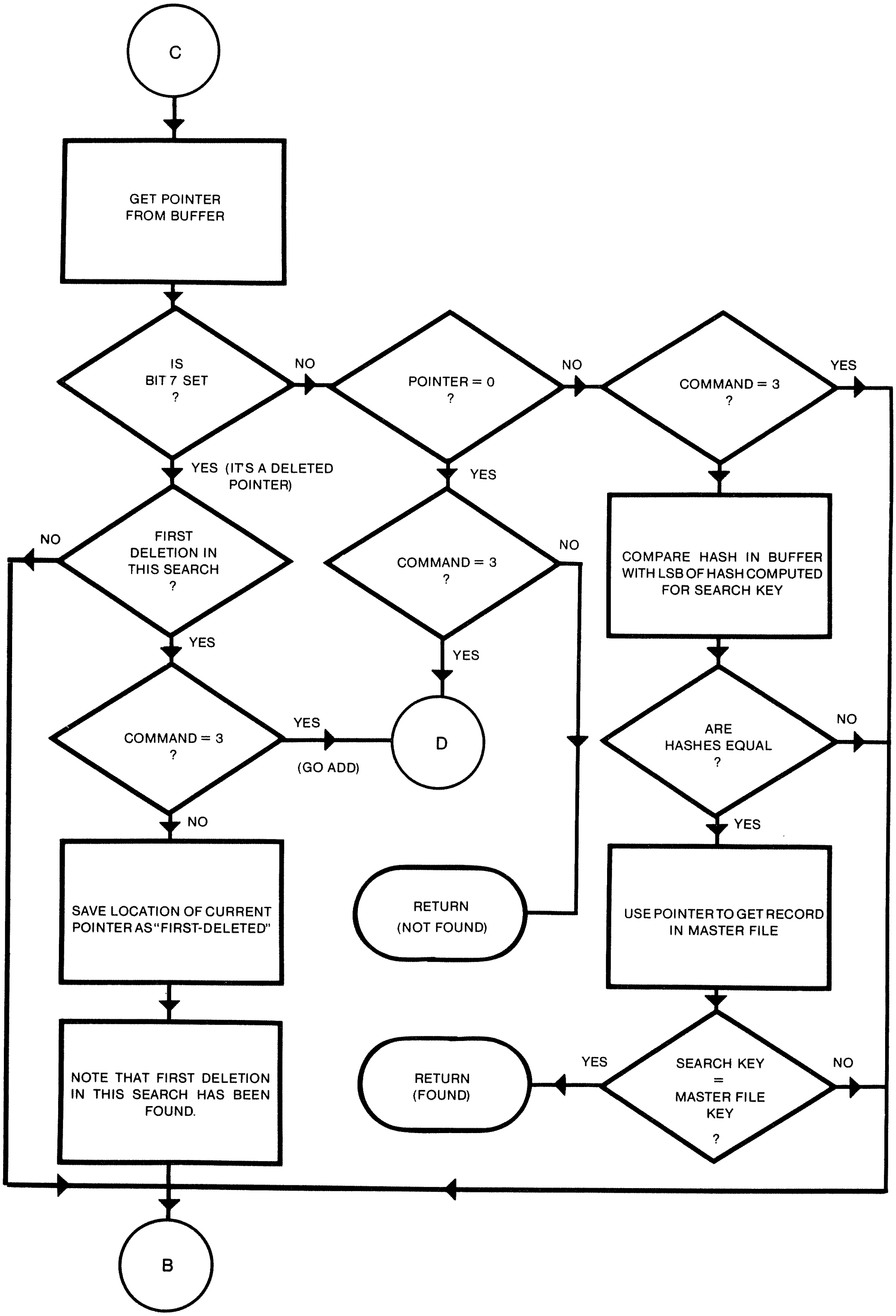
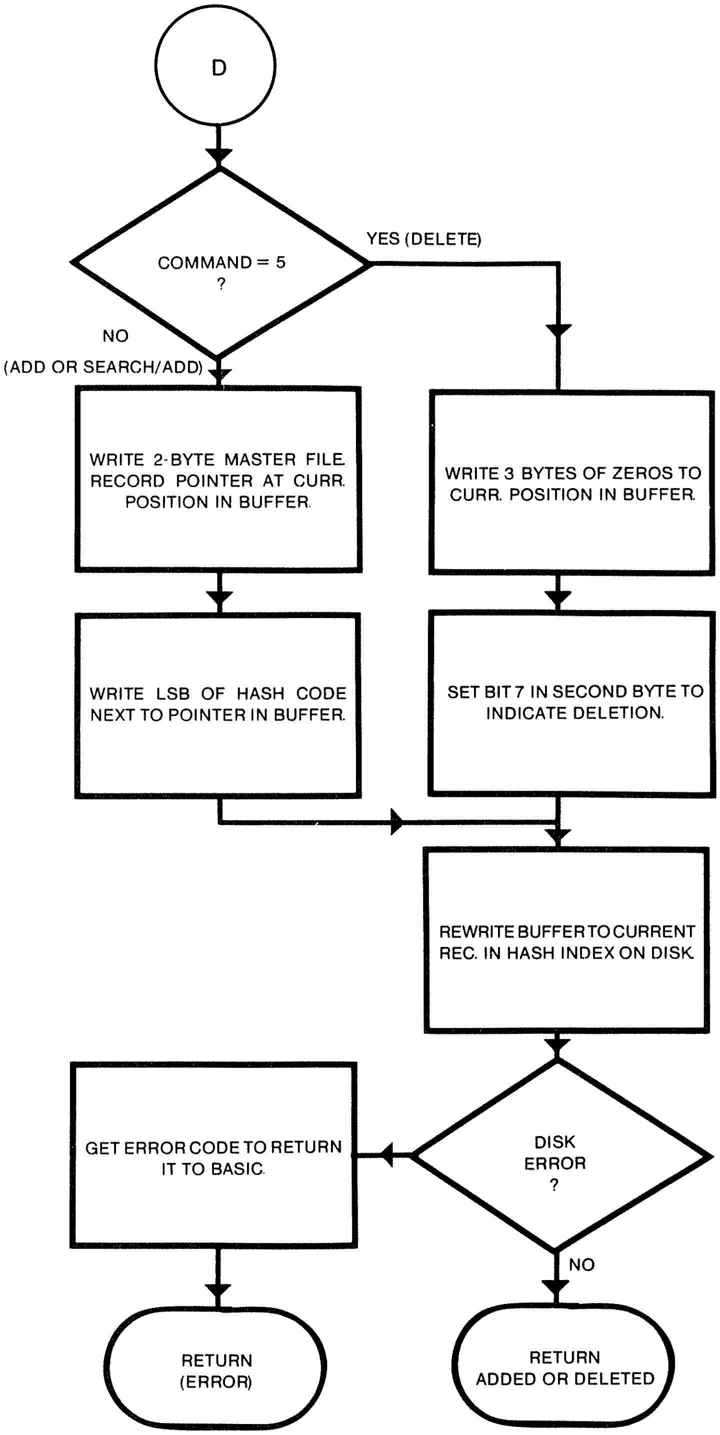

C
GET POINTER FROM BUFFER
IS BIT 7 SET ?
NO
POINTER = 0 ?
NO
COMMAND = 3 ?
YES
YES (IT'S A DELETED POINTER)
YES
NO
FIRST DELETION IN THIS SEARCH ?
COMMAND = 3 ?
NO
COMPARE HASH IN BUFFER WITH LSB OF HASH COMPUTED FOR SEARCH KEY
YES
YES
COMMAND = 3 ?
YES
D
(GO ADD)
ARE HASHES EQUAL ?
NO
NO
YES
SAVE LOCATION OF CURRENT POINTER AS "FIRST-DELETED"
RETURN (NOT FOUND)
USE POINTER TO GET RECORD IN MASTER FILE
NOTE THAT FIRST DELETION IN THIS SEARCH HAS BEEN FOUND.
RETURN (FOUND)
YES
SEARCH KEY = MASTER FILE KEY ?
NO
B

Figure 14.4d — *KEYACCES Flowchart*


D
COMMAND = 5
?
YES (DELETE)
NO
(ADD OR SEARCH/ADD)
WRITE 2-BYTE MASTER FILE RECORD POINTER AT CURR. POSITION IN BUFFER.
WRITE 3 BYTES OF ZEROS TO CURR. POSITION IN BUFFER.
WRITE LSB OF HASH CODE NEXT TO POINTER IN BUFFER.
SET BIT 7 IN SECOND BYTE TO INDICATE DELETION.
REWRITE BUFFER TO CURRENT REC. IN HASH INDEX ON DISK.
DISK
ERROR
?
GET ERROR CODE TO RETURN IT TO BASIC.
NO
RETURN
(ERROR)
RETURN
ADDED OR DELETED

KEYACCES/DEM

Implementing KEYACCES in a BASIC program can be a rather involved process, especially the first time around. You have to understand how it works and what the different commands do. Beyond that, the parameters in your control array must be correct. KEYACCES/DEM is a demonstration program that will help you to get familiar with the capabilities of KEYACCES, and to test it to your satisfaction.

The organization and operation of KEYACCES/DEM is similar to TREESAM/BAS. You can look at the contents of the relevant variables, you are able to try the different commands, and you may examine how the index is stored on disk. Unlike TREESAM/BAS, KEYACCES/DEM is not intended to be a utility program. You have no control over the file names or key and data field lengths unless you modify the program.

To run KEYACCES/DEM you must first get KEYACCES into memory at **FOOO**. (That's –4096 in integer notation.) You can do this by running KALOADER/BAS first. Upon starting KEYACCES/DEM, you get the following message on your display:

```
KEYACCES DEMONSTRATION PROGRAM

THIS PROGRAM SETS UP A MASTERFILE AND INDEX
EACH MASTERFILE RECORD IS 32 BYTES LONG AND CONTAINS 2 FIELDS:
FIELD 1 IS AN 8-BYTE KEY, FIELD 2 IS A 24-BYTE DATA FIELD.
FOR THE DEMO, YOU MAY HAVE UP TO 254 LOGICAL RECORDS.

KEYACCES MUST BE IN MEMORY AT -4Ø96
IF IT IS NOT, PRESS <BREAK> AND RUN KALOADER/BAS
OTHERWISE PRESS <ENTER>...
```

As the message indicates, each logical record in the master file is 32 bytes long. Since KEYACCES/DEM uses the random file handler, it automatically handles the blocking of the 32-byte logical records into 256-byte physical records. The two fields for the demonstration are simply called the "key field" and the "data field". The key field is 8 bytes, and the data field is 24 bytes. The information that you put into the fields is up to you. As an example, your keys could be catalog numbers and your data fields could contain product descriptions. Really, though, the purpose at this time is to get familar with KEYACCES, so the demonstration program doesn't label the fields specifically.

The demonstration is limited to 254 logical records. This limit was chosen only to make it so that an inordinate amount of disk space wouldn't be required. If you wish, you can change the program to provide for more records or fewer. A file containing three physical records is used for the index. That requires 1 gran on TRS-80 Models I, II, and III. A file containing 32 physical records is created for the master file. That's 7 granules with Model I or II TRSDOS, or 11 granules with Model III TRSDOS. You may want to go back to DOS to make sure the disk you're using has enough space before you start the demonstration.

The other thing we can note from the startup message is that KEYACCES/DEM does no checking to verify that KEYACCES is in memory. It is up to you to make sure it has been loaded.

The next message shown on the screen asks what disk operating system you are using:

```
ARE YOU USING:
<A> TRSDOS 1.3 ON A MODEL 3
<B> TRSDOS 2.3 ON A MODEL 1
<C> TRSDOS 2.Ø ON A MODEL 2
<D> SOMETHING ELSE?
```

You answer this question by pressing "A", "B", "C" or "D". KEYACCES/DEM has the disk data control block addresses for the three most popular versions of TRSDOS. If you are using a different disk operating system you will need to press "D" to enter the file 1 and file 2 DCB addresses yourself. (See Appendix 1 for an explanation of how to find DCB addresses.)

In the event you select "D" the prompt displayed is:

```
FILE 1 DATA CONTROL BLOCK ADDRESS?
```

If, for example, you are using NEWDOS80 Version 2.0 you enter 26315 here.

Next, in the event you selected "D", the program asks you for the file 2 data control block address. (To continue the example of NEWDOS80 2.0, your response is 26616.)

Now KEYACCES/DEM tells you what files it will use for the demonstration:

```
TWO FILES ARE USED FOR THE DEMONSTRATION:

FILE 1 IS MASTER FILE, KADEM/MAS:Ø, LENGTH = 32 RECORDS
FILE 2 IS INDEX  FILE, KADEM/IND:Ø, LENGTH =  3 RECORDS

PRESS <ENTER> TO OPEN THEM, OTHERWISE PRESS <BREAK>...
```

"KADEM/MAS" is used as the master file and "KADEM/IND" is used as the hash file index. Both will be opened on drive 0 unless you change the program. Next, KEYACCES/DEM may ask you:

```
DO YOU WANT TO INITIALIZE THEM TO AN EMPTY CONDITION?
```

This question is asked only if both files already exist on the diskette in drive 0, so that on subsequent runs of the program you can elect to retain the information they contain or start from a clean slate. Valid responses are "Y" for yes or "N" for no. If either file has a length of zero records the question is skipped — KEYACCES/DEM assumes that you do want to initialize the files.

The initialization process creates the index by writing three records containing 256 hexadecimal zeros. It creates the master file by writing 32 records containing 256 hexadecimal zeros. Then it calls the labeled file handler subroutine that sets up the file statistics for the master file:

```
Next logical record:     2  (Record 1 is used for statistics.)
Last record deleted:     Ø
Logical records active:  Ø
File validity:           Ø  (Not used in this program.)
File synchronization:    Ø  (Not used in this program.)
```

After the files have been opened and initialized, the menu is displayed:

```
KEYACCES DEMONSTRATION PROGRAM
================================================================

<1> DISPLAY CONTROL INFORMATION AND STATISTICS
<2> SEARCH, ADD, DELETE
<3> DISPLAY HASH INDEX
<4> LIST MASTER FILE RECORDS
<5> CLOSE FILES AND END
================================================================
PRESS THE NUMBER OF YOUR SELECTION...
```

Displaying the Control Array

Menu selection 1 displays the contents of the KEYACCES control array, and it shows how many logical records are active. The display will change as you add and delete records. The illustration shows how it might appear before any activity has taken place when the program is run on a Model II:

```
CONTROL ARRAY CONTENTS
===============================================================
KA( Ø )= Ø                          KA( 8 )= 28549
KA( 1 )= Ø                          KA( 9 )= Ø
KA( 2 )=-27223                      KA( 1Ø )= 3
KA( 3 )= 27715                      KA( 11 )= Ø
KA( 4 )= Ø                          KA( 12 )= Ø
KA( 5 )= 8                          KA( 13 )= Ø
KA( 6 )= 32                         KA( 14 )= Ø
KA( 7 )= Ø                          KA( 15 )= Ø

MASTER FILE CONTAINS Ø ACTIVE LOGICAL RECORDS
THIS DEMO LIMITS YOU TO 254 ACTIVE LOGICAL RECORDS
===============================================================
PRESS <ENTER>...
```

Search, Add, Delete

Selection 2 from the main menu lets you test the search, add and delete capabilities of KEYACCES. Upon pressing "2" you get the following display:

```
SF, SN           SEARCH - FIRST, NEXT
AU, AN, AF, AC   ADD    - UNIQUE, NON-UNIQUE, FIFO, CURRENT
DC               DELETE - CURRENT
------------- UP-ARROW RETURNS TO MENU -------------------
COMMAND: ..
```

This section of the KEYACCES/DEM program accepts seven different 2-letter commands. "SF" is "Search First", "SN" is "Search Next", and so forth. You may give any command to the program by typing the two letters and pressing <ENTER>. Simply pressing <ENTER> repeats the last command you supplied. Any invalid command redisplays the list of commands that are available. When you have finished using this section of the program, you may press <up-arrow> and <ENTER> to return to the main menu. <Up-arrow> <ENTER> also allows you to abort any entry in this section of the program, returning to the prior entry. (Model II users may press <shift> <left-bracket> <ENTER> instead of <up-arrow>.)

The "SF" command calls KEYACCES command 1, "search". It is used to access a record in the master file by key. If no key matches the search key you supply, a "not found" message is returned:

```
COMMAND: SF
SEARCH FIRST          KEY:  JOE
                      ** NOT FOUND **
COMMAND: ..
```

The "AC" command calls KEYACCES command 4, "add". It may be used only after a search that has resulted in a "not found" condition. The key to be added is the key that was used in the last search. You supply the information that is to go into the data field. The program adds the key and data to the next available record in the master file and updates the index file so that subsequent searches can access it:

```
COMMAND: AC
ADD CURRENT              DATA: JONES
ADDING AS RECORD 2
COMMAND: ..
```

Having added the key "JOE" with a data field that contains "JONES", you can test the "SF" command again:

```
COMMAND: SF
SEARCH FIRST             KEY:  JOE
FOUND IN RECORD 2        KEY:  JOE
                         DATA: JONES
```

The "SN" command calls KEYACCES in "resume search" mode. After a search-first (or a search-next) it lets you see if there are any more duplicates for the key you supplied. Search-next redisplays the key that is active, and checks the hash index and master file:

```
COMMAND: SN
SEARCH NEXT              KEY:  JOE
                         ** NOT FOUND **
COMMAND: ..
```

Here it told you that there are no duplicate records in the master file having a key of "JOE". Suppose, though, that you want to add a "JOE SMITH" to the file. You can do it at this point with the add current command, since the search has ended in a "not found" condition:

```
COMMAND: AC
ADD CURRENT              DATA: SMITH
ADDING AS RECORD 3
COMMAND: ..
```

Now you have two records in the master file having a key of "JOE". You can redisplay them with "SF" and "SN":

```
COMMAND: SF
SEARCH FIRST             KEY:  JOE
FOUND IN RECORD 2        KEY:  JOE
                         DATA: JONES
COMMAND: SN
SEARCH NEXT              KEY:  JOE
FOUND IN RECORD 3        KEY:  JOE
                         DATA: SMITH
COMMAND: SN
SEARCH NEXT              KEY:  JOE
                         ** NOT FOUND **
COMMAND: ..
```

The last "search next" resulted in a "not found" showing you that there are no more duplicates in the file having "JOE" in the key field.

The KEYACCES/DEM program expands KEYACCES command 3, "search/add", and KEYACCES command 4, "add" into 4 different add commands. A study of the way they are programmed in KEYACCES/DEM can help you to decide how to organize the add capability in your own programs.

The "AU" command is "add unique". It is a one-step add command that accepts the key and data fields, and adds them to the master file only if no other key field in the master file matches the new key. It automatically calls KEYACCES in "search" mode and then "add" mode:

```
COMMAND: AU
ADD UNIQUE                KEY:  BILL
                          DATA: ADAMS
ADDING AS RECORD 4
COMMAND: ..
```

If you attempt an "AU" with a key that is already in the master file, you get a message that a duplicate exists:

```
COMMAND: AU
ADD UNIQUE                KEY:  JOE
                          DATA: WILLIAMS
                          ** DUPLICATE EXISTS - NOT ADDED **
COMMAND: ..
```

"AN" and "AF" provide two ways to add new records to the master file without checking for duplicates. They both use KEYACCES command 3, and for most purposes they act identically. The only difference is that "AN" recovers deleted positions in the hash file index, making it so that duplicates won't necessarily be recalled in the sequence that they were added. "AF" does not recover deleted positions, so that duplicate keys will be returned with "search next" commands on a "first in, first out" basis.

"AN" is "add non-unique". You simply enter the key field and data field, and the new record is added:

```
COMMAND: AN
ADD NON-UNIQUE            KEY:  JOE
                          DATA: JOHNSON
ADDING AS RECORD 5
COMMAND: ..
```

"AF" is "add FIFO". Like add-non-unique, you just enter the key and data:

```
COMMAND: AF
ADD FIFO                  KEY:  JOE
                          DATA: WILLIAMS
ADDING AS RECORD 6
```

You won't be able to see any difference between "AN" and "AF" until you make some deletions.

We've already discussed the "AC" command. KEYACCES/DEM does not let you use "AC" unless the previous command was a "SF" or a "SN" that resulted in a "not found".

"Add current" calls KEYACCES with command 5, "add", which assumes that you've already positioned to the correct record and subrecord in the hash index.

"DC" is "delete current". It calls KEYACCES with command 5, "delete". Like "AC" it must follow certain commands that bring you to a desired position in the hash index. "DC" must follow a "SF" or "SN" command that has resulted in a "found" condition. KEYACCES/DEM insures that this is the case. Suppose you want to delete "BILL" from the master file and index. You do a "SF" and then a "DC":

```
COMMAND: SF
SEARCH FIRST            KEY:  BILL
FOUND IN RECORD 4       KEY:  BILL
                        DATA: ADAMS
COMMAND: DC
DELETE CURRENT          DELETING FROM MASTER FILE
                        DELETING FROM INDEX
COMMAND: ..
```

Now let's say you want to delete the second "JOE" record from the files. The commands are:

```
COMMAND: SF
SEARCH FIRST            KEY:  JOE
FOUND IN RECORD 2       KEY:  JOE
                        DATA: JONES
COMMAND: SN
SEARCH NEXT             KEY:  JOE
FOUND IN RECORD 3       KEY:  JOE
                        DATA: SMITH
COMMAND: DC
DELETE CURRENT          DELETING FROM MASTER FILE
                        DELETING FROM INDEX
COMMAND:
```

Now if you add another "JOE" key with the "AN" command, it will be recalled as the second "JOE". On the other hand, if you were to add another "JOE" with the "AF" command, it would be recalled as the last duplicate "JOE" key.

Try as many combinations of the search, add and delete commands as you wish. The whole purpose is to help familiarize yourself with the way that the KEYACCES commands work and to get a feel for the speed that this accessing method can provide. When you are finished, you can return to the menu by pressing <up-arrow> <ENTER>, (or <shift> <left-bracket> <ENTER>.)

Viewing the Hash Index

Menu selection 3 lets you display the hash file index. As you know from the technical discussions, each physical record in a KEYACCES hash index contains 85 subrecords. Each subrecord may be in one of three states: active, empty or deleted. An active hash index subrecord contains a pointer to a master file record number and a 1-byte hash code with a value ranging from 1 to 255. An empty subrecord is one that hasn't been used, so it contains zeros. A deleted hash index subrecord has the most significant bit set in the pointer field. (All other bits are 0, so a deleted subrecord has a value of -32768. It is sufficient, though, for the program to simply check for a negative pointer field to identify a deleted entry.) KEYACCES/DEM takes these conditions into consideration so you can see the structure of the hash index.

When you press <3> from the main menu, the screen will show the contents of the hash index physical record accessed by the last command you executed while in "search, add, delete" mode. In the case that you did not use option 2 from the main menu, the first hash index record is shown.

Let's say, for example, that you did a "SF" for "JOE" after the additions and deletions that were performed in the prior discussion. The search for "JOE" just happened to hash to record 2, subrecord 35 in the index. A hash code of 146 was generated from the key. The following is displayed:

```
LAST COMMAND:  HASH RECORD 2 , SUBRECORD 35 , HASH CODE 146
   ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,
   ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,
   ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,
   ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,
   ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,
   ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,    2 146,  DEL   Ø,
     5 146,    6 146,  ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,
   ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,
   ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,
   ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,
   ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,
   ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,
   ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,
   ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,  ...   Ø,
   ...   Ø,      <N> NEXT, <P> PREV, OR <ENTER> TO EXIT...
```

As you can see, the 35th position in the record has a "2 146". This tells KEYACCES to check record 2 in the master file if a search key generates a hash code of 146. Notice that there are 2 other positions in the index that are marked with a 1-byte hash code of 146. These are the duplicate "JOE" keys, pointing to masterfile records 5 and 6. The "DEL" in position 36 shows that an entry was deleted from the index. Now, if we add another "JOE" key with the "AN" command it will replace the "DEL". If you add another "JOE" with "AF", the pointer and hash code will go into position 39.

After displaying a record from the hash index, you can press <N> to display the next record. Repeated depressions of <N> cause the display to circle back to record 1, just as KEYACCES does when it searches the index. Since the demonstration program has a hash index of 3 records, <N> shows what's in record 3, and another <N> shows what's in record 1. <P> circles through the index in the reverse direction. <ENTER> takes you back to the KEYACCES/DEM program menu.

Listing the Master File

Selection 4 from the menu lists the records in the master file. If you've been following the sample entries, pressing <4> gives the following display:

```
RECORD #    KEY FIELD     DATA FIELD
===============================================================
     2      JOE           JONES
     3      DELETED
     4      DELETED
     5      JOE           JOHNSON
     6      JOE           WILLIAMS
END OF FILE...PRESS <ENTER>
```

You can see that 3 records are active currently, and 2 positions in the master file are empty because of the deletions. If you wish, you can see how the labeled files handler recovers the deletions in the master file: Go back and add "DON WILSON", "DAN JACOBS" and "PAUL MASON". Returning to the list routine gives the following:

```
RECORD #    KEY FIELD     DATA FIELD
==============================================================
     2      JOE           JONES
     3      DAN           JACOBS
     4      DON           WILSON
     5      JOE           JOHNSON
     6      JOE           WILLIAMS
     7      PAUL          MASON

END OF FILE...PRESS <ENTER>
```

After this, you may also want to go back to look at the hash index. Record 1 shows the following:

```
HASH INDEX PHYSICAL RECORD 1
  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,
  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,
  ...  Ø,  DEL  Ø,  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,
  ...  Ø,    4 164,  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,
  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,
  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,
  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,
  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,
  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,
  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,
  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,
  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,
  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,
  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,  ...  Ø,    7 198,
  ...  Ø,     <N> NEXT, <P> PREV, OR <ENTER> TO EXIT...
```

You can observe that the keys "DON" and "PAUL" have hashed to record 1 by noticing record numbers 4 and 7 in the pointer fields. You also notice a "DEL". That's where the pointer for "BILL" was stored before you deleted the record containing "BILL ADAMS".

Ending KEYACCES/DEM

Menu selection 5 closes the two files used by KEYACCES/DEM, and it returns you to BASIC's READY mode. After the program is terminated you may wish to kill the files it used. Your commands are:

```
KILL "KADEM/MAS"
KILL "KADEM/IND"
```

Error Conditions with KEYACCES/DEM

Error codes may be of two types with KEYACCES/DEM. If the error is in the master file, you'll get the standard error codes given by the random files handler. The error code, file number and physical record number are displayed, and you have an opportunity to retry by pressing <ENTER>, or to abort by pressing any other key.

The error codes given by KEYACCES are shown with a minus sign in front of the number, which you can look up in your disk operating system owner's manual. You are given the ability to retry with these errors also.

Figure 14.5 — *KEYACCES/DEM*

```
Ø '"KEYACCES/DEM"
1 CLEAR1ØØØ              'CLEAR SOME STRING STORAGE SPACE
2 DEFINTA-Z              'DEFINE ALL VARIABLES AS INTEGER
3 DEFUSR8=-4Ø96          'USE USR8 FOR KEYACCES AT FØØØ
4 DIMKA(15):KY$="":J=Ø   'INITIALIZE KEYACCES VARIABLES
5 HS=3  'FOR DEMO LIMIT HASH INDEX SIZE TO 3
6 MS=32 'FOR DEMO LIMIT MASTERFILE SIZE TO 32 PHYSICAL RECORDS
7 M2=Ø  'PROGRAM MAKES M2=1 IF TRSDOS 2.Ø ON MODEL 2

1Ø CLS:PRINT"KEYACCES DEMONSTRATION PROGRAM":PRINT
11 PRINT"THIS PROGRAM SETS UP A MASTERFILE AND INDEX"
12 PRINT"EACH MASTERFILE RECORD IS 32 BYTES LONG AND CONTAINS 2
FIELDS:"
13 PRINT"FIELD 1 IS AN 8-BYTE KEY.  FIELD 2 IS A 24-BYTE DATA FI
ELD."
14 PRINT"FOR THE DEMO, YOU MAY HAVE UP TO";HS*85-1;"LOGICAL RECO
RDS."
15 PRINT
16 PRINT"KEYACCES MUST BE IN MEMORY AT -4Ø96."
17 PRINT"IF IT IS NOT, PRESS <BREAK> AND RUN KALOADER/BAS"
18 PRINT"OTHERWISE PRESS <ENTER>..."
19 GOSUB4Ø5ØØ:IFA$<>CHR$(13)THEN19

2Ø CLS:PRINT"ARE YOU USING:"
21 PRINT"<A> TRSDOS 1.3 ON A MODEL 3"
22 PRINT"<B> TRSDOS 2.3 ON A MODEL 1"
23 PRINT"<C> TRSDOS 2.Ø ON A MODEL 2"
24 PRINT"<D> SOMETHING ELSE?"
25 GOSUB4Ø5ØØ:A%=INSTR("ABCD",A$):IFA%=ØTHEN25
26 PRINT:ONA%GOTO31,32,33,34
31 D1=26182:D2=26542:CLS:PRINT"MODEL 3, TRSDOS 1.3":GOTO4Ø
32 D1=263Ø3:D2=26593:CLS:PRINT"MODEL 1, TRSDOS 2.3":GOTO4Ø
33 D1=27715:D2=28549:CLS:M2=1:PRINT"MODEL 2, TRSDOS 2.Ø":GOTO4Ø
34 INPUT"FILE 1 DATA CONTROL BLOCK ADDRESS";D1
35 INPUT"FILE 2 DATA CONTROL BLOCK ADDRESS";D2:CLS
4Ø PRINT"FILE 1 DCB IS";D1;TAB(32);"FILE 2 DCB IS";D2

5Ø PRINT
51 PRINT"TWO FILES ARE USED FOR THE DEMONSTRATION:":PRINT
52 PRINT"FILE 1 IS MASTER FILE, KADEM/MAS:Ø, LENGTH =";MS;"RECOR
DS"
```

```
53 PRINT"FILE 2 IS INDEX  FILE, KADEM/IND:Ø, LENGTH =";HS;"RECOR
DS"
54 PRINT
55 PRINT"PRESS <ENTER> TO OPEN THEM, OTHERWISE PRESS <BREAK>..."
56 GOSUB4Ø5ØØ:IFA$<>CHR$(13)THEN56
57 GOSUB58ØØØ               'DIMENSION ARRAYS & GET REC LENGTHS
58 FS$="KADEM/MAS:Ø":PRINT"OPENING ";FS$:PF=1:GOSUB5825Ø
59 FS$="KADEM/IND:Ø":PRINT"OPENING ";FS$:PF=2:GOSUB5825Ø

6Ø IFLOF(1)=ØORLOF(2)=ØTHEN7Ø
61 PRINT"DO YOU WANT TO INITIALIZE THEM TO AN EMPTY CONDITION?
(Y/N) ";
62 GOSUB4Ø5ØØ:A%=INSTR("YN",A$):IFA%=ØTHEN62ELSEPRINTA$
63 IFA$="N"THEN8Ø

7Ø PRINT"BUILDING THE MASTER FILE..."
71 FIELD1,128ASA$,128ASB$:LSETA$=STRING$(128,Ø):LSETB$=A$
72 PF=1:FORX=1TOMS:PR(1)=X:GOSUB583ØØ:NEXT
73 GOSUB584Ø8               'PUT STATS IN REC 1 OF MASTER FILE
75 PRINT"BUILDING THE INDEX FILE..."
76 FIELD2,128ASA$,128ASB$:LSETA$=STRING$(128,Ø):LSETB$=A$
77 PF=2:FORX=1TOHS:PR(2)=X:GOSUB583ØØ:NEXT
8Ø PF=1:GOSUB584Ø5         'GET STATS FROM REC 1 OF MASTER FILE

9Ø 'LOAD KEYACCES CONTROL ARRAY CONSTANTS......
91 KA(2)=VARPTR(KY$)       'VARPTR OF SEARCH STRING
92 KA(3)=D1                'MASTER FILE DCB ADDRESS (FILE 1)
93 KA(4)=Ø                 'MASTER FILE OFFSET
94 KA(5)=8                 'MASTER FILE BLOCKING FACTOR
95 KA(6)=32                'MASTER FILE LOGICAL RECORD LENGTH
96 KA(7)=Ø                 'KEY OFFSET
97 KA(8)=D2                'HASH INDEX DCB ADDRESS (FILE 2)
98 KA(9)=Ø                 'HASH INDEX OFFSET
99 KA(1Ø)=HS               'HASH INDEX SIZE (3 PHYSICAL RECORDS)

1ØØ CLS:PRINT"KEYACCES DEMONSTRATION PROGRAM"
1Ø1 PRINTSTRING$(63,"="):PRINT
11Ø PRINT"<1> DISPLAY CONTROL INFORMATION AND STATISTICS"
12Ø PRINT"<2> SEARCH, ADD, DELETE"
13Ø PRINT"<3> DISPLAY HASH INDEX"
14Ø PRINT"<4> LIST MASTER FILE RECORDS"
15Ø PRINT"<5> CLOSE FILES AND END"
16Ø PRINT:PRINTSTRING$(63,"="):PRINT
19Ø PRINT"PRESS THE NUMBER OF YOUR SELECTION..."
191 GOSUB4Ø5ØØ:A%=INSTR("12345",A$):IFA%=ØTHEN191ELSEONA%GOTO1ØØ
Ø,2ØØØ,3ØØØ,4ØØØ,2ØØ

2ØØ CLS:PRINT"CLOSING...":FORPF=1TO2:GOSUB5829Ø:NEXT
21Ø END

1ØØØ CLS:PRINT"CONTROL ARRAY CONTENTS":PRINTSTRING$(63,"=")
1Ø1Ø FOR X=ØTO7
```

```
1Ø2Ø PRINT"KA(";X;")=";KA(X);TAB(32);"KA(";X+8;")=";KA(X+8)
1Ø3Ø NEXT
1Ø4Ø PRINT:PRINT"MASTER FILE CONTAINS";LØ(1,3);"ACTIVE LOGICAL R
ECORDS"
1Ø41 PRINT"THIS DEMO LIMITS YOU TO";HS*85-1;"ACTIVE LOGICAL RECO
RDS"
1Ø5Ø PRINTSTRING$(63,"="):PRINT"PRESS <ENTER>...":GOSUB4Ø5ØØ:GOT
O1ØØ

2ØØØ 'SEARCH, ADD, DELETE, ---------------------------------
2Ø1Ø CLS:CD$="":LC$="  "
2Ø2Ø PRINT"SF, SN          SEARCH - FIRST, NEXT
2Ø21 PRINT"AU, AN, AF, AC  ADD    - UNIQUE, NON-UNIQUE, FIFO, CU
RRENT"
2Ø23 PRINT"DC              DELETE - CURRENT"
2Ø25 PRINT"----------- UP-ARROW RETURNS TO MENU ---------------"
2Ø3Ø LINEINPUT"COMMAND: ";A$:IFA$=CHR$(91)THEN1ØØELSEIFA$=""THEN
PRINTCD$:A$=CD$
2Ø31 A%=INSTR("SF,SN,AU,AN,AF,AC,DC",A$):IF(A%=Ø)OR(LEN(A$)<>2)T
HENPRINT"INVALID COMMAND!":GOTO2Ø2ØELSECD$=A$:ON(A%-1)/3+1GOSUB2
1ØØ,211Ø,212Ø,213Ø,214Ø,215Ø,216Ø
2Ø4Ø GOTO2Ø3Ø
2Ø99 '------------------------------------------------------

21ØØ PRINT"SEARCH FIRST";TAB(25):LINEINPUT"KEY:  ";A$:IFA$=CHR$(
91)THENRETURNELSEKY$=A$:LC$="  "
21Ø1 KA(Ø)=1:GOSUB55ØØØ:IFJ<ØTHENRETURN
21Ø2 IFJ=ØTHENPRINTTAB(25)"** NOT FOUND **":LC$="*N":RETURN
21Ø3 PF=1:PR(1)=KA(14)+M2:PP(1)=PR(1):LS=KA(15):LR(1)=J:GOSUB58Ø
1Ø
21Ø4 PRINT"FOUND IN RECORD";LR(1);TAB(25)
21Ø5 PRINT"KEY:  ";FH$(1)
21Ø6 PRINTTAB(25)"DATA: ";FH$(2)
21Ø7 LC$="*F":RETURN
21Ø9 '------------------------------------------------------

211Ø PRINT"SEARCH NEXT";TAB(25):IFLC$<>"*F"THENPRINT"* NOT PERMI
TTED *":RETURNELSEPRINTTAB(25);"KEY:  ";KY$
2111 KA(Ø)=2:GOSUB55ØØØ:IFJ<ØTHENRETURN
2112 GOTO21Ø2
2119 '------------------------------------------------------

212Ø PRINT"ADD UNIQUE";:PRINTTAB(25):LINEINPUT"KEY:  ";A$:IFA$=C
HR$(91)THENRETURNELSEKY$=A$:LC$="  "
2121 PRINTTAB(25)"DATA: ";
2122 LINEINPUTA$:IFA$=CHR$(91)THEN212ØELSEDA$=A$
2123 KA(Ø)=1:GOSUB55ØØØ:IFJ<ØTHENRETURNELSEIFJ>ØTHENPRINTTAB(25)
"** DUPLICATE EXISTS - NOT ADDED **":RETURN
2124 GOSUB551ØØ:IFA$=CHR$(91)THENRETURN
2125 KA(Ø)=4:KA(1)=LR(1):GOSUB55ØØØ
2126 RETURN
2129 '------------------------------------------------------
```

```
213Ø PRINT"ADD NON-UNIQUE";TAB(25)
2131 LINEINPUT"KEY:  ";A$:IFA$=CHR$(91)THENRETURNELSEKY$=A$:LC$=
"  "
2132 PRINTTAB(25)"DATA: ";
2133 LINEINPUTA$:IFA$=CHR$(91)THEN213ØELSEDA$=A$
2134 KA(13)=Ø   'FORCE DELETED SPACE TO BE RECOVERED IN INDEX
2135 GOSUB551ØØ:IFA$=CHR$(91)THENRETURN
2136 KA(Ø)=3:KA(1)=LR(1):GOSUB55ØØØ:RETURN
2139 '---------------------------------------------------

214Ø PRINT"ADD FIFO";TAB(25)
2141 LINEINPUT"KEY:  ";A$:IFA$=CHR$(91)THENRETURNELSEKY$=A$:LC$=
"  "
2142 PRINTTAB(25)"DATA: ";
2143 LINEINPUTA$:IFA$=CHR$(91)THEN214ØELSEDA$=A$
2144 KA(13)=-1 'FORCE NEW ADD AT END OF CHAIN OF DUPLICATES
2145 GOTO2135
2149 '---------------------------------------------------

215Ø PRINT"ADD CURRENT";:PRINTTAB(25):IFLC$<>"*N"THENPRINT"* NOT
 PERMITTED *":RETURN
2151 PRINTTAB(25);"DATA: ";:LINEINPUTA$:IFA$=CHR$(91)THENRETURNE
LSEDA$=A$
2152 GOSUB551ØØ:IFA$=CHR$(91)THENRETURN
2153 KA(Ø)=4:KA(1)=LR(1):GOSUB55ØØØ
2154 LC$="  ":RETURN
2159 '---------------------------------------------------

216Ø PRINT"DELETE CURRENT";TAB(25):IFLC$<>"*F"THENPRINT"* NOT PE
RMITTED *":RETURN
2161 PRINTTAB(25)"DELETING FROM MASTER FILE":PF=1:GOSUB5844Ø
2162 PRINTTAB(25)"DELETING FROM INDEX":KA(Ø)=5:GOSUB55ØØØ
2163 LC$="  ":RETURN
2169 '---------------------------------------------------

3ØØØ 'SHOW HASH INDEX ------------------------------------
3Ø1Ø CLS:IFLC$=""THEN3Ø2ØELSEPRINT"LAST COMMAND: HASH RECORD";AS
C(MKI$(KA(12)))+1;", SUBRECORD";ASC(MID$(MKI$(KA(12)),2))+1;
3Ø11 PRINT", HASH CODE";ASC(MKI$(KA(11)))
3Ø12 PF=2:PR(2)=ASC(MKI$(KA(12)))+1:GOSUB5822Ø
3Ø15 GOSUB31ØØ:GOTO3Ø3Ø
3Ø2Ø PR(2)=1:GOTO3Ø5Ø
3Ø3Ø IFA$=CHR$(13)THEN1ØØELSEIFA$="N"THENPR(2)=PR(2)+1ELSEPR(2)=
PR(2)-1
3Ø4Ø IFPR(2)<1THENPR(2)=HS
3Ø41 IFPR(2)>HSTHENPR(2)=1
3Ø5Ø CLS:PRINT"HASH INDEX PHYSICAL RECORD";PR(2)
3Ø51 PF=2:GOSUB5822Ø:GOSUB31ØØ:GOTO3Ø3Ø

31ØØ 'DISPLAY CURRENT HASH INDEX RECORD
311Ø FORLS=ØTO84          'FOR EACH SUBRECORD
312Ø PF=2:GOSUB58Ø2Ø      'GO FIELD IT
```

```
313Ø IFCVI(FH$(3))<ØTHENPRINT"  DEL   Ø,";:GOTO315Ø
3131 IFCVI(FH$(3))=ØTHENPRINT"  ...   Ø,";:GOTO315Ø
314Ø PRINTUSING"#####";CVI(FH$(3));  'DISPLAY POINTER
3141 PRINTUSING" ###";ASC(FH$(4));   'DISPLAY 1 BYTE HASH
3142 PRINT",";
315Ø IF(LS+1)/6=INT((LS+1)/6)THENPRINT
316Ø NEXT
317Ø PRINT"      <N> NEXT, <P> PREV, OR <ENTER> TO EXIT...";
318Ø GOSUB4Ø5ØØ:IFINSTR("NP"+CHR$(13),A$)=ØTHEN318ØELSERETURN

4ØØØ 'LIST RECORDS FROM MASTER FILE ------------------------
4Ø1Ø PF=1:LR(1)=2
4Ø2Ø CLS:PRINT"RECORD#    KEY FIELD     DATA FIELD":PRINTSTRING$
(63,"=")
4Ø3Ø IFLR(1)>LØ(1,1)-1THENPRINT:PRINT"END OF FILE...PRESS <ENTER
>":GOSUB4Ø5ØØ:GOTO1ØØ
4Ø4Ø GOSUB582ØØ
4Ø5Ø PRINTUSING"######     ";LR(1);
4Ø6Ø IFASC(FH$(2))=255THENPRINT"DELETED":GOTO4Ø8Ø
4Ø7Ø PRINTFH$(1);"      ";FH$(2)
4Ø8Ø LR(1)=LR(1)+1
4Ø9Ø IFLR(1)/1Ø=INT(LR(1)/1Ø)THENPRINT"PRESS <ENTER> TO CONTINUE
...";:GOSUB4Ø5ØØ:IFA$<>CHR$(13)THEN1ØØELSE4Ø2Ø
41ØØ GOTO4Ø3Ø

4Ø5ØØ A$=INKEY$:IFA$<>""THENRETURNELSE4Ø5ØØ

55ØØØ J=USR8(VARPTR(KA(Ø)))      'CALL KEYACCES USR SUBROUTINE
55ØØ5 IFJ>=ØTHENRETURN           'RETURN IF NO ERROR
55Ø1Ø PRINTTAB(25)"** KEYACCES ERROR CODE ";J;" **"
55Ø2Ø PRINTTAB(25)"PRESS <R> TO RETRY, OTHERWISE <ENTER>";
55Ø3Ø GOSUB4Ø5ØØ:IFA$="R"THEN55ØØØELSEIFA$<>CHR$(13)THEN55Ø3Ø
55Ø4Ø RETURN

551ØØ 'ADD A NEW MASTERFILE LOGICAL RECORD
5511Ø IFLØ(1,3)>=HS*85-1THENPRINTTAB(25)"** LIMIT REACHED - NOT
ADDED **":A$=CHR$(91):RETURN
5512Ø PF=1                       'SPECIFY MASTER FILE (FILE 1)
55121 PP(1)=Ø 'BUFFER CONTENTS UNCERTAIN SO FORCE "HARD GET"
55122 GOSUB5842Ø                 'GET AND FIELD RECORD TO ADD
5513Ø LSETFH$(1)=KY$             'PUT KEY IN NEW RECORD BUFFER
5514Ø LSETFH$(2)=DA$             'PUT DATA IN NEW RECORD BUFFER
5515Ø PRINT"ADDING AS RECORD";LR(1)  'TELL THE OPERATOR
5516Ø GOSUB5843Ø                 'CALL 5843Ø TO ADD NEW RECORD
5517Ø A$="":RETURN               'RETURN

58ØØØ A%=2:DIMPR(A%),PP(A%),LR(A%),LL(A%),FD$(A%)
58ØØ1 LL(1)=32:LL(2)=3
58ØØ2 RETURN
58Ø1Ø FIELDPF,LS*LL(PF)ASFD$(PF),8ASFH$(1),24ASFH$(2)
58Ø11 RETURN
58Ø2Ø FIELDPF,LS*LL(PF)ASFD$(PF),2ASFH$(3),1ASFH$(4)
```

```
58Ø21 RETURN
582ØØ LB=INT(256/LL(PF)):LS=LR(PF)-LB*INT((LR(PF)-1)/LB)-1:PR(PF
)=INT((LR(PF)-1)/LB)+1
582Ø5 ONPFGOSUB58Ø1Ø,58Ø2Ø
5821Ø IFPR(PF)=PP(PF)THENRETURN
5822Ø PP(PF)=PR(PF):ONERRORGOTO589ØØ:GETPF,PR(PF):ONERRORGOTOØ:R
ETURN
5825Ø GOSUB5829Ø:ONERRORGOTO5891Ø:OPEN"R",PF,FS$:ONERRORGOTOØ:PP
(PF)=Ø:RETURN
5829Ø ONERRORGOTO5893Ø:CLOSEPF:ONERRORGOTOØ:RETURN
583ØØ ONERRORGOTO5892Ø:PUTPF,PR(PF):ONERRORGOTOØ:RETURN
584ØØ PR(PF)=1:LØ(PF,4)=KV%:LØ(PF,5)=KS%:GOSUB5841Ø:GOSUB5821Ø:F
ORA%=1TO5:LSETFØ$(A%)=MKI$(LØ(PF,A%)):NEXT:GOSUB583ØØ:RETURN
584Ø5 PR(PF)=1:GOSUB5841Ø:GOSUB5821Ø:FORA%=1TO5:LØ(PF,A%)=CVI(FØ
$(A%)):NEXT:RETURN
584Ø8 LØ(PF,1)=9/LL(PF)+2:LØ(PF,5)=KS%:FORA%=2TO4:LØ(PF,A%)=Ø:NE
XT:GOTO584ØØ
5841Ø FIELDPF,2ASFØ$(1),2ASFØ$(2),2ASFØ$(3),2ASFØ$(4),2ASFØ$(5):
RETURN
5842Ø IFLØ(PF,2)=ØTHEN58421ELSELR(PF)=LØ(PF,2):GOSUB582ØØ:FIELDP
F,LS*LL(PF)+2ASFD$(PF):LØ(PF,Ø)=CVI(RIGHT$(FD$(PF),2)):RETURN
58421 LR(PF)=LØ(PF,1):LØ(PF,Ø)=-1:GOSUB582ØØ:RETURN
5843Ø GOSUB583ØØ:LØ(PF,3)=LØ(PF,3)+1:IFLØ(PF,Ø)>-1THENLØ(PF,2)=L
Ø(PF,Ø)ELSELØ(PF,1)=LØ(PF,1)+1
58431 GOSUB584ØØ:RETURN
5844Ø FIELDPF,LS*LL(PF)ASFD$(PF),LL(PF)+(LL(PF)=256)ASFD$(PF):LS
ETFD$(PF)=MKI$(LØ(PF,2))+STRING$(LL(PF)-2+(LL(PF)=256),255)
58441 GOSUB583ØØ:LØ(PF,2)=LR(PF):LØ(PF,3)=LØ(PF,3)-1:GOSUB584ØØ:
RETURN
589ØØ A$="DISK READ ERROR":GOTO5899Ø
5891Ø A$="CAN'T OPEN DISK FILE":GOTO5899Ø
5892Ø A$="DISK WRITE ERROR":GOTO5899Ø
5893Ø A$="CAN'T CLOSE DISK FILE":GOTO5899Ø
5899Ø A1$="":A%=VARPTR(A1$):POKEA%,64:POKEA%+1,192:POKEA%+2,63:A
2$=A1$:A%=PEEK(16416):A1%=PEEK(16417)
58991 PRINT@96Ø,CHR$(143);A$;TAB(22)"(E=";MID$(STR$(ERR/2),2);"
F=";MID$(STR$(PF),2);" R=";MID$(STR$(PR(PF)),2);")";TAB(41);"PRE
SS <ENTER> TO RETRY!";CHR$(143);
58992 A$=INKEY$:IFA$=""THEN58992
58993 PRINT@96Ø,CHR$(31);
58994 LSETA1$=A2$:POKE16416,A%:POKE16417,A1%
58995 IFA$<>CHR$(13)THENRESUME1ØØ
58996 RESUME
```

Model II Modifications for KEYACCES

The KEYACCES routine performs well on the Model II with TRSDOS 2.0 or 2.0a. Several modifications are required. It is possible to save a dozen bytes or more because the DIRRD and DIRWR supervisor calls are used. The changes listed below don't realize the savings because they just "patch" the Model I and III version by replacing unneeded operations with "no-operations", or NOPs. This was done to simplify our discussions by keeping the line numbers and byte positions the same.

To the KEYACCES assembly listing make the following changes:

```
As shown:   CD7FØA  ØØ31Ø    CALL  ØA7FH       ;PUT USR ARGUMENT IN HL

Change to:  CD5D44  ØØ31Ø    CALL  Ø445DH      ;PUT USR ARGUMENT IN HL

As shown:   FD6EØA  Ø195Ø    LD    L,(IY+1Ø)   ;
            FD66ØB  Ø196Ø    LD    H,(IY+11)   ;HL HAS NRN FOR DCB
            2B      Ø197Ø    DEC   HL          ;HL = CURRENT REC #
Change to:  FD6EØ7  Ø195Ø    LD    L,(IY+7)    ;
            FD66Ø8  Ø196Ø    LD    H,(IY+8)    ;HL = CURRENT REC #
            ØØ      Ø197Ø    NOP               ;NO DECREMENT NECESSARY

As shown:   112ØØØ  Ø2Ø3Ø    LD    DE,2ØH      ;ADD 32 TO DCB
Change to:  114ØØØ  Ø2Ø3Ø    LD    DE,4ØH      ;ADD 64 TO DCB

As shown:   CD4244  Ø2Ø9Ø    CALL  Ø4442H      ;POSITION
            2ØØ3    Ø21ØØ    JR    NZ,J5C      ;JUMP IF DISK ERROR
            CD3644  Ø211Ø    CALL  Ø4436H      ;READ
Change to:  3E23    Ø2Ø9Ø    LD    A,35        ;LOAD DIRRD CODE
            CF      Ø21ØØ    RST   8           ;DO SUPERVISOR CALL
            ØØ      Ø211Ø    NOP               ;
            ØØ      Ø2111    NOP               ;
            ØØ      Ø2112    NOP               ;
            ØØ      Ø2113    NOP               ;
            ØØ      Ø2114    NOP               ;

As shown:   C39AØA  Ø219Ø    JP    ØA9AH       ;RET ERROR CODE TO BASIC
Change to:  C37A44  Ø219Ø    JP    Ø447AH      ;RET ERROR CODE TO BASIC

As shown:   C39AØA  Ø28ØØ    JP    ØA9AH       ;RET NOT-FOUND TO BASIC
Change to:  C37A44  Ø28ØØ    JP    Ø447AH      ;RET NOT-FOUND TO BASIC

As shown:   FD6EØA  Ø351Ø    LD    L,(IY+1Ø)   ;
            FD66ØB  Ø352Ø    LD    H,(IY+11)   ;HL=PHYSICAL REC#
Change to:  FD6EØ7  Ø351Ø    LD    L,(IY+7)    ;
            FD66Ø8  Ø352Ø    LD    H,(IY+8)    ;HL=PHYSICAL REC#

As shown:   C39AØA  Ø357Ø    JP    ØA9AH       ;RETURN REC# TO BASIC
Change to:  C37A44  Ø357Ø    JP    Ø447AH      ;RETURN REC# TO BASIC

As shown:   FD6EØA  Ø385Ø    LD    L,(IY+1Ø)   ;
            FD66ØB  Ø386Ø    LD    H,(IY+11)   ;HL HAS NEXT REC #
            2B      Ø387Ø    DEC   HL          ;PUT IT BACK TO CURRENT
            FD75ØA  Ø388Ø    LD    (IY+1Ø),L   ;
            FD74ØB  Ø389Ø    LD    (IY+11),H   ;NRN IS ADJUSTED
Change to:  FD4EØ7  Ø385Ø    LD    C,(IY+7)    ;
            FD46Ø8  Ø386Ø    LD    B,(IY+8)    ;BC HAS CUR REC #
            ØØ      Ø387Ø    NOP               ;
            ØØ      Ø388Ø    NOP               ;
            ØØ      Ø3881    NOP               ;
            ØØ      Ø3882    NOP               ;
            ØØ      Ø389Ø    NOP               ;
```

```
             00      03891    NOP              ;
             00      03892    NOP              ;

As shown:    CD3944  03920    CALL  4439H      ;WRITE THE RECORD
Change to:   3E2C    03920    LD    A,44       ;LOAD DIRWR CODE
             CF      03921    RST   8          ;DO SUPERVISOR CALL

As shown:    C39A0A  04000    JP    0A9AH      ;RET ERROR CODE TO BASIC
Change to:   C37A44  04000    JP    0447AH     ;RET ERROR CODE TO BASIC
```

Make the following changes to the magic array format listing for KEYACCES:

```
As shown:  32717  -6902  -7715
Change to: 24013  -6844  -7715

As shown:   -758   2670  26365  11019  -4681
Change to:  -758   1902  26365      8  -4681

As shown:   4577     32  -3815
Change to:  4577     64  -3815

As shown:  -6659 -12847  17474    800  14029 -11964   2600
Change to: -6659  16081 -12509      0      0 -12032   2600

As shown:   9071 -25917 -12022  17355
Change to:  9071  31427 -11964  17355

As shown:    884 -25917   6154 -11944
Change to:   884  31427   6212 -11944

As shown:  28413   -758   2918  30173
Change to: 28413   -761   2150  30173

As shown:    870 -25917  -7926   1086
Change to:   870  31427  -7868   1086

As shown:   -559   2670  26365  11019  30205   -758   2932  -6691
Change to:  -559   1870  18173      8      0      0      0  -6691

As shown:  -6659  14797   -700  -8735
Change to: -6659  11326   -561  -8735

As shown:   9071 -25917     10
Change to:  9071  31427     68
```

The following changes are required for the poke format listing of KEYACCES:

```
As shown:  205 127  10 229
Change to: 205  93  68 229

As shown:  110  10 253 102  11  43 183
Change to: 110   7 253 102   8   0 183

As shown:   17  32   0
Change to:  17  64   0
```

```
As shown:  2Ø9 2Ø5  66  68  32   3 2Ø5  54  68 2Ø9
Change to: 2Ø9  62  35 2Ø7   Ø   Ø   Ø   Ø   Ø 2Ø9

As shown:  195 154  1Ø 2Ø9
Change to: 195 122  68 2Ø9

As shown:  195 154  1Ø  24
Change to: 195 122  68  24

As shown:  11Ø  1Ø 253 1Ø2  11 221
Change to: 11Ø   7 253 1Ø2   8 221

As shown:  195 154  1Ø 225
Change to: 195 122  68 225

As shown:  253 11Ø  1Ø 253 1Ø2  11  43 253 117  1Ø 253 116  11 221
Change to: 253  78   7 253  7Ø   8   Ø   Ø   Ø   Ø   Ø   Ø   Ø 221

As shown:  229 2Ø5  57  68 253
Change to: 229  62  44 2Ø7 253

As shown:  195 154  1Ø
Change to: 195 122  68
```

The changes to poke format must also be made to the KALOADER/BAS program. You must also delete lines 40 through 42 and 110 through 122 from KALOADER/BAS.

The modifications shown for the Model II version of KEYACCES cause one minor incompatibility with the Model I and III version. In the Model II version shown, upon return from a sucessful search, control array element 14 contains the physical record number in the master file minus 1. Thus, if the key is found in record 1, element 14 will contain 0. You can adjust for this in your BASIC program by adding 1 to element 14 after a search (as is done in KEYACCES/DEM).

Alternatively, you can ignore element 14 and use the LOC function to find the current record number in the master file after a search. The other way to correct this inconsistency is to insert the following into the KEYACCES assembly listing:

```
Ø3521     INC   HL          ;ADJUST HL SO REC# IS CORRECT
```

This difference is caused by the fact that under the Model I and III operating systems, the 10th and 11th bytes of the Data Control Block contain the next record number. On the Model II, the 7th and 8th bytes are used, but they contain the current record number.

To run KEYACCES/DEM on the Model II, you will need to change KEYACCES as noted. You'll probably do this by making a change to KALOADER/BAS. See Appendix 3 for instructions on installing the PEEK and POKE capability that is required for KALOADER/BAS.

You'll also need to make the Model II modifications noted for the random file handler. These changes will affect lines 58220, 58221, 58990, 58991, 58993, and 58994. Since KEYACCES/DEM builds the demo files in advance, the changes in 58220 and 58221 (which prevent an error condition when you attempt to GET beyond the end of file) are optional.

Disk File Searching

Disk file searching can be viewed in two ways. One type of searching was covered in Chapters 10 through 14, where we looked at the Memory Index Handler, TREESAM and KEYACCES as three techniques for keyed accessing of random files. These methods are designed mainly for fast primary-key accessing. To enhance the speed, the file layout and program logic were designed so a more-or-less direct route could be followed from search key to the desired record. At the expense of memory or disk overhead you are able to search quite quickly.

We'll talk about another way to view random file searching in this chapter. "Record-by-record" best describes this other approach to searching. Given a search key, you start at the first logical record in the file, get it, do a comparison, and move on to the next if the keys don't match. You continue with each record of the file until you reach the end, or a record containing a key that meets the search criteria is found.

It is easy to program a record-by-record search in BASIC. You can set up a FOR-NEXT loop that calls subroutines to get and field each record in the file. When a record has been fielded into the buffer, you use simple IF-THEN statements. Suppose you have a variable length record file, and you want to make a printout of all records whose 7th through 17th bytes match a search key contained in KY$. Once the file's been opened you can use this logical organization:

```
FIELD BF%, 6 AS A$, 1Ø AS B$  'FIELD BUFFER, BF%
FOR X = 1 TO LOF(1)           'FOR EACH RECORD IN THE FILE
GET 1, X                      'GET IT
IF KY$ = B$ THEN GOSUB ....   'IF FOUND, PRINT OR PROCESS IT
NEXT                          'REPEAT FOR NEXT IN FILE
```

Or, suppose you're using the labeled files handler. You can use the following to print or process all the records whose B$ field matches KY$:

```
FOR X = 2 TO LØ(PF,1)-1       'FOR EACH RECORD IN THE FILE
GOSUB 582ØØ                   'GET AND FIELD IT
IF KY$ = B$ THEN GOSUB ....   'IF FOUND, PRINT OR PROCESS IT
NEXT                          'REPEAT
```

Of course, you don't have to use a FOR-NEXT loop. You can provide logic to get the records one-by-one. If the EOF function tells your program that the end of file has been reached, you end the search.

The beauty of these methods is that they are quite flexible and easy to program. The problem with them is that record-by-record searches can be quite slow, especially if you have a large file and the key you are seeking is somewhere near the end. When you have blocked more than one logical record per physical record and the program has to field each subrecord, the process is even slower.

One way to improve upon record-by-record searches in which you have more than one logical record per physical record is to use the INSTR function. Instead of fielding each subrecord, use INSTR on the contents of the entire buffer. If a potential match is found, have your program determine whether or not the key is in the correct field position of the buffer. If it's not, continue your INSTR search of the buffer, and of subsequent records. If it is, call your subroutine to field the buffer and process the record. You can get an idea of how this approach works if you examine how INSTR is used in the Memory Index Handler. The INSTR method can be relatively fast, but it still might not be as speedy as you would like. One big drawback is that it is limited to searches for exact matches on your search key.

DFSEARCH

DFSEARCH is one answer to the problems of searching random files. It is a 254-byte relocatable USR subroutine that can be called from your BASIC program to put machine language speed into your searches. Being a relocatable subroutine, you can put it anywhere in protected memory by poking it in, or by using any of the methods described in *BASIC Faster and Better & Other Mysteries.*

DFSEARCH lets you specify a string to be used as a search key. You also tell it information about your disk file, such as the record length it uses, the blocking factor, where you want to begin your search, and which portion of each record to examine. DFSEARCH doesn't limit you to exact matches. You can, if you wish, request "less than", "less than or equal", "greater than", "greater than or equal" or "not equal" searches.

Search times will depend on the length of your file and the speed of your disk drives, but as an example, a disk file filling an entire TRS-80 Model I 35-track diskette can be searched from beginning to end in about 23 seconds. The alternatives in BASIC can take several minutes to search through the same file.

DFSEARCH is designed for use with random files. It may be used in conjunction with the random disk file handler subroutines, or you may wish to use other methods. The logical record length may be from 1 to 256 bytes. The blocking factor (logical records per 256 byte physical record) may be from 1 to 255. You can also use DFSEARCH with variable-length record files.

DFSEARCH is especially valuable in programs where you are printing selected records from a disk file according to requests by the operator. You may, for example, wish to print all records from an inventory file having a certain supplier code. You may be searching an accounts payable file for all invoices that will come due prior to a selected date. Perhaps you want to find all deleted records, or alternatively, all active records. Maybe you want to load data from certain records to a memory array for sorting. DFSEARCH can do all these things quite quickly.

Unlike KEYACCES, TREESAM, and other keyed indexing systems, DFSEARCH requires no special "overhead" space in the disk file, and no special procedures when the record is added, deleted or changed. These methods are desirable for primary access to disk files. DFSEARCH is often the method of choice for unlimited secondary ways to search and access random disk files.

Loading the DFSEARCH USR Routine

Like KEYACCES, DFSEARCH is a machine language USR subroutine, but you don't need to understand Z-80 assembly language programming to use it. You can load DFSEARCH into any 254 contiguous bytes of protected memory. That means that if you put it right at the top of memory in a 48K TRS-80, you will need to specify a memory size of 65282 or less when entering BASIC. You can avoid the need to reserve memory it by loading it into a magic array or a string. If you wish, you can store the whole thing in a random disk file physical record, GET it into a buffer, and execute it there. *BASIC Faster and Better & Other Mysteries* gives you more information about how to implement these techniques.

DSLOADER/BAS

DSLOADER/BAS is a BASIC program that shows one way to load DFSEARCH/BAS into protected memory. It takes the 254 bytes and POKEs them from **FE00** to **FEFD**. (In decimal that's 65024 through 65277. In integer notation it's –512 through –257.) To use it as listed, you'll need to specify a memory size of 65024 or less. Of course you can change the address. Just modify line 20 and make appropriate changes to the memory size you specify upon entering BASIC.

The operation of DSLOADER/BAS is like KALOADER/BAS. It stops you if you haven't reserved enough memory. After poking the numbers, it requests information that is needed to make one final required POKE:

```
DOES YOUR DISK OPERATING SYSTEM USE:

A> A 5Ø-BYTE DCB, LIKE TRSDOS 1.3 FOR THE MODEL 3
B> A 32-BYTE DCB, LIKE TRSDOS 2.3 FOR THE MODEL 1
```

If you are unable to answer this question, you'll need to check Appendix 1. It tells you how to find the data control block addresses and the length of the DCB that is used for your DOS. Once you know what applies, you can enter <A> or <B> to set up DFSEARCH properly. Your response tells DSLOADER/BAS whether to poke 50 or 32 into the 121st byte of the routine.

After DSLOADER/BAS has finished its job, you can load or run programs that use the DFSEARCH routine. DSLOADER tells you the number that you can use in your DEFUSR statement. Alternatively, you can go back to DOS READY and use the DUMP library command to put DFSEARCH into a disk file. That way, whenever you want to use it, you can "LOAD DFSEARCH" before entering BASIC, and it will be there in memory when you need it.

Figure 15.1 — *DSLOADER/BAS*

```
Ø 'DSLOADER/BAS
1Ø CLS:PRINT"DFSEARCH USR SUBROUTINE LOADER"
2Ø BA%=&HFEØØ: A%=BA% 'DFSEARCH ADDRESS - MODIFY AS DESIRED
21 BA!=ASC(MKI$(BA%))+ASC(MID$(MKI$(BA%),2))*256
3Ø PRINT:PRINT"THIS PROGRAM POKES DFSEARCH TO MEMORY,":PRINT"STA
RTING AT ADDRESS:";BA!
4Ø MS!=PEEK(16561)+PEEK(16562)*256+1
```

```
41 PRINT:PRINT"CURRENT MEMORY SIZE SETTING IS";MS!
42 IFMS!>BA!THENPRINT"YOU'VE NOT RESERVED ENOUGH MEMORY!  PLEASE
 RESTART BASIC.":END
5Ø PRINT:PRINT"PRESS <ENTER> TO BEGIN...":LINEINPUTA$
51 PRINT"POKING IN DFSEARCH..."
1ØØ FORX=1T0254                 'FOR EACH BYTE OF DFSEARCH
1Ø1 READ B%                     'READ A BYTE TO BE POKED
1Ø2 POKE A%, B%                 'POKE THE BYTE INTO THE ADDRESS
1Ø3 A%=A%+1                     'POINT TO NEXT ADDRESS
1Ø4 NEXT                        'REPEAT UNTIL DONE
1Ø5 PRINT
11Ø PRINT"DOES YOUR DISK OPERATING SYSTEM USE:"
111 PRINT"  A> A 5Ø-BYTE DCB, LIKE TRSDOS 1.3 FOR THE MODEL 3"
112 PRINT"  B> A 32-BYTE DCB, LIKE TRSDOS 2.3 FOR THE MODEL 1"
12Ø LINEINPUTA$:IFA$<>"A"ANDA$<>"B"THEN11Ø
121 IFA$="A"THENPOKEBA%+12Ø,5Ø  'POKE BUFFER OFFSET FROM DCB
122 IFA$="B"THENPOKEBA%+12Ø,32  'POKE BUFFER OFFSET FROM DCB
2ØØ PRINT:PRINT"YOU MAY NOW USE DFSEARCH"
21Ø PRINT"YOUR DEFUSR ADDRESS IS  ";BA%
22Ø END
6Ø52Ø 'DFSEARCH  254 BYTES
6Ø521 DATA 2Ø5,127,1Ø,229,221,225,78,221,11Ø,4,221,1Ø2,5,7Ø,35,9
4,35,86,197,213,221,86,1Ø,221,11Ø,2,221,1Ø2,3,229,43,125,1Ø8
6Ø522 DATA 38,Ø,92,6,16,253,33,Ø,Ø,41,23,48,1,44,253,41,253,35,1
83,237,82,48,3,25,253,43,16,237,221,116,3Ø,175,221,78,12,36,37
6Ø523 DATA 4Ø,3,129,24,25Ø,111,229,253,229,225,221,94,8,221,86,9
,25,221,94,6,221,86,7,213,253,225,35,221,117,28,221,116,29,253
6Ø524 DATA 78,1Ø,253,7Ø,11,229,183,237,66,193,11,245,253,229,225
,17,32,Ø,25,241,4Ø,27,253,229,2Ø9,2Ø5,66,68,32,3,2Ø5,54,68,4Ø
6Ø525 DATA 14,38,255,47,111,35,193,193,193,193,195,154,1Ø,24,199
,193,9,221,94,14,22,Ø,25,2Ø9,221,78,3Ø,221,7Ø,1Ø,221,54,3Ø,Ø
6Ø526 DATA 229,19,217,225,2Ø9,193,197,213,229,4,5,4Ø,55,26,19Ø,3
2,1Ø,35,19,16,248,2Ø3,65,32,43,24,12,56,6,2Ø3,73,32,35,24,4,2Ø3
6Ø527 DATA 81,32,29,217,225,213,12,121,144,48,9,221,94,12,22,Ø,2
5,2Ø9,24,2ØØ,33,Ø,Ø,229,221,11Ø,28,221,1Ø2,29,24,167,217,221
6Ø528 DATA 113,3Ø,235,221,117,2,221,116,3,43,24,147
```

The Control Array

Communication between your BASIC program and the DFSEARCH USR routine is done with a 16-element control array. This integer array has the same layout as the control array used for the KEYACCES USR subroutine, so your BASIC program can use the same array for both if you wish. You'll be loading information into the array to specify the parameters for your search. Then, when calling DFSEARCH, your argument with the USR call is the VARPTR of the first element in the control array. Here are the array elements and how they are used:

Element 0 is loaded by your BASIC program before calling DFSEARCH. It is a number from 1 to 7 that specifies the type of search you want:

1 = Search for the first record in which the specified field is equal to the search key.

2 = Search for the first record in which the specified field is less than the search key.

3 = Find the first record in which the field is less than or equal.

4 = Find the first record in which the field is greater than the search key.

5 = Search for the first record in which the specified field greater than or equal to the search key.

6 = Find the first record in which the specified field is not equal to the search key.

7 = Don't do a search. Just get the starting record specified and do some computations that will aid in fielding.

Element 1 is loaded by your BASIC program to specify the logical record number at which the search is to begin. The first logical record of the file is considered to be logical record 1. Upon return from a successful search, element 1 contains the number of the next logical record, so you don't need to reload it if you want to resume your search.

Element 2 is loaded by your BASIC program to specify the VARPTR of the string you are using for your search key. If your search string is stored in a simple (non-array) variable, you'll only need to load element 2 once, early in your program.

Element 3 is the address of the Data Control Block for the disk file you are using. Appendix 1 tells you how to find the Data Control Block Addresses.

Element 4 is the file offset. In most cases you'll want to specify this as zero, indicating that the first logical record is in the first physical record of the file. In some cases, though, you may wish to start data storage at positions other than the first physical record, so that you can store other types of information at the beginning of the file. Suppose, for example, that you want to consider the 35th physical record of the file as logical record 1. You'd load element 4 with 34.

Element 5 is the blocking factor. It must be a number from 1 to 255. You load element 5 with the number of subrecords you are storing in each physical record. If, for example, you are blocking four 64-byte logical records per physical record, element 5 should be loaded with 4. In the case of variable length records, blocking is done automatically for you, so you should load element 5 with 1.

Element 6 is the logical record length. It may be from 1 to 256 bytes.

Element 7 is the key offset within each logical record. You load element 7 to specify the point in each logical record at which the comparisons are to begin. If you are searching based on a field beginning at the first byte of each record, element 7 is loaded with 0. If your comparisons begin at the third byte of each record, element 7 is loaded with 2, because 2 bytes precede the comparison field.

Elements 8 through 13 are unused by the DFSEARCH USR subroutine. They are left unused so that the same control array can be shared with KEYACCES.

Element 14 is not loaded by your BASIC program. Instead, after a DFSEARCH call, your BASIC program can use it to find the number of the physical record at which the search terminated. It is provided for the convenience of your BASIC program if you are using TRSDOS 2.3. If you've got other versions of TRSDOS, you can use the LOC function to get the same information. If you are using the random disk file handler subroutines, you may wish to load PR%(PF%) and PP%(PF%) with this number after a search.

Element 15 returns the number of preceding subrecords within the physical record when a key is found. If, for example, the key is found in the 4th subrecord, element 15 is returned as 3. This is for the convenience of your BASIC program in cases where you will want to FIELD the record after a successful search. If you are using the random disk file handler subroutines, you may wish to load LS% with the contents of element 15 after a search. Then you can GOSUB directly to your logical record fielding subroutine, 58010, 58020, 58030 or whatever, depending on the file buffer number.

Dimensioning and Loading

It is very important for your program to dimension your control array with a minimum of 16 elements. If, for example, you are using C% as your control array, you will need a statement such as:

```
DIM C%(15)
```

For more information about some of the variations you may use in dimensioning, you can refer to the KEYACCES instructions.

Once you've dimensioned your control array, you can pre-load the values that will remain constant throughout your program. You can do this in one program line if you wish. Here are sample statements that do the job:

```
DIM C%(15)        'DIMENSION C% AS THE CONTROL ARRAY
KY$=""            'INITIALIZE STRING TO BE USED AS SEARCH KEY
C%(2)=VARPTR(KY$) 'PRE-LOAD SEARCH KEY VARPTR
C%(3)=26883       'USE FILE 3, UNDER MODEL 1 TRSDOS 2.3
C%(4)=Ø           'FILE BEGINS AT PHYSICAL RECORD 1 OF FILE 3
C%(5)=4           'BLOCKING FACTOR IS 4
C%(6)=64          'LOGICAL RECORD LENGTH IS 64 BYTES
```

Your DEFUSR Statement

To call it from BASIC, you need to tell the system where DFSEARCH resides in memory. This is done with a DEFUSR statement. BASIC maintains a list of up to ten USR routine addresses. DEFUSR0 defines the first, DEFUSR1 defines the second, and so forth. You can use any one of them for DFSEARCH. Usually this can be done early in your program with a statement such as:

```
DEFUSR4 = -512
```

or,

```
DEFUSR4 = &HFEØØ
```

Both of these define the same address. It is the address where DSLOADER/BAS as listed pokes DFSEARCH. We will arbitrarily use USR4 for all our examples.

Opening and Closing

The DFSEARCH USR routine does no opens or closes. It is up to your BASIC program to perform these functions. No special considerations are involved, except that the file to be searched must have been opened in random mode. The file buffer number you use doesn't matter, but you must specify the correct DCB address in control array element 3.

Searching the File

To search for a record within the file, simply load the required elements of your control array and call the DFSEARCH USR routine, using the VARPTR of your control array as the argument. If the search is successful, the logical record number containing the first match will be returned to BASIC. If the search is unsuccessful, the entire disk file will have been searched, and "–28", indicating "end of file encountered", will be returned. If an error condition is encountered during the search, the minus value of the error code will be returned. The error codes used (as with KEYACCES) are the TRSDOS error codes listed in your *Disk System Owner's Manual.*

Let's assume, for example that you've loaded DFSEARCH, defined it as USR routine 4 with the DEFUSR command, and you've preloaded the control array, C%, as shown above. Let's say that, starting at the 25th byte of each 64-byte logical record is a 2-byte field that contains a state code, such as "CA" for California, "NY" for New York, and so forth. If file 3 has been opened, you can use the following logic to search for the first logical record in the file containing "CA":

```
5ØØ KY$="CA"                'LOAD THE SEARCH KEY
5Ø1 C%(Ø)=1                 'SEARCH FOR EXACT MATCH
5Ø2 C%(1)=1                 'START FROM FIRST LOGICAL RECORD
5Ø3 C%(7)=24                'LOAD KEY OFFSET
51Ø J%=USR4(VARPTR(C%(Ø))) 'CALL DFSEARCH USR SUBROUTINE
52Ø PRINT J%                'PRINT RESULTS OF THE SEARCH
```

If you want the logic to find all the logical record numbers containing "CA", add the following lines:

```
53Ø IF J%=-28 THEN PRINT "WHOLE FILE SEARCHED!" : GOTO ....
531 IF J%<Ø   THEN PRINT "DISK ERROR!"          : GOTO ....
55Ø PRINT "CONTINUING SEARCH... " : GOTO 51Ø
```

If you are using the random file handler subroutines and you want to print information from the logical record when a match is found, you can field it and print it by adding the following lines before line 550:

```
535 PF%=3                   'SPECIFY FILE NUMBER
536 LS%=C%(15)              'LOAD PRECEDING SUBRECORD COUNT
537 PR%(PF%)=C%(14)         'LOAD PHYSICAL RECORD NUMBER
538 PP%(PF%)=C%(14)         'LOAD PHYSICAL RECORD NUMBER
539 GOSUB 58Ø3Ø             'GO FIELD THE RECORD FOR FILE 3
54Ø PRINT .... : PRINT ...  'PRINT INFORMATION FROM THE RECORD
```

If you are not using the random disk file handler subroutines, you can field the record and print it with statements similar to:

```
535 FIELD 3, C%(15)*64 AS DM$, 24 AS AD$, 2 AS ST$, 38 AS XX$
54Ø PRINTDM$ : PRINTST$ : PRINT XX$
```

Notice that you used the number returned in element 15 of the control array, multiplied by the logical record length, to create a dummy field, DM$. Following the dummy field, you have the fields that correspond to the logical record layout. In this example, you have 3 data fields, AD$, ST$ and XX$.

Searching for Partial Matches

Notice that none of the 16 elements of a DFSEARCH control array specifies the length of the field to be searched. Only the position within the logical record is indicated. The comparison length is controlled only by the length of the search key you provide. If, in the above example, you wanted to find all the records with state codes that start with "C" , you could change line 500 so it reads:

```
5ØØ KY$="C"                'LOAD THE SEARCH KEY
```

With this change, state codes such as "CA", "CT" and "CO" are all considered matches.

If you want DFSEARCH to find exact matches on an entire field you have to make your search key equal to the length of the field you want to search. Usually you can do this by padding the search key with blanks. You also need to insure that the operator doesn't enter a search key that's longer than you intend. It may be necessary to truncate characters from the right to make the key the proper length. On the other hand, the fact that DFSEARCH disregards the way you've fielded a file can be used to advantage. You may want to make your search key span more than one adjacent field.

Searching for Inequalities

You can change element 0 of the control array to search for records having a field greater or less than a key. In the above example, you could find all records having a state code that is alphabetically greater than or equal to "MA" by changing lines 500 and 501 to read:

```
5ØØ KY$="MA"               'LOAD THE SEARCH KEY
5Ø1 C%(Ø)=5                'SEARCH FOR ALL GREATER OR EQUAL
```

You can find all records that have a state code other than "NY" with the following logic:

```
5ØØ KY$="NY"               'LOAD THE SEARCH KEY
5Ø1 C%(Ø)=6                'SEARCH FOR ALL NOT EQUAL
```

DFSEARCH does not support "less than" or "greater than" searches for numbers stored in MKI$, MKS$ or MKD$ format. *BASIC Faster and Better & Other Mysteries* shows some ways to get around this limitation.

Using DFSEARCH to Compute and GET

If you use a null string for the seach key, or if you load 7 into element 0 of the control array, the logical record specified by element 1 is retrieved. No search is done. You can use this technique as a fast and convenient method to compute the physical record and subrecord number and to do your GET. Upon return from DFSEARCH, elements 14 and 15 of the control array contain the physical record number and number of preceding subrecords, respectively, and the disk file buffer contains the physical record.

To return, one-by-one, all records in the file, change the example so line 500 reads:

```
5ØØ KY$=""                 'LOAD THE SEARCH KEY
```

You can change line 501 so it reads:

```
5Ø1 C%(Ø)=7                'SEARCH FOR ALL KEYS
```

Either approach will retrieve each record in the file, one-by-one.

Demonstrating and Testing DFSEARCH

DFSEARCH/BAS is a short program that you can use to demonstrate and test the DFSEARCH USR routine. Since it lets you enter the parameters from the keyboard, you can keep it in your library as a utility program that will be handy from time to time.

Before using DFSEARCH/BAS, you'll want to create a test file to try out some of its features. It is most convenient when testing routines of this type to simply use numbers as the test file data. That way you can be sure the routine is working properly and you know what keys are in the file.

The following program creates a file named "TEST" on drive 1. It contains 150 logical records. Each logical record contains the STR$ equivalent of the logical record number, left justified in an 85-byte field. So, for example, the first 2 bytes of logical record 29 are "2" and "9". The logical records are blocked 3 per physical record, so the file contains 50 physical records:

Figure 15.2 — *TESTFILE PROGRAM*

```
Ø 'NOTE: THIS PROGRAM BUILDS A FILE FOR TESTING DFSEARCH
1Ø OPEN"R",1,"TEST:1"      'OPEN THE FILE
2Ø FIELD1,85ASF$(1),85ASF$(2),85ASF$(3)  'FIELD IT
3Ø FORX=1TO5Ø              'FOR EACH PHYSICAL RECORD
4Ø FORY=1TO3               'FOR EACH SUBRECORD
41 LSETF$(Y)=MID$(STR$((X-1)*3+Y),2)     'PUT DATA IN FIELD
42 PRINT F$(Y)             'DISPLAY THE FIELD TO BE WRITTEN
43 NEXT                    'REPEAT FOR NEXT SUBRECORD
6Ø PUT1,X                  'WRITE IT TO DISK
7Ø NEXT                    'REPEAT FOR NEXT PHYSICAL RECORD
8Ø CLOSE                   'CLOSE THE FILE
```

Now, when you run DFSEARCH/BAS, you can specify "TEST:1" as the file, 0 as the file offset, 3 as the blocking factor, and 85 as the logical record length. Here's how the video display looks when you run it with TRSDOS 1.3 on a TRS-80 Model III

```
FILE NAME AND DRIVE ? TEST:1
OPENING AS FILE 1
FILE 1 DCB ADDRESS  ? 26182
FILE OFFSET         ? Ø
BLOCKING FACTOR     ? 3
LOGICAL REC LENGTH  ? 85
```

To search for the first record whose left two bytes match the search key "75", you can enter the following:

```
KEY                 ? 75
KEY OFFSET          ? Ø
STARTING RECORD NO. ? 1
COMMAND             ? 1
```

After entry of these parameters, DFSEARCH/BAS searches the file. Two numbers are returned, 75 and –28. The 75 indicates that logical record 75 contains the first match.

– 28 is captured by the program and is not displayed. It indicates that no further matches are found, so an "END OF FILE" message is shown.

If you want to find all records whose leftmost byte is "7", you can use "7" as the key. The program will display 7, and 70 through 79 as the logical records that contain a match.

If you want to find all records having "0" in the second byte, you can use "0" as the key, and 1 as the key offset.

You can try many other variations for the parameters. Here is the DFSEARCH/BAS program listing:

Figure 15.3 — *DFSEARCH/BAS*

```
Ø 'DFSEARCH/BAS
1 CLS:CLEAR1ØØØ          'CLEAR SCREEN & 1ØØØ BYTES STRING SPACE
2 DEFINTA-Z:J=Ø          'DEFINE ALL VARIABLES AS INTEGER
3 KY$=""                 'INITIALIZE SEARCH KEY VARIABLE
4 DEFUSR4= -512          'DEFINE ADDRESS OF DFSEARCH ROUTINE
5 DIM DS(15)             'USE DS AS THE CONTROL ARRAY
1Ø IFPEEK(-512)<>2Ø5 OR PEEK(-512+253)<>147 THEN PRINT "PLEASE R
UN DSLOADER/BAS TO PUT DFSEARCH IN MEMORY":END
1ØØ LINEINPUT"FILE NAME AND DRIVE ? ";FS$
11Ø PRINT "OPENING AS FILE 1"
12Ø OPEN"R",1,FS$                'OPEN THE REQUESTED FILE
13Ø INPUT"FILE 1 DCB ADDRESS  ";DS(3)
14Ø INPUT"FILE OFFSET         ";DS(4)
15Ø INPUT"BLOCKING FACTOR     ";DS(5)
16Ø INPUT"LOGICAL REC LENGTH  ";DS(6)
17Ø DS(2)=VARPTR(KY$)            'LOAD VARPTR OF THE SEARCH KEY
18Ø PRINT                        'PRINT A BLANK LINE

2ØØ LINEINPUT"SEARCH KEY             ? ";KY$
21Ø IFKY$=""THEN END           'CLOSE & END IF NO KEY ENTERED

22Ø INPUT"KEY OFFSET           ";DS(7)
23Ø INPUT"STARTING RECORD NO. ";DS(1)
24Ø INPUT"COMMAND              ";DS(Ø)
25Ø IF DS(Ø)=Ø THEN END      'CLOSE & END IF Ø COMMAND ENTERED

3ØØ J=USR4(VARPTR(DS(Ø)))    'CALL DFSEARCH
31Ø IF J=-28 THEN PRINT "END OF FILE" : GOTO 2ØØ
32Ø IF J<1 THEN PRINT "ERROR CODE ";J : GOTO 2ØØ
33Ø PRINT "FOUND, LOGICAL RECORD";J;" PHYSICAL RECORD";DS(14)
34Ø GOTO 3ØØ                 'CONTINUE SEARCH UNTIL EOF OR ERROR
```

Notice that DFSEARCH/BAS does a quick check to verify that DFSEARCH has been loaded into protected memory where it expects it. It does this in line 10 by looking at what should be the first and last bytes of the routine. DFSEARCH/BAS as listed requires that you first run DSLOADER/BAS, to put the USR routine at address –512. If you want to put DFSEARCH in a different memory location, be sure to change the –512's in lines 4 and 10.

Notice that line 310 looks for error code –28. This code indicates that the end of the file has been encountered. Other error conditions will fall through to line 320. Your *Disk System Owner's Manual* gives the translations. One code that you may possibly get is –38. It corresponds to error code 38, "I/O Attempt to Unopen File". The usual cause of this is an incorrect DCB address. In the event of any error, you may want to break out of the the program to display the contents of DS(1) and DS(14). They will tell you the current logical record and physical record number that DFSEARCH/BAS is attempting to read.

Magic Array and Poke Formats

The DFSEARCH magic array chart gives the 128 numbers that may be loaded into an integer array to execute DFSEARCH. To execute DFSEARCH in an integer array you don't need to protect memory. Your DEFUSR should define the VARPTR of element 0 in the array as the starting address for DFSEARCH. Remember that it is important to do your DEFUSR just before you call to DFSEARCH because arrays tend to move in memory. This technique is discussed in *BASIC Faster and Better & Other Mysteries.*

The POKE chart gives you another list of the 254 numbers that can be POKEd into protected memory. You can put them in data statements of a BASIC program, in a sequential file, in one record of a random file, or you can use DUMP so that you'll be able to put DFSEARCH in memory with your operating system's LOAD command.

Remember that DFSEARCH as it is listed makes the assumption that the disk buffer starts 32 bytes above the first byte of the data control block. On TRSDOS 1.3 for the Model III and some other disk operating systems, the buffer starts 50 bytes above the first byte of the DCB. You'll need to make the following modifications:

- If you are using the magic array method you will need to change the 61st element from "32" to "50".
- If you are using the poke method you'll need to modify the 121st byte of the routine from "32" to "50".
- If you are reassembling DFSEARCH, you'll need to change line 990.

Magic Array Format, 122 Elements

```
 32717  -69Ø2  -7715  -8882   1134  26333  17925  24Ø99  22Ø51
-1Ø811  22237  -895Ø    622  26333  -69Ø9  32Ø43   9836  23552
  41Ø2   87Ø1      Ø   5929    3Ø4   -724   -727 -18653  21229
   816   -743   4139  -8723   7796  -8785   315Ø   95Ø8    8Ø8
  6273  28666   -539  -77Ø7  24285  -8952   239Ø  -8935   163Ø
 22237 -11ØØ1  -7683  -8925   7285  29917   -739   2638  18173
 -69Ø1  -4681 -16Ø62  -28Ø5  -6659   4577     32  -3815   6952
 -6659 -12847  17474    8ØØ  14Ø29  1Ø3Ø8   9742  12287   9Ø71
-15935 -15935 -25917   6154 -15929  -8951   3678     22 -12ØØ7
 2Ø189  -893Ø   263Ø  14Ø45     3Ø   5Ø93  -7719 -15919 -1Ø811
  1253  1Ø245   6711   8382   897Ø   4115 -1332Ø   8257   6187
 14348 -13562   8265   6179 -13564   8273  -9955 -1Ø783  3Ø988
 12432  -8951   3166     22 -12ØØ7 -14312     33  -6912  28381
 -8932   7526 -2276Ø  -8743   7793  -8725    629  29917  11Ø11
-2788Ø
```

Poke Format, 254 Bytes

```
2Ø5 127  1Ø 229 221 225  78 221 11Ø   4 221 1Ø2   5  7Ø  35  94
 35  86 197 213 221  86  1Ø 221 11Ø   2 221 1Ø2   3 229  43 125
1Ø8  38   Ø  92   6  16 253  33   Ø   Ø  41  23  48   1  44 253
 41 253  35 183 237  82  48   3  25 253  43  16 237 221 116  3Ø
175 221  78  12  36  37  4Ø   3 129  24 25Ø 111 229 253 229 225
221  94   8 221  86   9  25 221  94   6 221  86   7 213 253 225
 35 221 117  28 221 116  29 253  78  1Ø 253  7Ø  11 229 183 237
 66 193  11 245 253 229 225  17  32   Ø  25 241  4Ø  27 253 229
2Ø9 2Ø5  66  68  32   3 2Ø5  54  68  4Ø  14  38 255  47 111  35
193 193 193 193 195 154  1Ø  24 199 193   9 221  94  14  22   Ø
 25 2Ø9 221  78  3Ø 221  7Ø  1Ø 221  54  3Ø   Ø 229  19 217 225
2Ø9 193 197 213 229   4   5  4Ø  55  26 19Ø  32  1Ø  35  19  16
248 2Ø3  65  32  43  24  12  56   6 2Ø3  73  32  35  24   4 2Ø3
 81  32  29 217 225 213  12 121 144  48   9 221  94  12  22   Ø
 25 2Ø9  24 2ØØ  33   Ø   Ø 229 221 11Ø  28 221 1Ø2  29  24 167
217 221 113  3Ø 235 221 117   2 221 116   3  43  24 147
```

Figure 15.4 — *DFSEARCH Assembly Listing*

```
                 ØØØØØ ;DFSEARCH
                 ØØØØ1 ;
FEØØ             ØØ22Ø        ORG     ØFEØØH          ;ORIGIN - RELOCATABLE
FEØØ CD7FØA      ØØ23Ø        CALL    ØA7FH           ;PUT USR ARGUMENT IN HL
FEØ3 E5          ØØ24Ø        PUSH    HL              ;
FEØ4 DDE1        ØØ25Ø        POP     IX              ;IX POINTS TO CONTROL ARRAY
                 ØØ26Ø ;THE FOLLOWING LOGIC PUTS COMMAND AND STRING POINTERS ON STK
FEØ6 4E          ØØ27Ø        LD      C,(HL)          ;C HAS COMMAND
FEØ7 DD6EØ4      ØØ28Ø        LD      L,(IX+4)        ;
FEØA DD66Ø5      ØØ29Ø        LD      H,(IX+5)        ;HL HAS SKEY VARPTR
FEØD 46          ØØ3ØØ        LD      B,(HL)          ;B HAS SKEY LENGTH
FEØE 23          ØØ31Ø        INC     HL              ;
FEØF 5E          ØØ32Ø        LD      E,(HL)          ;
FE1Ø 23          ØØ33Ø        INC     HL              ;
FE11 56          ØØ34Ø        LD      D,(HL)          ;DE POINTS TO SKEY
FE12 C5          ØØ35Ø        PUSH    BC              ;SAVE LENGTH AND COMMAND
FE13 D5          ØØ36Ø        PUSH    DE              ;SAVE POINTER TO SEARCH KEY
                 ØØ37Ø ;THE FOLLOWING LOGIC COMPUTES STARTING PHYSICAL RECORD
FE14 DD56ØA      ØØ38Ø        LD      D,(IX+1Ø)       ;D HAS BLOCKING FACTOR
FE17 DD6EØ2      ØØ39Ø        LD      L,(IX+2)        ;
FE1A DD66Ø3      ØØ4ØØ        LD      H,(IX+3)        ;HL HAS LOGICAL REC NUMBER
FE1D E5          ØØ41Ø        PUSH    HL              ;SAVE REC NUMBER ON STACK
FE1E 2B          ØØ42Ø        DEC     HL              ;FIRST LOGICAL REC IS Ø
FE1F 7D          ØØ43Ø        LD      A,L             ;DIVIDE HL BY D
FE2Ø 6C          ØØ44Ø        LD      L,H             ;CONTINUE DIVISION
FE21 26ØØ        ØØ45Ø        LD      H,ØH            ;CONTINUE DIVISION
FE23 5C          ØØ46Ø        LD      E,H             ;CONTINUE DIVISION
FE24 Ø61Ø        ØØ47Ø        LD      B,1ØH           ;CONTINUE DIVISION
FE26 FD21ØØØØ    ØØ48Ø        LD      IY,ØH           ;CONTINUE DIVISION
```

```
FE2A 29       ØØ49Ø J1A     ADD     HL,HL          ;CONTINUE DIVISION
FE2B 17       ØØ5ØØ         RLA                    ;CONTINUE DIVISION
FE2C 3ØØ1     ØØ51Ø         JR      NC,J1B         ;CONTINUE DIVISION
FE2E 2C       ØØ52Ø         INC     L              ;CONTINUE DIVISION
FE2F FD29     ØØ53Ø J1B     ADD     IY,IY          ;CONTINUE DIVISION
FE31 FD23     ØØ54Ø         INC     IY             ;CONTINUE DIVISION
FE33 B7       ØØ55Ø         OR      A              ;CONTINUE DIVISION
FE34 ED52     ØØ56Ø         SBC     HL,DE          ;CONTINUE DIVISION
FE36 3ØØ3     ØØ57Ø         JR      NC,J1C         ;CONTINUE DIVISION
FE38 19       ØØ58Ø         ADD     HL,DE          ;CONTINUE DIVISION
FE39 FD2B     ØØ59Ø         DEC     IY             ;CONTINUE DIVISION
FE3B 1ØED     ØØ6ØØ J1C     DJNZ    J1A            ;CONTINUE DIVISION
FE3D DD741E   ØØ61Ø         LD      (IX+3Ø),H      ;SAVE PRECEDING SUBRECS
              ØØ62Ø ;THE FOLLOWING LOGIC FINDS OFFSET TO FIRST LOGICAL RECORD
FE4Ø AF       ØØ63Ø         XOR     A              ;CLEAR ACCUMULATOR
FE41 DD4EØC   ØØ64Ø         LD      C,(IX+12)      ;C HAS RECORD LENGTH
FE44 24       ØØ65Ø J1D     INC     H              ;
FE45 25       ØØ66Ø J1F     DEC     H              ;SET Z FLAG IF H IS ZERO
FE46 28Ø3     ØØ67Ø         JR      Z,J1E          ;JUMP IF ZERO
FE48 81       ØØ68Ø         ADD     A,C            ;
FE49 18FA     ØØ69Ø         JR      J1F            ;
FE4B 6F       ØØ7ØØ J1E     LD      L,A            ;HL = OFFSET TO LOGICAL REC
FE4C E5       ØØ71Ø         PUSH    HL             ;SAVE IT ON STACK
              ØØ72Ø ;THE FOLLOWING LOGIC FINDS FIRST PHYSICAL RECORD FOR SEARCH
FE4D FDE5     ØØ73Ø         PUSH    IY             ;
FE4F E1       ØØ74Ø         POP     HL             ;HL HAS DIVISION QUOTIENT
FE5Ø DD5EØ8   ØØ75Ø         LD      E,(IX+8)       ;
FE53 DD56Ø9   ØØ76Ø         LD      D,(IX+9)       ;DE HAS PHYS REC OFFSET
FE56 19       ØØ77Ø         ADD     HL,DE          ;HL HAS START PHYS REC
              ØØ78Ø ;
              ØØ79Ø ;THE FOLLOWING LOGIC POINTS IY TO FILE DCB ADDRESS
FE57 DD5EØ6   ØØ8ØØ         LD      E,(IX+6)       ;
FE5A DD56Ø7   ØØ81Ø         LD      D,(IX+7)       ;DE HAS FILE DCB ADDRESS
FE5D D5       ØØ82Ø         PUSH    DE             ;
FE5E FDE1     ØØ83Ø         POP     IY             ;IY HAS FILE DCB ADDRESS
              ØØ84Ø ;
              ØØ85Ø ;THE FOLLOWING LOGIC GETS PHYSICAL RECORD POINTED BY HL
FE6Ø 23       ØØ86Ø J2A     INC     HL             ;HL=NEXT PHYS RECORD NUMBER
FE61 DD751C   ØØ87Ø         LD      (IX+28),L      ;
FE64 DD741D   ØØ88Ø         LD      (IX+29),H      ;CONTROL 14 HAS REC NUMBER
FE67 FD4EØA   ØØ89Ø         LD      C,(IY+1Ø)      ;
FE6A FD46ØB   ØØ9ØØ         LD      B,(IY+11)      ;BC HAS REC NUMBER IN BUFFER
FE6D E5       ØØ91Ø         PUSH    HL             ;SAVE DESIRED RECORD NUMBER
FE6E B7       ØØ92Ø         OR      A              ;CLEAR CARRY
FE6F ED42     ØØ93Ø         SBC     HL,BC          ;COMPARE DESIRED TO CURRENT
FE71 C1       ØØ94Ø         POP     BC             ;RESTORE DESIRED REC TO BC
FE72 ØB       ØØ95Ø         DEC     BC             ;Ø IS FIRST REC FOR DOS CALL
FE73 F5       ØØ96Ø         PUSH    AF             ;SAVE FLAGS
FE74 FDE5     ØØ97Ø         PUSH    IY             ;
FE76 E1       ØØ98Ø         POP     HL             ;HL POINTS TO DCB
FE77 112ØØØ   ØØ99Ø         LD      DE,2ØH         ;ADD 32 TO DCB ADDRESS
              ØØ991 ;NOTE:  CHANGE ABOVE TO "LD DE,32H" IF DCB IS 5Ø BYTES
```

```
                ØØ992 ;       AS IT IS FOR TRS-8Ø MODEL 3 DISK BASIC
FE7A 19         Ø1ØØØ         ADD     HL,DE           ;HL POINTS TO BUFFER
FE7B F1         Ø1Ø1Ø         POP     AF              ;RESTORE FLAGS
FE7C 281B       Ø1Ø2Ø         JR      Z,J2B           ;NO GET REQUIRED IF EQUAL
FE7E FDE5       Ø1Ø3Ø         PUSH    IY              ;
FE8Ø D1         Ø1Ø4Ø         POP     DE              ;DE POINTS TO FILE DCB
FE81 CD4244     Ø1Ø5Ø         CALL    Ø4442H          ;POSITION
FE84 2ØØ3       Ø1Ø6Ø         JR      NZ,J2C          ;JUMP IF DISK ERROR
FE86 CD3644     Ø1Ø7Ø         CALL    Ø4436H          ;READ
FE89 28ØE       Ø1Ø8Ø J2C     JR      Z,J2B           ;SKIP FOLLOWING IF NO ERROR
FE8B 26FF       Ø1Ø9Ø         LD      H,ØFFH          ;
FE8D 2F         Ø11ØØ         CPL                     ;
FE8E 6F         Ø111Ø         LD      L,A             ;
FE8F 23         Ø112Ø         INC     HL              ;HL HAS MINUS OF ERROR CODE
                Ø113Ø ;THE FOLLOWING LOGIC RELIEVES STACK AND RETURNS TO BASIC
FE9Ø C1         Ø114Ø J9A     POP     BC              ;
FE91 C1         Ø115Ø J9B     POP     BC              ;
FE92 C1         Ø116Ø         POP     BC              ;
FE93 C1         Ø117Ø         POP     BC              ;
FE94 C39AØA     Ø118Ø         JP      ØA9AH           ;RETURN HL TO BASIC
                Ø119Ø ;THE FOLLOWING LOGIC HANDLES A HALF-JUMP FOR RELOCATABILITY
FE97 18C7       Ø12ØØ J2AH    JR      J2A             ;
                Ø121Ø ;THE FOLLOWING LOGIC POINTS HL TO LOGICAL RECORD IN BUFFER
FE99 C1         Ø122Ø J2B     POP     BC              ;GET OFFSET
FE9A Ø9         Ø123Ø         ADD     HL,BC           ;HL POINTS TO RECORD
FE9B DD5EØE     Ø124Ø         LD      E,(IX+14)       ;
FE9E 16ØØ       Ø125Ø         LD      D,Ø             ;
FEAØ 19         Ø126Ø         ADD     HL,DE           ;ADD OFFSET IN LOGICAL REC
FEA1 D1         Ø127Ø         POP     DE              ;DE= LOGICAL RECORD NUMBER
FEA2 DD4E1E     Ø128Ø         LD      C,(IX+3Ø)       ;C HAS PRECEDING SUBRECORDS
FEA5 DD46ØA     Ø129Ø         LD      B,(IX+1Ø)       ;B HAS BLOCKING FACTOR
FEA8 DD361EØØ   Ø13ØØ         LD      (IX+3Ø),Ø       ;LOAD SUBRECORD # FOR NEXT
FEAC E5         Ø131Ø J3A     PUSH    HL              ;
FEAD 13         Ø132Ø         INC     DE              ;DE=NEXT RECORD NUMBER
FEAE D9         Ø133Ø         EXX                     ;EXCHANGE REGISTERS
                Ø134Ø ;THE FOLLOWING LOGIC DOES THE COMPARISONS
FEAF E1         Ø135Ø         POP     HL              ;GET POINTER TO DISK BUFFER
FEBØ D1         Ø136Ø         POP     DE              ;GET POINTER TO SKEY
FEB1 C1         Ø137Ø         POP     BC              ;GET SKEY LENGTH & COMMAND
FEB2 C5         Ø138Ø         PUSH    BC              ;PUT ON STACK AGAIN
FEB3 D5         Ø139Ø         PUSH    DE              ;PUT ON STACK AGAIN
FEB4 E5         Ø14ØØ         PUSH    HL              ;PUT ON STACK AGAIN
FEB5 Ø4         Ø141Ø         INC     B               ;
FEB6 Ø5         Ø142Ø         DEC     B               ;
FEB7 2837       Ø143Ø         JR      Z,MATCH         ;ZERO SKEY LENGTH IS MATCH
FEB9 1A         Ø144Ø CPLOOP  LD      A,(DE)          ;GET BYTE FROM SKEY
FEBA BE         Ø145Ø         CP      (HL)            ;COMPARE TO BYTE IN BUFFER
FEBB 2ØØA       Ø146Ø         JR      NZ,NOTEQ        ;JUMP IF NOT EQUAL
FEBD 23         Ø147Ø         INC     HL              ;POINT TO NEXT BYTE
FEBE 13         Ø148Ø         INC     DE              ;POINT TO NEXT BYTE
FEBF 1ØF8       Ø149Ø         DJNZ    CPLOOP          ;DECREMENT COUNT AND REPEAT
FEC1 CB41       Ø15ØØ EQ      BIT     Ø,C             ;DO WE WANT AN EQUAL?
```

```
FEC3 2Ø2B     Ø151Ø         JR      NZ,MATCH       ;WE'VE GOT MATCH IF WE DO.
FEC5 18ØC     Ø152Ø         JR      NOMA           ;OTHERWISE, NO MATCH
FEC7 38Ø6     Ø153Ø NOTEQ   JR      C,GREATR       ;IF C FLAG, SKEY IS GREATER
FEC9 CB49     Ø154Ø LESS    BIT     1,C            ;DO WE WANT A LESS?
FECB 2Ø23     Ø155Ø         JR      NZ,MATCH       ;WE'VE GOT MATCH IF WE DO.
FECD 18Ø4     Ø156Ø         JR      NOMA           ;OTHERWISE, NO MATCH
FECF CB51     Ø157Ø GREATR  BIT     2,C            ;DO WE WANT A GREATER?
FED1 2Ø1D     Ø158Ø         JR      NZ,MATCH       ;WE'VE GOT A MATCH IF WE DO.
FED3 D9       Ø159Ø NOMA    EXX                    ;EXCHANGE REGISTERS
FED4 E1       Ø16ØØ         POP     HL             ;GET BUFFER POINTER
FED5 D5       Ø161Ø         PUSH    DE             ;SAVE NEXT LOGICAL REC #
FED6 ØC       Ø162Ø         INC     C              ;INCREMENT COUNT OF SUBRECS
FED7 79       Ø163Ø         LD      A,C            ;PREPARE FOR SUBTRACT
FED8 9Ø       Ø164Ø         SUB     B              ;SUBTRACT
FED9 3ØØ9     Ø165Ø         JR      NC,NEXTPH      ;NEXT PHYSICAL IF CARRY
FEDB DD5EØC   Ø166Ø         LD      E,(IX+12)      ;ADD LOGICAL RECORD LENGTH
FEDE 16ØØ     Ø167Ø         LD      D,Ø            ;DE HAS LOGICAL REC LENGTH
FEEØ 19       Ø168Ø         ADD     HL,DE          ;POINT TO NEXT SUBRECORD
FEE1 D1       Ø169Ø         POP     DE             ;RESTORE RECORD NUMBER
FEE2 18C8     Ø17ØØ         JR      J3A            ;REPEAT FOR NEXT SUBRECORD
FEE4 21ØØØØ   Ø171Ø NEXTPH  LD      HL,Ø           ;NEW OFFSET IS ZERO
FEE7 E5       Ø172Ø         PUSH    HL             ;SAVE OFFSET ON STACK
FEE8 DD6E1C   Ø173Ø         LD      L,(IX+28)      ;
FEEB DD661D   Ø174Ø         LD      H,(IX+29)      ;HL HAS CURRENT PHYS REC
FEEE 18A7     Ø175Ø         JR      J2AH           ;GO GET NEXT
FEFØ D9       Ø176Ø MATCH   EXX                    ;EXCHANGE REGISTERS
FEF1 DD711E   Ø177Ø         LD      (IX+3Ø),C      ;SAVE SUBRECORD NUMBER
FEF4 EB       Ø178Ø         EX      DE,HL          ;RETURN RECORD NUMBER FOUND
FEF5 DD75Ø2   Ø179Ø         LD      (IX+2),L       ;
FEF8 DD74Ø3   Ø18ØØ         LD      (IX+3),H       ;CONTROL 1 HAS NEXT REC
FEFB 2B       Ø181Ø         DEC     HL             ;RET CURRENT NUMBER TO BASIC
FEFC 1893     Ø182Ø         JR      J9B            ;RETURN HL TO BASIC
FE91          Ø183Ø         END                    ;
ØØØØØ TOTAL ERRORS
```

Assembly Listing Cross-reference

```
CPLOOP FEB9 Ø144Ø   Ø149Ø
EQ     FEC1 Ø15ØØ
GREATR FECF Ø157Ø   Ø153Ø
J1A    FE2A ØØ49Ø   ØØ6ØØ
J1B    FE2F ØØ53Ø   ØØ51Ø
J1C    FE3B ØØ6ØØ   ØØ57Ø
J1D    FE44 ØØ65Ø
J1E    FE4B ØØ7ØØ   ØØ67Ø
J1F    FE45 ØØ66Ø   ØØ69Ø
J2A    FE6Ø ØØ86Ø   Ø12ØØ
J2AH   FE97 Ø12ØØ   Ø175Ø
J2B    FE99 Ø122Ø   Ø1Ø2Ø Ø1Ø8Ø
J2C    FE89 Ø1Ø8Ø   Ø1Ø6Ø
J3A    FEAC Ø131Ø   Ø17ØØ
J9A    FE9Ø Ø114Ø
```

```
J9B    FE91 Ø115Ø   Ø182Ø
LESS   FEC9 Ø154Ø
MATCH  FEFØ Ø176Ø   Ø143Ø Ø151Ø Ø155Ø Ø158Ø
NEXTPH FEE4 Ø171Ø   Ø165Ø
NOMA   FED3 Ø159Ø   Ø152Ø Ø156Ø
NOTEQ  FEC7 Ø153Ø   Ø146Ø
```

Model II Modifications for DFSEARCH

DFSEARCH may be used on the Model II under TRSDOS 2.0 or 2.0a. Some modifications are required to handle differences in the way that arguments are passed between BASIC and USR subroutines, differences in the DCB layout, and the supervisor call capability.

To the DFSEARCH assembly listing make the following changes:

```
As shown:  CD7FØA  ØØ23Ø    CALL  ØA7FH       ;PUT USR ARGUMENT IN HL
Change to: CD5D44  ØØ23Ø    CALL  Ø445DH      ;PUT USR ARGUMENT IN HL

As shown:  FD4EØA  ØØ89Ø    LD    C,(IY+1Ø)   ;
Change to: FD4EØ7  ØØ89Ø    LD    C,(IY+7)    ;

As shown:  FD46ØB  ØØ9ØØ    LD    B,(IY+11)   ;BC HAS REC NO. IN BUFF
Change to: FD46Ø8  ØØ9ØØ    LD    B,(IY+8)    ;BC HAS REC NO. IN BUFF

As shown:  112ØØØ  ØØ99Ø    LD    DE,2ØH      ;ADD 32 TO DCB ADDRESS
Change to: 114ØØØ  ØØ99Ø    LD    DE,4ØH      ;ADD 64 TO DCB ADDRESS
As shown:  CD4244  Ø1Ø5Ø    CALL  Ø4442H      ;POSITION
           2ØØ3    Ø1Ø6Ø    JR    NZ,J2C      ;JUMP IF DISK ERROR
           CD3644  Ø1Ø7Ø    CALL  Ø4436H      ;READ
Change to: 3E23    Ø1Ø5Ø    LD    A,35        ;LOAD DIRRD CODE
           CF      Ø1Ø6Ø    RST   8           ;DO SUPERVISOR CALL 35
           ØØ      Ø1Ø7Ø    NOP               ;  5 OPTIONAL NO-OP'S
           ØØ      Ø1Ø71    NOP               ;  TO KEEP MOD 2 VERSION
           ØØ      Ø1Ø72    NOP               ;  ALIGNED WITH
           ØØ      Ø1Ø73    NOP               ;  MOD 1 & 3 VERSIONS.
           ØØ      Ø1Ø74    NOP               ;

As shown:  C39AØA  Ø118Ø    JP    ØA9AH       ;RETURN HL TO BASIC
Change to: C37A44  Ø118Ø    JP    Ø447AH      ;RETURN HL TO BASIC
```

To the magic array listing for DFSEARCH make the following changes:

```
As shown:   32717  -69Ø2  -7715
Change to:  24Ø13  -6844  -7715

As shown:    -739   2638  18173  -69Ø1  -4681
Change to:   -739   187Ø  18173  -69Ø4  -4681

As shown:    4577     32  -3815
Change to:   4577     64  -3815

As shown:   -6659 -12847  17474    8ØØ  14Ø29  1Ø3Ø8   9742
Change to:  -6659  16Ø81 -125Ø9      Ø      Ø  1Ø24Ø   9742
```

```
As shown:  -15935 -25917   6154 -15929
Change to: -15935  31427   6212 -15929
```

The following changes need to be made to the POKE format listing for use of DFSEARCH on the TRS-80 Model II:

```
As shown:   2Ø5  127   1Ø  229
Change to:  2Ø5   93   68  229

As shown:    78   1Ø  253   7Ø   11  229
Change to:   78    7  253   7Ø    8  229

As shown:    17   32    Ø
Change to:   17   64    Ø

As shown:   2Ø9  2Ø5   66   68   32    3  2Ø5   54   68   4Ø
Change to:  2Ø9   62   35  2Ø7    Ø    Ø    Ø    Ø    Ø   4Ø

As shown:   195  154   1Ø   24
Change to:  195  122   68   24
```

Appendix 3 tells you how to give your Model II a PEEK and POKE capability. The changes listed for POKE format must be made to the data statements in DSLOADER/BAS if you wish to use it. You must also modify DSLOADER/BAS by deleting lines 40 through 42 and 110 through 122.

Error Prevention and Recovery

One of the main themes of this book is error prevention and recovery. In disk applications, error conditions can be a fact of life, and your programs should be prepared to handle them. In this chapter we'll look at some programming techniques that can help to prevent problems. We'll also look at helpful techniques for recovery if disk problems do occur.

TRS-80 Disk & Other Mysteries by H. C. Pennington is an excellent book that is devoted to recovering damaged disk files. It goes into the details of how disk directories and various types of common files are organized, and it tells you how to go in and fix them when all other methods have failed. This chapter looks at some simple "first-aid" techniques that I have used from BASIC.

Insuring Proper Disk Insertion

In applications where you are using multiple disk drives or multiple diskettes, it can be very important to verify that the operator has inserted the proper diskettes in the correct disk drives before your program opens the disk files. As you know, Disk BASIC will open a random file on whatever drive your program specifies. If you've got the wrong diskette inserted, the program will put the file name in that disk's directory. Suddenly it appears to the operator that the data has been lost. Perhaps records will be recalled and nothing but zeros show up in the data fields.

I usually solve the problem of verifying that the correct diskette is present by recording a dummy file name onto each diskette in the system. This file name serves as a diskette "label" that my program can check for, just before opening the files.

In a three-disk accounts receivable system, for example, I'll have my DOS diskette in drive 0, a diskette containing my customer file in drive 1, and a diskette with this month's transactions in drive 2. When originally setting up the system, I record the label "ARDISK1" onto the diskette that will be used in drive 1, and "ARDISK2" onto the diskette that will be used in drive 2. (I don't worry about the drive 0 diskette, because the operator couldn't start up the computer without it.)

On the external "peel-and-stick" labels, the diskettes are also labeled "ARDISK0", "ARDISK1" and "ARDISK2". (Three sets are used, labeled A, B and C to designate the backup version.)

To record a "label" onto a diskette, I simply open a sequential file in output mode, and then close it. Since I immediately close the file, it has a length of 0, and no data storage space is used. For example, to label the "ARDISK1" diskette, I insert a formatted diskette in drive 1, and then I issue the following command:

```
OPEN"O",1,"ARDISK1:1":CLOSE
```

To label the "ARDISK2" diskette, I insert a formatted diskette in drive 2, and then I type:

```
OPEN"O",1,"ARDISK2:2":CLOSE
```

Then, in programs where one or both of those diskettes are to be inserted, I simply test for the presence of the "ARDISK1" file name on drive 1 and the "ARDISK2" file name on drive 2. Upon startup of a particular accounts receivable program, the operator sees a video display similar to the one illustrated below:

```
ACCOUNTS RECEIVABLE ADDITIONS, CHANGES, AND INQUIRIES
=========================================================


THE FOLLOWING DISKETTES MUST BE INSERTED:

DRIVE Ø ----> ARDISKØ
DRIVE 1 ----> ARDISK1
DRIVE 2 ----> ARDISK2

USE SET LABELED WITH LATEST DATE -- A,B, OR C
=========================================================
PRESS <ENTER> WHEN YOU ARE READY...
```

When the operator presses the enter key, the computer displays the following message on the bottom line of the screen:

```
VERIFYING THAT DISKETTES ARE INSERTED PROPERLY...
```

If, for example, the "ARDISK2" file name is not found on the diskette in drive 2 (or if any other disk error condition is encountered), the computer displays the message:

```
CHECK THE DISK IN DRIVE 2. PRESS <ENTER> TO TRY AGAIN...
```

The operator may then correct the condition and press ENTER, and the message:

```
OPENING FILES...
```

. . . is displayed on the bottom line. From any of the locations where the operator is instructed to press <ENTER>, the up-arrow key may be pressed instead. My programs check for <up-arrow> as a signal to abort the current program, in which case it will return to the previous program or menu routine.

This diskette verification routine is located in my programs as a subroutine. Since it varies with each application, I don't consider it part of my standard random disk file handler. (I sometimes use different line numbers for the routine.)

The logic is shown below. Notice that we test for the presence of the proper label by opening the dummy file name as a sequential file in input mode. This generates an "file not found" error, which is handled in line 26020. If the dummy file name is found, the dummy file is closed and the program continues.

The diskette verification subroutine calls subroutine 40500. Line 40500 contains a one-line subroutine that is discussed in *BASIC Faster and Better & Other Mysteries.*

```
40500 A$=INKEY$:IFA$=""THEN40500ELSERETURN
```

It waits for the operator to press a single key, and returns the result of the key depression in A$.

An example of the diskette verification subroutine as it might be included in a program that does accounts receivable additions, changes and inquiries is shown below.

Diskette Verification Subroutine (Model I/III)

```
26000 CLS:PRINT"ACCOUNTS RECEIVABLE ADDITIONS, CHANGES, AND INQUIRIES
";STRING$(63,131)
26001 PRINT"THE FOLLOWING DISKETTES MUST BE INSERTED:"
26002 PRINT"
DRIVE 0 ---->   ARDISK0
DRIVE 1 ---->   ARDISK1
DRIVE 2 ---->   ARDISK2
"
26003 PRINT"USE SET LABELED WITH LATEST DATE -- A,B, OR C
";STRING$(63,131)
26005 PRINT@832,CHR$(31);"
PRESS <ENTER> WHEN YOU ARE READY...";:GOSUB40500:IFA$=CHR$(91)TH
ENRETURN

26010 PRINT@832,CHR$(31);"
VERIFYING THAT DISKETTES ARE INSERTED PROPERLY...";
26011 ONERRORGOTO26020
26012 A$="1":OPEN"I",1,"ARDISK1:1":CLOSE1
26013 A$="2":OPEN"I",1,"ARDISK2:2":CLOSE1
26019 ONERRORGOTO0:RETURN

26020 PRINT@832,CHR$(31);"
CHECK THE DISK IN DRIVE ";A$;". PRESS <ENTER> TO TRY AGAIN...";
26021 GOSUB40500:IFA$=CHR$(91)THENRESUME26019ELSERESUME26010
```

Checking a Random File

The basic idea of the diskette verification subroutine can be taken to another level. Many times it is useful to find out if a random file exists on a drive before opening it. If the file doesn't exist you may want to branch to a special creation routine, or you may want to inform the operator so that disks can be swapped if necessary.

OPEN"R" never generates a "file not found" error, so you've got to use a little trick. Before opening it in random mode, open it as a sequential file in input mode. If the file is found, you can immediately close it and reopen in random mode. If the error code indicates that the file is not found, you can RESUME to the routine that handles that condition. Lines 800 through 860 of TREESAM/BAS illustrate this technique.

Pre-Allocating Files

Every disk operating system that I've seen for the TRS-80 automatically allocates disk file space for you as you add new physical records to a disk file. This way, you don't need to know how long a file will be when you first open it. Dynamic disk file allocation is a great feature, but in real-world applications, it can also lead to problems. I've found that I can save many headaches and emergencies by by-passing the dynamic allocation feature for random files.

Instead of letting the disk operating system allocate my random files, I decide exactly how much space each file will require, and I create each file with the full length when I'm setting up the application. Here are the advantages:

- I know, in advance, exactly how many records each random file will be able to contain. There are no "disk full" surprises.
- If the files don't get closed properly, due to a power failure or operator error, nothing is lost. (If files are not pre-allocated, everything added in the current run, beyond the EOF at the time of the last close, is lost.)
- Disk operation is faster. Files are more likely to be organized in contiguous blocks on the diskette, so disk drive head movement is minimized. Also, the disk operating system doesn't need to take the time to expand the file when we do a PUT. This is all done when we pre-allocate the file.
- If there are any bad spots on the diskette, I can find them before I attempt to put valuable data on it.

Most disk operating systems, such as TRSDOS 1.3 for the Model III, have a built in "CREATE" command, which performs the pre-allocation function. With the "CREATE" library command, you can, for instance, create a file of 100 256-byte physical records, each containing 256 hexadecimal zeros.

You can do the same thing with a simple FOR-NEXT loop in BASIC, but you have the advantage of being able to specify whether the file will be filled with zeros, blanks, hexadecimal FF's, or anything else. Let's say you want to pre-allocate a file named "DATA" on drive 1, for 200 physical records, each record being pre-filled with zeros. Here are the commands you can use:

```
10 CLEAR 500                        'CLEAR SPACE FOR STRINGS
20 OPEN"R",1,"DATA:1"               'OPEN THE FILE
30 FIELD 1, 128 AS A1$, 128 AS A2$  'FIELD THE FIRST 255 BYTES
50 LSET A1$ = STRING$(128,0)        'LOAD ZEROS TO FIRST FIELD
60 LSET A2$ = STRING$(128,0)        'LOAD ZEROS TO SECOND FIELD
70 FOR X = 1 TO 200                 'FOR EACH RECORD DESIRED...
75 PRINT X                          'PRINT THE COUNT
80 PUT 1, X                         'WRITE THE BUFFER TO DISK
90 NEXT                             'REPEAT
100 CLOSE                           'CLOSE TO LOCK IN EOF.
```

Lines 700 through 733 of TREESAM/BAS give an example of how the idea of pre-allocating files may be applied in practice.

When deciding on how many physical records to pre-allocate, be aware that your disk operating system allocates disk space in "grans". On TRSDOS 2.3 for the Model I, a gran holds 5 physical records. With Model III TRSDOS, a gran contains 3 physical records. Other disk operating systems may have other gran sizes. Because files are allocated in grans, to get the most storage space, you should select file lengths that don't leave a gran partially unused. If, on your disk operating system, a gran contains 5 physical records, you should allocate your file as a multiple of 5 physical records. If your DOS stores 3 physical records per gran, use a multiple of 3.

BACKUP Procedures

The need to have backup copies of disks that contain valuable information should be obvious by now. We most often think of backup as a protection against disk drive and diskette malfunctions. In practice, it's probably more important as a protection against operator errors such as killing the wrong files, entering the wrong date or failing to close properly.

It is amazing to find that many people use BACKUP improperly. Some don't have adequate backup. Others have too many backups and soon run into problems by using obsolete copies. Over the past few years I've standardized on some techniques that have worked well. They are outlined on the next few pages.

Never label the paper diskette envelopes. It's too easy to lose them or to put the wrong diskette in an envelope. It's much more simple to leave them unlabeled. Instead, use "peel and stick" labels. For 5 1/4" diskettes I prefer "Avery #5253" labels. They are available at office supply stores.

For diskettes containing valuable programs (but not data files), keep three copies. They should be labeled "MASTER", "EXTRA" and "WORK". Never use the "MASTER" copy except to create an "EXTRA" or "WORK" copy. Use only the "WORK" copy in everyday sessions when you are running the programs. If you need to update a program, make your update on the "WORK" copy and back it up onto the "EXTRA" copy. Assuming that backup goes well, copy the "EXTRA" disk to the "MASTER" disk. Label all three disks with the current date whenever you make a change.

For diskettes that contain data files or programs and data, keep three copies. They should be labeled "A", "B" and "C". As you go from day to day, session to session, you cycle from "A" to "B" to "C" and back to "A". You make your backup after each session, and you label the diskette you duplicated onto with the current date.

As an example, let's say you have an accounts receivable system that stores data on drives 1 and 2. The diskettes have been given names for convenience. "ARDISK1" is the disk that goes in drive 1, and "ARDISK2" is the disk normally used in drive 2. In all, six data disks are used in the system, labeled "ARDISK1-A", "ARDISK1-B", "ARDISK1-C", "ARDISK2-A", "ARDISK2-B" and "ARDISK2-C". In any session, you always use the "A" disks together, or the "B" disks together, or the "C" disks together. To determine whether or not the session is to be an "A", "B" or "C" session, you look on the labels to see which pair of diskettes has the most recent date. Then you follow the "A", "B" or "C" session procedure:

"A" Session Procedures

Use "ARDISK1-A" and "ARDISK2-A".

When you've completed the session:

Backup "ARDISK1-A" to "ARDISK1-B".

Backup "ARDISK2-A" to "ARDISK2-B".

Write current date on "ARDISK1-B" and "ARDISK2-B".

"B" Session Procedures

Use "ARDISK1-B" and "ARDISK2-B".

When you've completed the session:

Backup "ARDISK1-B" to "ARDISK1-C".

Backup "ARDISK2-B" to "ARDISK2-C".

Write current date on "ARDISK1-C" and "ARDISK2-C".

"C" Session Procedures

Use "ARDISK1-C" and "ARDISK2-C"

When you've completed the session:

Backup "ARDISK1-C" to "ARDISK1-A".
Backup "ARDISK2-C" to "ARDISK2-A".
Write current date on "ARDISK1-A" and "ARDISK2-A".

Notice that you write the backup date on the diskettes you copy onto — not the source diskettes. That way, when you want to begin any session, you just look for the set: "A", "B" or "C" that has the most recent date. The dates may be written one after another on the label. You use the last date written on a label to determine the applicable date. When a label is filled you can replace it with a new one.

An important aspect of this backup procedure is that you use the diskette you last duplicated onto, not the disk you backed up from. This is because you have no foolproof way of knowing if a diskette has been duplicated successfully until you attempt to use the duplicate. If you attempt to use it and it proves faulty, you can always redo the backup. We know there are no problems on the disk you copied from in the prior session because you have already backed it up successfully.

The idea of using three copies is important. At the beginning of any session, you have two versions that are current, and one that is one session out of date. In a worst-case situation, you will need to back up from the oldest copy and redo the prior session. If problems occur in today's work, you can simply redo yesterday's backup and start today's work over.

Backup Problems

When a disk doesn't backup properly, you've got to decide whether the problem is in the "source" or the "destination." Destination problems are relatively easy to solve. You just use a different diskette or a different drive. Source problems can be more difficult. The first thing to do is try again. If you still have problems, you can try using different drives for the source and destination. The detailed BACKUP procedure presented in Chapter 2 gives you an idea of some of the things you can do.

If you've determined that the problem is on the source diskette, and you tried BACKUP a few times unsuccessfully, you need to resort to other methods. First you should consider redoing the backup from the last session and redoing today's work. If you are unwilling to accept that, you can try some other things.

Often a disk that can't be backed up with BACKUP can be duplicated with a file-by-file backup. Here's how it works:

1. You display a directory of the disk to be duplicated. If the directory can be displayed, you may continue. Otherwise, you can't use this method.
2. Write down the disk ID and the name of each file that is in the directory.
3. Now format a disk with the same disk ID shown in step 2.
4. For each file name you've written down, use COPY to transfer it from your source disk to the destination disk. If a given file can't be copied, make a note of it, kill it from the destination diskette, and go on to the next.

5. If you've copied all the files from one disk to the other you're done. It's amazing how often this works when a regular BACKUP won't.

The file-by-file copy procedure is a little different if the disk you are attempting to duplicate is a system diskette. Instead of using FORMAT, in step 3 you prepare your destination diskette by making a backup of a system diskette that you know is good. Then you kill all files that are unneeded, and you resume at step 4. If an AUTO startup was programmed onto the system diskette you're attempting to copy, you will need to put your destination disk in drive 0 after the process is complete and re-enter the AUTO command.

Copy in BASIC

Perhaps you've tried the file-by-file backup method, and you've found that a certain file won't copy. You can write a short program in BASIC that gives you more control of the copy process. Your BASIC program can give you the ability to retry or skip bad sectors. COPY1/BAS is one example of such a program:

Figure 16.1 — *COPY1/BAS*

```
Ø  'COPY1/BAS
1Ø CLEAR 1ØØØ : DEFINT A-Z
1ØØ CLS: PRINT"BASIC COPY UTILITY PROGRAM" : PRINT
11Ø LINEINPUT "SOURCE FILE NAME AND DRIVE:      ";SF$
12Ø LINEINPUT "DESTINATION FILE NAME AND DRIVE: ";DF$
13Ø PRINT"OPENING ";SF$ :OPEN"R",1,SF$
14Ø PRINT"FILE CONTAINS";LOF(1);" SECTORS"
15Ø PRINT"OPENING ";DF$ :OPEN"R",2,DF$
16Ø FIELD 1, 128ASA1$,128ASA2$
17Ø FIELD 2, 128ASB1$,128ASB2$

2ØØ R=1
21Ø IF R>LOF(1)THEN 3ØØ
22Ø PRINT"GETTING SECTOR";R
23Ø ONERRORGOTO1ØØØ: GET 1,R : ONERRORGOTOØ
24Ø LSET B1$=A1$ : LSET B2$=A2$
25Ø PRINT"WRITING SECTOR";R
26Ø ONERRORGOTO2ØØØ: PUT 2,R : ONERRORGOTOØ
27Ø R=R+1
28Ø GOTO21Ø

3ØØ ONERRORGOTOØ : CLOSE : END

1ØØØ PRINT "SECTOR";R;" ERROR CODE";ERR/2+1
1ØØ1 LINEINPUT"<R> RETRY, <S> SKIP, <F> FIX, <A> ABORT : ";A$
1ØØ2 IF A$="R"THENRESUME23Ø
1ØØ3 IF A$="S"THENLSETB1$=STRING$(128,Ø):LSETB2$=B1$:RESUME25Ø
1ØØ4 IF A$="F"THENRESUME1Ø5Ø
1ØØ5 IF A$="A"THENRESUME3ØØ
1ØØ6 GOTO1ØØØ
```

```
1Ø5Ø PRINT"YOU MAY NOW TRY SPECIAL PROCEDURES TO FIX SECTOR";R
1Ø51 PRINT"WHEN DONE, TYPE <CONT> OR <GOTO 24Ø>"
1Ø52 ONERRORGOTOØ:STOP
1Ø53 GOTO24Ø

2ØØØ PRINT "SECTOR";R;" ERROR CODE";ERR/2+1
2ØØ1 LINEINPUT"<R> RETRY, <S> SKIP, <A> ABORT : ";A$
2ØØ2 IF A$="R"THENRESUME26Ø
2ØØ3 IF A$="S"THENRESUME27Ø
2ØØ4 IF A$="A"THENRESUME3ØØ
2ØØ5 GOTO1ØØØ
```

Notice that when COPY1/BAS encounters a problem in the source file, you can enter <R> to try again with the sector that's giving you problems. Sometimes you'll be successful after a few retries. It also sometimes helps to remove and re-insert the source diskette just before a retry.

If repeated retries don't work, you can skip the bad sector. When you skip, COPY1/BAS writes a buffer full of hexadecimal zeroes to the corresponding destination file record. Then it goes on to the next source file record. If you take the skip option, it is important to be aware of the implications on the destination file. If you're copying a BASIC program that's in compressed format, the skip causes all program lines past the current sector to be lost, but at least it will be loadable. If you are copying a data file, it's likely that one or more logical records will be lost, depending on the file's organization. If you're copying a pre-allocated file that was created by the labeled files handler, perhaps nothing's been lost because the faulty sector is beyond the last active record. Anytime you skip, it is important to go back and check the file's contents after you've done the copy.

The fix option can be helpful, especially if you know how the file is organized. It breaks out the COPY1/BAS program, and you can use direct commands to fill the source buffer with data to be copied. There are several ways to do this.

One way is to simply type "GOTO 240". This takes whatever data has been read into the source buffer and copies it to the destination buffer, and continues. Many times when a disk I/O error is reported, most or all of the data in the buffer is okay. Perhaps only one bit is wrong, or the system's self-checking controls are invalid.

Another fix is to look at the source buffer by writing a temporary fielding statement. If you know the format of the buffer, you can use commands to check its contents, and perhaps you can LSET information back into it. Then you can "GOTO 240" to continue the copy.

If you've got another copy of the file, you can also try patching. Let's say, for example, that record 10 in the file you are trying to copy is bad. But you remember that you have another copy of the file in which record 10 might be okay. You simply open that file as file 3, get record 10, and LSET the data from record 10 into A1$ and A2$ of the source file buffer. Then you CLOSE 3 and goto line 240 to continue the copy.

The abort option in COPY1/BAS just closes both files and ends the copy program. Your destination file is good up to the record number where you aborted.

Invalid Directories

In rare situations, you may display a directory and see that it is obviously invalid. Perhaps the granule count adds up to more than can logically fit on a disk. In some cases you may see a file name repeated more than once in the directory. Other times you may notice that the disk ID is scrambled or missing. Fortunately these problems are not common.

The usual remedy is to format or prepare a new diskette and to use the file-by-file copy method. In the case of duplicate file names, use RENAME to rename one of them temporarily. If it's not obvious which one is the correct version of the file you can use LIST or you can write a quick BASIC program to check the contents of each. Then you can copy the correct version to the new diskette you are creating.

Recovering Files not Closed

If you've failed to close a sequential output file, its name will be in the directory but it will have a LOF and EOF of 0. When you attempt to open it for input you get an EOF indication immediately. The remedy involves a few steps:

1. Decide how many 256-byte records you think the file should contain. Estimate high.
2. Open the sequential file in random mode with OPEN"R".
3. Do a PUT to one record beyond the number you determined in step 1.
4. CLOSE the file.
5. Run the COPY2/BAS program shown below to copy as much data from the bad file as you can into a new one. You do this by repeatedly entering "C" in response to the prompt. If and when a line of bad or irrevelant data is displayed, you can press "E" to close both files and end, or you can enter "T" to re-type the data line.
6. KILL the bad file and RENAME the new one.

Figure 16.2 — *COPY2/BAS*

```
0 'COPY2/BAS
10 CLEAR 1000 : DEFINT A-Z
100 CLS : PRINT "SEQUENTIAL FILE COPY UTILITY PROGRAM " : PRINT
110 LINEINPUT"SOURCE FILE NAME AND DRIVE:      ";SF$
120 LINEINPUT"DESTINATION FILE NAME AND DRIVE: ";DF$
130 PRINT"OPENING ";SF$ :OPEN"I",1,SF$
140 PRINT"OPENING ";DF$ :OPEN"O",2,DF$
200 IF EOF(1) THEN 300
210 LINEINPUT#1,S$
220 PRINT S$
230 LINEINPUT"<C> COPY, <S> SKIP, <T> RE-TYPE, <E> END :";A$
231 IF A$="C"THEN 240
232 IF A$="S"THEN 200
233 IF A$="T"THENLINEINPUTS$:GOTO240
234 IF A$="E"THEN 300
235 GOTO 230
240 PRINT#2,S$
250 GOTO 200
300 CLOSE : END
```

For a random file that is missing information at the end because you failed to close it you can usually recover most or all of the data with the following steps:

1. Decide how many 256-byte records you think the file should contain. Estimate high.
2. OPEN the file in random mode.
3. Do a PUT to one record beyond the number you determined in step 1.
4. CLOSE the file.

Neither of these failure-to-close recovery techniques is sure-fire, but they will usually work if you catch the problem right after it occurs. Your chances for success are almost 100 percent if there is only the one file on the disk. If there are other files on the disk and some of them were created or extended in length since your failure to close, the data in your problem file might not be recoverable. It's also important to perform these fixes on the diskette where the problem occurred. Some versions of BACKUP use the LOF and EOF values in the directory to determine what data is copied, so the data to be recovered might not be present on a backup diskette.

Jumbled Programs

BASIC stores programs in memory in a compressed format with pointers that connect one program line to the next. It's possible for these pointers to get mixed up. Sometimes the cause is a temporary power fluctuation. Other times a disk problem changes some of the data. Perhaps a program has erroneously opened a program file and put irrelevant data into it. Maybe you've been experimenting with the POKE statement, and you've caused the problem.

The symptoms are line numbers that appear more than once when you do a listing, endless loops of line numbers, lines that you are unable to list or delete individually, and unusual graphics characters or other "mumbo jumbo" in your program listing. To solve the problem, your best bet is to go to a backup copy of your program. If you don't have one, here's what you do:

1. Save the program in ASCII, using the "A" option.
2. Reload the program.
3. Now you can edit, add program lines, and delete as required.

The load from an ASCII program file automatically puts the line numbers back into sequence and deletes duplicates. It restores the integrity of the internal pointers.

An optional improvement on this recovery technique is to use COPY2/BAS after step 1. This way you can do deletions and some editing before the program is reloaded.

Direct Statement in File

If you are unable to load a program that has been saved in ASCII format because of a "direct statement in file" error, you can fix it with a short program that inputs the program as a sequential file and outputs it as another sequential file, modified to be loadable. The following program does the job:

```
1Ø CLEAR 1ØØØ
2Ø LINEINPUT"FILE NAME AND DRIVE OF PROGRAM TO BE FIXED: ";SF$
3Ø LINEINPUT"FILE NAME AND DRIVE FOR CORRECTED VERSION: ";DF$
```

```
40 PRINT"OPENING ";SF$ : OPEN"I",1,SF$
50 PRINT"OPENING ";DF$ : OPEN"O",2,DF$
60 IF EOF(1) THEN CLOSE : END
70 LINE INPUT#1,A$      'GET A LINE FROM UNLOADABLE PROGRAM
80 IF A$<"0"ORA$>"65529" THEN 60  'SKIP IF BAD LINE NUMBER
90 IF LEN(A$)>240 THEN A$=LEFT$(A$,240)  'FORCE VALID LENGTH
100 PRINT#2,A$          'OUTPUT GOOD LINE TO DESTINATION FILE
110 GOTO 60             'REPEAT FOR NEXT LINE IN SOURCE FILE
```

The Final Analysis

In the final analysis, your success with disk files depends on how well you understand them. As your data files are expanded and updated from day to day, you are accumulating quite an investment of work and time. If you are involved in writing programs, I hope this book has helped you to see new ways to make that investment pay off in faster and better programs. If you are are mainly a user of programs written by others, I hope this book has increased your understanding so that you can protect your investment of work and time.

APPENDIX 1

Finding Disk Buffer and DCB Addresses

The memory addresses for the disk buffers and the disk Data Control Blocks are sometimes necessary when you want to do PEEKs and POKEs to achieve special effects. The DCB addresses are needed when you use machine language USR subroutines that access BASIC disk files.

Unfortunately, the addresses vary, depending on the disk operating system you are using, the revision number of the DOS, and whether or not you are using variable length files. Because so many companies are selling disk operating systems for the TRS-80, it is no longer possible in a book like this to list all of them and the addresses they use. Instead, I've devised a simple step-by-step procedure so that you can get the addresses you need. I can't guarantee that this procedure will work for all TRS-80 disk operating systems, but it has worked for all that I tested.

The steps in the procedure are shown below. For the sample data, the illustrations show the information that is gotten for TRSDOS 1.3 on a TRS-80 Model III.

1. Get out a sheet of paper. On it write "A=", "B=" and "C=" on three lines.
2. Go to DOS Ready mode.
3. Start up BASIC, specifying 1 file. No special entry is required for memory size.

   With TRSDOS 1.3 on the Model III, your display shows:

```
How Many Files? 1
Memory Size?
TRS-8Ø Model III Disk BASIC Rev 1.3
(c)(p) 198Ø by Tandy Corp. All Rights Reserved.
Created 5-July-8Ø
38,922 Free Bytes   1 Files
Ready
>
```

4. Type the following:

```
A$=CHR$(PEEK(&H4ØA4))+CHR$(PEEK(&H4ØA5))
PRINT CVI(A$)
```

Note: Replace **40A4** and **40A5** with **2B4F** and **2B50** with MOD II TRSDOS 2.0.

5. Write down the answer you get next to "A=" on your sheet of paper.

For TRS-80 Model III Disk BASIC, the number is 26541.

6. Go back to DOS Ready by entering CMD "S", or by pressing the reset button.

7. Startup BASIC again, this time specifying 2 files.

8. Enter the same two commands shown in step 4.

9. Write down the answer you get next to "B=" on your sheet of paper.

For TRS-80 Model III Disk BASIC, the number is 26901.

10. Open a random file in file buffer 2. Use a file name that you are sure doesn't already exist. Here's how your command might look:

```
OPEN"R",2,"DUMMY:Ø"
```

11.Enter the following three commands:

```
FIELD2,1ASA$
B$=CHR$(PEEK(VARPTR(A$)+1))+CHR$(PEEK(VARPTR(A$)+2))
PRINT CVI(B$)
```

12. Write down the number displayed next to "C=" on your piece of paper.

For TRS-80 Model III Disk BASIC the number is 26592.

13. You may close and kill the file you opened in step 10. Your command is:

```
CLOSE : KILL"DUMMY:Ø"
```

14. Now you have three numbers next to A, B and C on your sheet of paper. Go ahead and type these into your computer.

For TRS-80 Model III Disk BASIC we enter:

```
A=26541
B=269Ø1
C=26592
```

15. To find how many bytes each file specified upon entering BASIC requires, enter the following command:

```
PRINT B-A
```

For TRS-80 Model III Disk BASIC, 360 bytes are required per file.

16. To find the address of the DCB for file 1, enter:

```
PRINT (A+1) - (B-A)
```

TRS-80 Model III Disk BASIC returns 26182.

17. To find the address of the buffer for file 1, enter:

```
PRINT C - (B-A)
```

TRS-80 Model III Disk BASIC returns 26232.

18. To find out how much space is used for each DCB, enter:

```
PRINT C - (A+1)
```

TRS-80 Model III Disk BASIC uses a 50 byte Data Control Block for each file. The buffer for each file starts 50 bytes beyond the beginning of that file's DCB.

19. To get a list of the DCB and file buffer addresses by file number, enter and run the following:

```
1Ø PRINT "FILE","DCB ADDR","BUF ADDR";
2Ø FOR X=1TO15 : PRINT
3Ø PRINTX,26182+(X-1)*36Ø,26232+(X-1)*36Ø;
4Ø NEXT
5Ø GOTO 5Ø
```

Replace "26182" with the number you got in step 16. Replace "26232" with the number you got in step 17. Replace the 360's with the number you got in step 15.

Note: If any of the numbers displayed are greater than 32767, subtract 65536. This subtraction will give a negative number that you can use use in PEEK or POKE statements or when loading integer variables.

20. If you will be using variable record length files, the numbers will be different. To find out what they are, repeat steps 1 through 19, but use the "V" option in steps 3 and 7 when you specify "How Many Files?".

Data Control Block Addresses

	TRSDOS 2.3	TRSDOS 1.3	TRSDOS 1.3	TRSDOS 2.Ø
	MODEL I	MODEL III	MODEL III	MODEL II
File 1	263Ø3	26182	26438	27715
File 2	26593	26542	27Ø54	28549
File 3	26883	269Ø2	2767Ø	29383
File 4	27173	27262	28286	3Ø217
File 5	27463	27622	289Ø2	31Ø51
File 6	27753	27982	29518	31885
File 7	28Ø43	28342	3Ø134	32719
File 8	28333	287Ø2	3Ø75Ø	-31983
File 9	28623	29Ø62	31366	-31149
File 1Ø	28913	29422	31982	-30315
File 11	292Ø3	29782	32598	-29481
File 12	29493	3Ø142	-32322	-28647
File 13	29783	3Ø5Ø2	-317Ø6	-27813
File 14	3ØØ73	3Ø862	-31Ø9Ø	-26979
File 15	3Ø363	31222	-30474	-26145
DCB Length:	32	5Ø	5Ø	64
Bytes Per File:	29Ø	36Ø	616	834

Notes:

To get buffer addresses, add DCB length to DCB Address.

Two columns are shown for Model III TRSDOS 1.3. The second column shows the DCB addresses when you enter BASIC with the "V" option. The "V" option is not applicable to Model I TRSDOS 2.3. Model II TRSDOS 2.0 always enters BASIC with an automatic variable length record capability, so it has no "V" option.

Appendix 2

The DELARRAY USR Subroutine

In Chapter 4 we talked about using sequential files as a substitute for data statements. In doing so, the example given was a 40-byte USR subroutine, DELARRAY. The delete array USR routine lets your BASIC program delete any array from memory and recover the space it occupied. It can be a very valuable addition to your library, so it is provided here as a little "bonus" program.

DELARRAY works with integer, single precision, double precision or string arrays of any size. An optional modification lets you delete double or triple dimensioned arrays. Once an array has been deleted from memory, all the data it contained is gone, but you are free to redimension the array if you wish so that new information can be put into it.

DELARRAY is a relocatable Z-80 USR subroutine, 40 bytes long. As such, you can load it or POKE it into any area of protected memory. Like all relocatable USR subroutines there are many ways to get it into memory. *BASIC Faster and Better & Other Mysteries* discusses several of them.

To call DELARRAY, you must have first done a DEFUSR to tell BASIC its address in memory. Your calling argument is the VARPTR of the 0 element of the array you want to delete.

DELARRAY/DEM is a demonstration that creates three arrays and then deletes them. As it does so, it displays the current memory available so that you can see what's happening.

Figure A2.1 — *DELARRAY/DEM*

```
Ø 'DELARRAY/DEM
1 CLEAR 1ØØ : DEFINTA-Z : J=Ø
1Ø BA=&HFFØØ:          'DELARRAY ADDRESS, MODIFY AS YOU WISH
11 BA!=ASC(MKI$(BA))+ASC(MID$(MKI$(BA),2))*256
2Ø IFPEEK(16561)+PEEK(16562)*256+1 > BA! THEN PRINT "PLEASE RE-E
NTER BASIC WITH MEMORY SIZE OF NO MORE THAN";BA! : END
3Ø FOR X=Ø TO 39 : READ A : POKE BA+X,A : NEXT
4Ø DEFUSRØ=BA          'DEFINE USR Ø AS DELARRAY AT FFØØ
1ØØ CLS:PRINT"ARRAY DELETING DEMONSTRATION"
11Ø PRINT"MEMORY FREE IS";MEM
```

```
12Ø PRINT"DIMENSIONING INTEGER ARRAY FOR 1ØØ ELEMENTS"
121 DIM I%(99)
122 PRINT"MEMORY FREE IS";MEM
13Ø PRINT"DIMENSIONING DOUBLE PRECISION ARRAY FOR 5Ø ELEMENTS"
131 DIM D#(49)
132 PRINT"MEMORY FREE IS";MEM
14Ø PRINT"DIMENSIONING A STRING ARRAY FOR 9ØØ ELEMENTS"
141 DIM S$(899)
142 PRINT"MEMORY FREE IS";MEM
2ØØ PRINT"DELETING THE STRING ARRAY"
2Ø1 J=USRØ(VARPTR(S$(Ø)))
2Ø2 PRINT"MEMORY FREE IS";MEM
21Ø PRINT"DELETING THE DOUBLE PRECISION ARRAY"
211 J=USRØ(VARPTR(D#(Ø)))
212 PRINT"MEMORY FREE IS";MEM
22Ø PRINT"DELETING THE INTEGER ARRAY"
221 J=USRØ(VARPTR(I%(Ø)))
222 PRINT"MEMORY FREE IS";MEM
3ØØ END
1ØØØ 'DATA STATEMENTS FOR THE DELETE-ARRAY USR SUBROUTINE
1ØØ1 DATA 2Ø5,127,1Ø,17,4,Ø,183,237,82,229,7Ø,43,78,43,43,43
1ØØ2 DATA 227,9,35,229,235,42,253,64,183,237,82,229,193,225
1ØØ3 DATA 2Ø9,4Ø,2,237,176,235,34,253,64,2Ø1
```

DELARRAY/DEM puts the DELARRAY routine at address **FF00**. That's –256 in integer notation, or 65280 in decimal. To run the program as shown, you will need to specify a memory size of 65280 or less upon entering BASIC. You can change the address by changing line 10. Notice that the program verifies that your memory setting is appropriate before it attempts to execute DELARRAY.

Lines 201, 211, and 221 show how DELARRAY is called. In each case, J is used as a dummy variable. Its contents after each call are unimportant to us since DELARRAY doesn't return any information to BASIC. You can use any integer variable as the dummy. Notice though, that you must initialize the variable before your first call to DELARRAY. This was done in line 1 when the program set J% equal to 0. Failure to do this would cause DELARRAY to fail, because any time you use a new simple variable in a program the addresses of all arrays change.

DELARRAY does no checking before it attempts to delete the array you specify, so it is very important that you set up the call correctly. You must use the VARPTR of the 0 element of the array as your argument or the results will be dangerously unpredictable.

You can change the 5th byte of the DELARRAY routine from 4 to 6 if you want to delete a double dimension array. To delete a triple dimension array, change it to 8. If you have a combination of single, double, and/or triple dimension arrays that are to be deleted in the same program, you can make the modification to DELARRAY by poking the appropriate address with 4, 6 or 8 just before your call.

DELARRAY can also be loaded and executed as a "magic array." This technique is discussed in *BASIC Faster and Better & Other Mysteries*. You can specify the dimension as single, double, or triple by changing the 3rd element to 4, 6 or 8. If you're storing DELARRAY in a magic array, be aware that it can't delete itself! An attempt to do so will cause havoc with the contents of memory.

As shown, DELARRAY/DEM is POKEd into memory from data statements. If you want to try the technique shown in Chapter 4, you can run DATAOUT/DEM. Then you can delete lines 1000 through 1003 in DELARRAY/DEM and change line 30 to:

```
3Ø OPEN"I",1,"DATAFILE:Ø":FORX=ØTØ39:INPUT#1,A:POKEBA+X,A:NEXT:CLOSE
```

You may also want to try the faster idea of storing the 40 bytes of DELARRAY in a random file buffer. To execute it you can simply get the record containing DELARRAY. You do your DEFUSR so that it points to the bufffer. This technique doesn't require you to reserve memory upon entering BASIC.

Magic Array Format, 20 Elements

```
32717   4362      4  -4681  -683Ø  117Ø8  1Ø86  11Ø51   2531
-6877  1Ø987  16637  -4681  -683Ø  -7743  1Ø449  -4862  -52ØØ
 -734 -14Ø16
```

Poke Format, 40 Bytes

```
2Ø5 127  1Ø  17   4   Ø 183 237  82 229  7Ø  43  78  43  43 43
227   9  35 229 235  42 253  64 183 237  82 229 193 225 2Ø9 4Ø
  2 237 176 235  34 253  64 2Ø1
```

Figure A2.1 — *DELARRAY Assembly Listing*

```
              ØØØØØ ;DELARRAY
              ØØØØ1 ;
FFØØ          ØØØ1Ø        ORG    ØFFØØH        ;ORIGIN (RELOCATABLE)
FFØØ CD7FØA   ØØØ2Ø        CALL   ØA7FH         ;HL HAS ELEMENT Ø VARPTR
FFØ3 11Ø4ØØ   ØØØ3Ø        LD     DE,ØØØ4H      ;GO BACK 4 BYTES TO GET SIZE
              ØØØ4Ø ;NOTE: CHANGE THE ØØØ4H TO ØØØ6H FOR DOUBLE DIMENSION
              ØØØ5Ø ;      CHANGE THE ØØØ4H TO ØØØ8H FOR TRIPLE DIMENSION
FFØ6 B7       ØØØ6Ø        OR     A             ;CLEAR CARRY
FFØ7 ED52     ØØØ7Ø        SBC    HL,DE         ;HL POINTS TO ARRAY SIZE
FFØ9 E5       ØØØ8Ø        PUSH   HL            ;SAVE IT
FFØA 46       ØØØ9Ø        LD     B,(HL)        ;
FFØB 2B       ØØ1ØØ        DEC    HL            ;
FFØC 4E       ØØ11Ø        LD     C,(HL)        ;BC HAS ARRAY SIZE - 5
FFØD 2B       ØØ12Ø        DEC    HL            ;
FFØE 2B       ØØ13Ø        DEC    HL            ;
FFØF 2B       ØØ14Ø        DEC    HL            ;HL POINTS TO ARRAY BASE
FF1Ø E3       ØØ15Ø        EX     (SP),HL       ;HL POINTS TO ELEMENT Ø
FF11 Ø9       ØØ16Ø        ADD    HL,BC         ;ADD OFFSET
FF12 23       ØØ17Ø        INC    HL            ;
FF13 E5       ØØ18Ø        PUSH   HL            ;SAVE ON STACK
FF14 EB       ØØ19Ø        EX     DE,HL         ;DE => NEXT ARRAY
FF15 2AFD4Ø   ØØ2ØØ        LD     HL,(4ØFDH)    ;HL => START OF FREE SPACE
FF18 B7       ØØ21Ø        OR     A             ;CLEAR CARRY
```

```
FF19 ED52     ØØ22Ø        SBC    HL,DE          ;HL = HOW MANY BYTES TO MOVE
FF1B E5       ØØ23Ø        PUSH   HL             ;
FF1C C1       ØØ24Ø        POP    BC             ;BC = HOW MANY BYTES TO MOVE
FF1D E1       ØØ25Ø        POP    HL             ;HL HAS "FROM" ADDRESS
FF1E D1       ØØ26Ø        POP    DE             ;DE HAS "TO" ADDRESS
FF1F 28Ø2     ØØ27Ø        JR     Z,NOMOV        ;NO MOVE IF BC IS ZERO
FF21 EDBØ     ØØ28Ø        LDIR                  ;MOVE REMAINING ARRAYS DOWN
FF23 EB       ØØ29Ø NOMOV  EX     DE,HL          ;HL HAS NEW POINTER
FF24 22FD4Ø   ØØ3ØØ        LD     (4ØFDH),HL     ;LOAD NEW FREE SPACE POINTER
FF27 C9       ØØ31Ø        RET                   ;RETURN TO BASIC
              ØØ32Ø        END
```

DELARRAY on the Model II

Model II BASIC provides an ERASE command that performs the same function, so as a practical matter, there is no need for DELARRAY. If you'd still like to try DELARRAY, you'll need to see Appendix 3 for information about how to install PEEK and POKE. Then you can replace "205,127,10" in the POKE format listing with "205,93,68". This causes a call to **445D**, rather than **A7F**, to get the argument from BASIC. Replace both references to "253,64" with "8,45". This change is necessary because the free space pointer on the Model II is at **2D08** rather than **40FD**. Delete lines 11 and 20 from DELARRAY/DEM.

The same Model II changes may be made to the magic array listing for DELARRAY. Change "32717 4362" to "24013 4420". Change "16637" to "11528". Change "–734 – 14016" to "2082 –14035".

Appendix 3

Most of the concepts and techniques discussed in this book are directly applicable to the TRS-80 Models II, 12, and 16 when operated with TRSDOS 2.0 or 2.0a. We'll discuss the main exceptions in this section.

TRS-80 Model II Error Code Translation

If you've got a TRS-80 Model II, you will need to translate the error code numbers from the corresponding Model I and III codes shown in this book. Note that Model II error codes are returned by "ERR", rather than "ERR/2+1", and the error descriptions are replaced by two-letter codes.

Model I & III		Model II	
ERR/2+1		ERR	
51	Field overflow	5Ø	FO
52	Internal error	51	IE
53	Bad file number	52	BN
54	File not found	53	FF
55	Bad file mode	54	BM
56	File already open	55	AO
58	Disk I/O error	56	IO
62	Disk full	59	DF
63	Input past end	6Ø	EF
64	Bad record number	61	RN
65	Bad file name	62	NM
67	Direct statement	65	DS
68	Too many files	66	FL
69	Disk write-protect	56	IO
7Ø	File access denied	56	IO

PEEK and POKE for the Model II

Some of the programs require a PEEK and POKE capability. This subject is discussed more fully in *BASIC Faster and Better & Other Mysteries*. To give BASIC under TRSDOS 2.0 or 2.0a these two additional commands, you can run POKEMOD/BAS:

Figure A3.1 — *POKEMOD/BAS*

```
Ø 'POKEMOD/BAS
1Ø DEFINTA-Z
2Ø DIMUS(46)
3Ø FORX=ØTO46:READUS(X):NEXT
4Ø J=Ø:DEFUSR=VARPTR(US(Ø)):J=USR(Ø)
5Ø END
8Ø DATA -13Ø23,8925,26611,15393,8917,26613,-6367,8748,26615
81 DATA -13Ø23,8938,26617,15393,8913,26619,4641,89Ø5,26621
82 DATA -13Ø23,8797,26623,17441,883Ø,26625,-15583,8955,26627
83 DATA 1491Ø,133Ø,8552,2Ø432,-1246,15912,12875,1Ø493,-12255
84 DATA 8773,1Ø757,17697,8779,1Ø759,-15583,8959,23259,2643Ø
85 DATA -891Ø,-1399Ø
```

POKEMOD/BAS gives your Model II a temporary PEEK and POKE capability that lasts as long as you are in BASIC. It should be run at the beginning of any BASIC session where PEEK or POKE will be required.

High Memory on the Model II

There are some special considerations that may be applicable if you wish to wish to use KEYACCESS, DFSEARCH, or other routines that are to be POKEd or loaded into high memory. Model II TRSDOS reserves an area of high memory for certain functions, namely DO, DEBUG, SETCOM, SPOOL and HOST.

If none of these functions are active, the addresses used for the USR subroutine examples in this book will normally work properly. If you wish to use any of the functions while a USR subroutine is in memory, you will need to take that into consideration when you decide on the address to use. The TRSDOS library command, STATUS tells you which of these functions, if any is active. Here's a sample display:

```
STATUS
HIGH MEMORY ADDRESS =FØØØ hex / 6144Ø dec.
 ON:
OFF:   DO      DEBUG   SETCOM  SPOOL    HOST
```

In this case none of the special functions is ON. The display shows that the high memory address is 61440. Upon entering BASIC there is no need to reserve memory if all your USR routines are to load above 61440.

To be safe in any case, you can reserve memory below the high memory address. Let's say, for example, that you want the 629-byte KEYACCES USR subroutine. You can subtract 629 from 61440 to get 60811. Then you can enter BASIC with a command like:

```
BASIC -F:2 -M:6Ø811
```

This would reserve 2 file buffers and protect memory from 60811 to 61440 for KEYACCES. Your DEFUSR address can be determined with:

```
PRINT HEX$(6Ø811)
```

That gives **ED8B**. To convert **ED8B** to integer format you can enter:

```
PRINT &HED8B
```

That gives –4725. Your DEFUSR statement could be:

```
DEFUSR8 = &HED8B
```

or

```
DEFUSR8 = -4725
```

Alternatively, you can use the CLEAR statement while in BASIC. To clear memory from 60811 to 61440, as well as 1000 bytes for string storage, you can say:

```
CLEAR 1ØØØ,6Ø811
```

Appendix 4

The DFBLOAD Diskette

Typing in the programs and examples from this book can be quite a time-consuming and error-prone process. If you wish, you can purchase the DFBLOAD diskette which contains all the major subroutines and programs from this book. The following files are included:

```
MPGCALC1/BAS      DATASUB/DEM       MPGCALCA2/DEM
DATAOUT/DEM       COPYSEQ/BAS       RANDOM1/DEM
RANDOM2/DEM       RANDOM3/DEM       RANDOM4/DEM
RANDOM5/DEM       RANDOM6/DEM       RANDOM7/DEM
RANDOM8/DEM       LFH/DEM           DFH/DEM
RBA/DEM           MIH/DEM           TREESAM/BAS
KALOADER/BAS      KEYACCES/DEM      DSLOADER/BAS
DFSEARCH/DEM      COPY1/BAS         COPY2/BAS
DELARRAY/DEM
```

Note: Other files may be provided.

All the files are BASIC programs. Those having the /DEM extension are primarily for demonstration purposes, but you can consider them "library" files. When you want to implement a subroutine that a /DEM program contains, you can delete unwanted lines, save in ASCII, and merge the wanted subroutines into other programs. That is, they serve a purpose beyond demonstrating concepts covered in this book.

MPGCALC1/DEM

This program from Chapter 3 is a simple demonstration of a mileage calculator. In the context of this book it is used to show techniques of writing, saving and loading a BASIC program. It also shows the concepts of program storage on disk and in memory, and the LIST capability from DOS READY. MPGCALC1/ BAS does not create or use disk files. No particular memory size or files setting is required upon entering BASIC.

Memory size:	No special setting.
Files setting:	No special setting.
Files used:	None
Files created:	None
Comments:	Self prompting.

DATASUB/DEM

This program from Chapter 3 demonstrates one use of data statements and error traping in a simple account number lookup problem. The read-data subroutine may be extracted from lines 900 – 904 for use in other programs.

Memory size:	No special setting.
Files setting:	No special setting.
Files used:	None
Files created:	None
Required free:	Ø grans.
Comments:	Self prompting.

MPGCALC2/DEM

This program from Chapter 4 is a modification of MPGCALC1/DEM. It demonstrates a simple application for saving the contents of BASIC variables in a sequential disk file and loading them back.

Memory size:	No special setting.
Files setting:	Specify at least 1 file.
Files used:	MPGCALC2/DAT:Ø (Optional)
Files created:	MPGCALC2/DAT:Ø (Optional)
Required free:	1 gran on drive Ø,
Comments:	Self prompting.

You may Kill "MPGCALC2/DAT" after the demonstration to recover the free space on your diskette.

DATAOUT/DEM

This program demonstrates the idea of building a sequential file from information contained in data statements. For the demonstration it outputs a list of 40 numbers into a sequential file. DATAOUT/DEM may be useful as a utility program if you change the data statements and output file name to suit your own special purposes.

Memory size:	No special setting.
Files setting:	Specify at least 1 file.
Files used:	None
Required created:	DATAFILE:Ø
Required free:	1 gran on drive Ø.
Comments:	No display, See discussion, chapter 4.

If you wish, you may use the DOS library command, LIST, to view the file created. You may also want to experiment with the input of the data contained in DATAFILE:0 as a way for loading numbers to be POKEd in DELARRAY/DEM. "DATAFILE:0" may be KILLed after the demonstration to recover free space on the diskette used.

COPYSEQ/BAS

This program from Chapter 4 demonstrates the idea of copying from one sequential file to another. The program may be used for that simple purpose, or it may be expanded for more complex applications as discussed in Chapter 4.

Memory size:	No special setting.
Files setting:	Specify at least 2 files.
Files used:	Any existing sequential file.
Files created:	You specify name of file to be created.
Required free:	Depends on files used.
Comments:	Self prompting.

COPYSEQ/BAS lets you supply the file names and drive numbers for the source and destination files. Since the primary purpose of this program is to demonstrate sequential file copying, no error trapping is done, other than end of file checking. The source file must already exist on the drive you specify, or the program will abort with a "file not found" error. Unless you specify an output file name that already exists, you must have free space equal to the size of the source file, or the program will abort with a "disk full" error.

One way to test COPYSEQ/BAS is to save a BASIC program with the "A" option. Then you can use it as your source file. After the copy is complete you can load the same program using the destination file name and drive that you specified.

If you are using COPYSEQ/BAS just to test and demonstrate the concept of copying sequential files, you will want to kill the file you specified as your destination file after you've completed the demonstration.

RANDOM1/DEM

This program from Chapter 5 demonstrates the use of the LOF function with a random file. To use RANDOM1/DEM you must follow the discussions and exercises that precede it in Chapter 5 so that "TESTFILE:0" will be present.

Memory size:	No special setting.
Files setting:	Specify at least 1 file.
Files used:	TESTFILE:Ø
Files created:	TESTFILE:Ø (If not already present.)
Required free:	Ø grans, assuming TESTFILE:Ø exists.
Comments:	No prompting. See Chapter 5.

You should not kill "TESTFILE:0" if you want to continue on to RANDOM2/DEM.

RANDOM2/DEM

This program demonstrates the use of the EOF function and the simplified GET statement. It is intended to be used as part of the tutorial in Chapter 5, and it assumes that "TESTFILE:0" is present.

Memory size:	No special setting.
Files setting:	Specify at least 1 file.
Files used:	TESTFILE:Ø
Files created:	TESTFILE:Ø (If not already present.)
Required free:	Ø grans, assuming TESTFILE:Ø exists.
Comments:	No prompting. See Chapter 5.

You should not kill "TESTFILE:0" if you want to continue on to RANDOM3/DEM.

RANDOM3/DEM

RANDOM3/DEM uses the STR$ function to put zeros into two new numeric fields in "TESTFILE:0". This program assumes you've been following the examples in Chapter 5, and it prepares the file so that RANDOM3/DEM can be run. Nothing is displayed on the screen during operation of RANDOM3/DEM, so you'll need to refer to the text and the remarks within the program to see exactly what it is doing.

Memory size:	No special setting.
Files setting:	Specify at least 1 file.
Files used:	TESTFILE:Ø
Files created:	TESTFILE:Ø (If not already present.)
Required free:	Ø grans, assuming TESTFILE:Ø exists.
Comments:	No prompting. See Chapter 5.

You should not kill "TESTFILE:0" if you want to continue on to RANDOM4/DEM.

RANDOM4/DEM

This program continues the training session on random files. It shows techniques of random access that allow you to update the contents of fields in random file records. It uses "TESTFILE:0" as created according to the instructions in Chapter 4. "RANDOM3/DEM" should be run before this program to clear the numeric fields.

Memory size:	No special setting.
Files setting:	Specify at least 1 file.
Files used:	TESTFILE:Ø
Files created:	TESTFILE:Ø (If not already present.)
Required free:	Ø grans, assuming TESTFILE:Ø exists.
Comments:	Self prompting. See Chapter 5.

RANDOM5/DEM and RANDOM6/DEM

These two programs are like RANDOM3/DEM and RANDOM4/DEM, but they show compressed numeric storage techniques. The prompts and data on the video display will be the same, so the main value of these programs is seeing the differences in the way they are coded. Like the others in this series, they assume that "TESTFILE:0" exists, "RANDOM5/DEM" must be run before "RANDOM6/DEM" to recreate the numeric fields in the proper format.

You may kill "TESTFILE:0" after completing the exercises from "RANDOM1/DEM" to "RANDOM6/DEM". Optionally, you may save the data you've created for use in "RANDOM7/DEM".

RANDOM7/DEM

This program shows how you can make additions, changes and inquiries to a random file. The file format from "RANDOM5/DEM" and "RANDOM6/DEM" is used so you can continue with "TESTFILE:0". Alternatively, you can kill "TESTFILE:0" before starting "RANDOM7/DEM" because it has all the logic required to create it.

Memory size:	No special setting.
Files setting:	Specify at least 1 file.
Files used:	TESTFILE:Ø
Files created:	TESTFILE:Ø (If not already present.)
Required free:	Model I,II - 1 gran for every 5 records.
	Model III - 1 gran for every 3 records.
Comments:	Self prompting.

Notice that "RANDOM7/DEM" uses 256-byte records. The number of records that you will be able to add to the file depends on how much free space is available on drive 0, and the number of 256-byte records per gran that your disk operating system allows. A "disk full" error will result if you run out of space.

You may kill "TESTFILE:0" after this demonstration program. All subsequent demonstrations in Chapter 5 create and use "TESTFILE:0" under a different and incompatible format.

RANDOM8/DEM

This program is a modification of "RANDOM7/DEM" that uses blocking techniques to provide more efficient data storage. Three 85-byte logical records are stored in each 256-byte physical record. See Chapter 5 for the complete discussion.

You should kill "TESTFILE:0" if it exists before running "RANDOM7/DEM" if you've been using any of the other random file demonstration programs in the series.

Memory size:	No special setting.
Files setting:	Specify at least 1 file.
Files used:	TESTFILE:Ø (Optional.)
Files created:	TESTFILE:Ø (If not already present.)
Required free:	Model I,II - 1 gran for every 15 records.
	Model III - gran for every 9 records.
Comments:	Self prompting.

You may kill "TESTFILE:0" after the demonstration.

LFH/DEM

This program demonstrates the labeled files handler from Chapter 7, and the random files handler from Chapter 6. Additions, changes, inquiries, and deletions are demostrated on a simple file that contains names and phone numbers. A detailed description of the operation of this program is given in Chapter 7.

The program allows entry of the filespec, so the name and drive of the file to be used is up to you. "TESTFILE:0" is used in the examples. You must take the option to initialize the file the first time you run this program, or any time you run the program after the same file has been used for a different demonstration.

Memory size:	No special setting.
Files setting:	Specify at least 1 file.
Files used:	TESTFILE:Ø (Optional)
Files created:	TESTFILE:Ø (If not already present.)
Required free:	Assuming the file doesn't already exist,
	Model I,II - 1 gran for every 35 records.
	Model III - 1 gran for every 21 records.
	Self prompting. See Chapter 7.

You may kill the file you created after the demonstration.

DFH/DEM

DFH/DEM demonstrates detail file techniques and concepts. It extends the LFH/DEM program to allow the recording of an unlimited number of comments that are to be related to each name and phone number. Two files are used. One is considered a "master" file and the other is a "detail" file. When you start the program, you can select the names and drives to be used. It's suggested that you follow the sample data entries in Chapter 8. You may use "TESTFILE:0" for the master file and "DETAIL:0" for the detail file. You must take the option to initialize the files the first time you run this program, or any time you run the program after the same files have been used for a different demonstration.

Memory size:	No special setting.
Files setting:	Specify at least 3 files.
Files used:	TESTFILE:Ø (Optional)
	DETAIL:Ø (Optional)
Files created:	TESTFILE:Ø (If not already present.)
	DETAIL:Ø (If not already present.)
Required free:	Assuming the files don't already exist.
	Master file:
	Model I,II - 1 gran for every 3Ø records.
	Model III - 1 gran for every 18 records.
	Detail file:
	Model I,II - 1 gran for every 4Ø records.
	Model III - 1 gran for every 24 records.
Comments:	Self prompting. See Chapter 8.

You may kill the files you created after the demonstration.

RBA/DEM

This program from Chapter 9 uses random byte addressing and sector spanning. It demonstrates the string-get and string-put subroutines. You can specify the file name and drive. Perhaps the best way to test it is to make a copy of another file and call it "TESTFILE:0". Then try the <P> and <G> commands to "put" and "get" strings at various positions in the file.

Memory Size:	No special setting.
Files setting:	Specify at least 1 file.
Files used:	TESTFILE:Ø (Optional.)
Files created:	Depends on file name specified.
Required free:	Depends on byte positions you select.
Comments:	Limited self prompting. See Chapter 9.

You may kill the file you use after the demonstration.

MIH/DEM

The Memory Index Handler from Chapter 10 is demonstrated by this program. Names and phone numbers are stored on disk and you can access them by short keys of up to 6 characters. For the first run you will want to take both "Initialize" options. Detailed instructions are given in Chapter 10. You may wish to use drive 0 for your files rather than drive 1 as shown.

Memory Size:	No special setting.
Files setting:	Specify at least 2 files.
Files used:	TESTFILE:1 (Optional.)
	TESTINDX:1 (Optional.)
Files created:	Depends on file names specified.
Required free:	Assuming the files don't already exist:
	Model I,II - 2 grans + 1 for every 35 records.
	Model III - 4 grans + 1 for every 21 records.
Comments:	Self prompting. See Chapter 1Ø.

TREESAM/BAS

This is the TREESAM utility and training program from Chapter 13, where instructions and a sample session are provided. It includes the standard TREESAM subroutines from Chapter 12. The file names are up to you. Be sure to take the "Create" option the first time you run it. Once you've implemented TREESAM in your own programs, you will probably want to keep TREESAM/BAS around as a "tool-kit" program for creation, diagnostics, changes and repairs.

Memory Size:	No special setting.
Files setting:	Specify at least 2 files.
Files used:	INDEX:Ø (Optional.)
	INDEX/SEQ:Ø (Optional.)
Files created:	Depends on file names specified.
Required free:	For Chapter 13 demo purposes, assuming the files don't already exist:
	Model I,II - 25 grans minimum recommended.
	Model III - 4Ø grans minimum recommended.
Comments:	Self prompting. See Chapter 13.

If you are using other than TRSDOS 2.3 on the Model I, you may want to change line 59060 back to the way it is listed in this book for greater speed. You may kill the files you created after the demonstration.

KALOADER/BAS

This program pokes the KEYACCES USR subroutine into memory so that KEYACCES/DEM can be used. You may also use this program to get KEYACCES into a disk file in a different format that may be more useful for your purposes. (Perhaps you will want to DUMP it, or change it to magic array format.)

Memory Size:	61439 or less.
Files setting:	No special setting required.
Files used:	None.
Files created:	None.
Required free:	None.
Comments:	See Chapter 14 and Appendix 1 first.

KEYACCES/DEM

This program demonstrates the KEYACCES USR subroutine. A file of up to 254 records is created, and you can quickly access individual records by 8-byte keys, using the machine-language hash index technique discussed in Chapter 14. You must run KALOADER/BAS prior to this program so that KEYACCES will be in memory. Be sure to take the "Initialize" option the first time you run the demo.

Memory Size:	61439 or less.
Files setting:	Specify at least 2 files.
Files used:	KADEM/MAS:Ø and KADEM/IND:Ø.
Files created:	KADEM/MAS:Ø and KADEM/IND:Ø.
Required free:	Assuming files don't already exist:
	Model I,II - 8 grans.
	Model III - 12 grans.
Comments:	See Chapter 14 and Appendix 1 first.

You may kill KADEM/MAS and KADEM/IND after the demonstration.

DSLOADER/BAS

This program pokes the DFSEARCH USR subroutine into protected memory. It should be run prior to DFSEARCH/DEM. You may also use it if you want to work with DFSEARCH in a different format or at a different address.

Memory Size:	65Ø24 or less.
Files setting:	No special setting required.
Files used:	None.
Files created:	None.
Required free:	None.
Comments:	See Chapter 15 and Appendix 1 first.

DFSEARCH/DEM

This program demonstrates DFSEARCH, the high-speed machine language random file search USR subroutine from Chapter 15. You must run DSLOADER/BAS first. The first time around, you'll also want a file on disk to practice with. Chapter 15 shows you how to create "TEST:1" for this purpose.

Memory Size:	65Ø24 or less.
Files setting:	Specify at least 1 file.
Files used:	TEST:1 (Optional. You specify name and drive.)
Files created:	File should already exist.
Required free:	DFSEARCH/DEM will not the file specified.
Comments:	Limited prompting. See Chapter 15 and Appendix 1 first.

You may kill the file you used for testing after the demonstration.

COPY1/BAS

This program can sometimes help you get around troublesome backups. It provides a file copy capability in BASIC with flexible retry, skip, fix and abort options. Chapter 16 tells you how to use it.

Memory Size:	No special setting.
Files setting:	Specify 3 files.
Files used:	You specify source and destination files.
Files created:	Destination file, if not already present.
Required free:	Assuming destination file is not present, enough space for a copy of the source.
Comments:	See Chapter 16 for helpful hints.

COPY2/BAS

This program can be used in recovery techniques on sequential files. It is discussed in Chapter 16 as a way to recover files not closed and jumbled program files. You may find other uses for it as a simple editor for sequential files. See COPY1/BAS for setup details.

DELARRAY/DEM

This is a demonstration of the DELARRAY USR routine. It shows how your BASIC program can erase arrays from memory, recovering the space, when they are no longer needed. It also serves as a short example of machine language USR routine programming. Be sure to look at Appendix 2 for a complete explanation.

Memory Size:	65280 or less.
Files setting:	No special setting required.
Files used:	None.
Files created:	None.
Required free:	None.
Comments:	See Appendix 2.

Suggested Reading

I consider myself quite lucky to live only 30 miles from Upland, California where my publisher, IJG Inc, is located. Meeting and getting to know the people there has opened my eyes to the real power of the TRS-80. The "Other Mysteries" series of books has gotten quite a reputation among TRS-80 users as a source of practical "how-to" information about Radio Shack microcomputers. Though they are not all focused on the subject of disk I/O, every one of the books has information of great value to those of us who are doing disk programming.

Rosenfelder, Lewis, *BASIC FASTER AND BETTER & OTHER MYSTERIES*, 1981, IJG Inc, Upland, California.

(This was my first "faster and better" book. If you don't have it yet, I urge you to get it. As a companion to *BASIC DISK I/O FASTER AND BETTER*, it has much more information about USR calls, data compression, searching and sorting. It has many compatible subroutines and function calls and useful appendices for hexadecimal conversions, DCB and buffer addresses.)

Pennington, H.C., *TRS-80 DISK AND OTHER MYSTERIES*, 1979, IJG Inc, Upland, California.

(This book was the first in the "Other Mysteries" series. It can be a real "life saver" because it concentrates on techniques of disk data recovery. In doing so, it gives excellent explanations of disk directories, boot sectors and more in language that even beginners can understand. I found the sections explaining the organization of BASIC program files, editor/assembler files, machine language object files, and word processing files especially helpful.)

Farvour, James, *MICROSOFT BASIC DECODED & OTHER MYSTERIES*, 1981, IJG Inc, Upland, California.

(This book decodes the BASIC interpreter that is stored in the TRS-80 Model I and III ROM. It gives addresses of valuable routines that can be called from USR subroutines. Though it is not for the TRS-80 Model II, you'll find that most of the routines are the same, but at different addresses. Searching for the same patterns in code opened up many new possibilites on the Model II for me. This book also has good information about diskette organization.)

Kitsz, Dennis Bathory, *THE CUSTOM TRS-80 & OTHER MYSTERIES*, 1982, IJG Inc, Upland, California.

(This book shows ways to modify the hardware of the TRS-80 to get new capabilities and improve reliability, but even if you're not willing to open up the case, you'll find the material valuable. As a disk system owner you'll like Chapter 11, which lists "111 Cures for the Common Crash".)

Wagner, Michael J., *MACHINE LANGUAGE DISK I/O & OTHER MYSTERIES*, 1982, IJG Inc, Upland, California.

(If KEYACCES and DFSEARCH whetted your appetite for more information about machine language disk I/O, this is the book for you. It gives addresses that your USR subroutines can call that aren't listed in the Model I TRSDOS manual, and it even shows how to go beyond the disk operating system so you can write your own disk I/O routines. This book will also help you to better understand the disk error messages that TRSDOS gives you.)

Farvour, James, *TRSDOS 2.3 DECODED & OTHER MYSTERIES*, 1983, IJG Inc., IJG Inc, Upland, California.

(This book gives the complete source listing for TRSDOS 2.3. If that isn't enough, each section is listed and explained in detail. Whether your interest is academic, or you've got a practical purpose in mind, this book is a gold mine of information.)

Other Books That Will Help You

For theory and background that isn't specific to any particular computer, certain books have been quite helpful to me. Among others, I've found it valuable to have the following books in my personal library:

Matthies, Leslie H., *THE NEW PLAYSCRIPT PROCEDURE*, Second Edition, 1977, Office Publications, INC., Stamford, Connecticut.

(This book describes a good approach that can be used for writing step-by-step operating instructions.)

Wirth, Niklaus, *ALGORITHMS + DATA STRUCTURES = PROGRAMS*, 1976, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

(You'll find a good discussion about B-trees and other common programming problems. Note that all examples are in the PASCAL programming language.)

Knuth, Donald E., *THE ART OF COMPUTER PROGRAMMING*, Volume 3, "Sorting and Searching," 1973, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts.

(This book also has a good section about B-trees, with mathematical analysis of the performance and efficiency of various programming techniques.)

Page, E.S. and L.B. Wilson, *INFORMATION REPRESENTATION AND MANIPULATION IN A COMPUTER*, Second Edition, 1978, Cambridge University Press, London, New York, Melbourne.

(The section on hash addressing provides good background reading. It also has good discussions about trees and sorting methods.)

Owner's Manuals

Perhaps it goes without saying, but there's no substitute for having the appropriate owner's manual by your side:

For the TRS-80 Model I. . .

TRSDOS & DISK BASIC REFERENCE MANUAL, 1979, Radio Shack, (Tandy Corporation), Fort Worth, Texas.

For the TRS-80 Model III. . .

TRS-80 MODEL III OPERATION AND BASIC LANGUAGE REFERENCE MANUAL.

TRS-80 MODEL III DISK SYSTEM OWNER'S MANUAL

For the TRS-80 Model II. . .

TRS-80 MODEL II OWNER'S MANUAL

Index to Major Subroutines

Subroutines — BASIC Disk I/O Faster & Better

58700-58745	**Detail File Handler**	195
58700-58703	Get Next Record in Chain	195
58710	Link and Add Record to Chain	195
58712	Load Pointers to Detail Record	195
58714-58715	Link Prior to New Detail Record	195
58718-58719	Link Master to New Detail Record	195
58730-58732	Delink One-Way Linked Detail	208
58740-58745	Delink a Detail Record	196
58800-58850	**Random Byte Addresssing Routines**	216-217
58800	String-Get (Calls 58850)	216
58810-58811	String-Put (Calls 58850)	217
58850	Compute, Get, and Field	217
59000-59600	**TREESAM Handler Subroutines**	263-266
59000	Get and Load Statistics	263
59005	Field and Get Statistics Record	263
59010-59011	Dimension Arrays and Field Node	264
59020	Get Data from Stack and Get Node	264
59030	Go Up one Level	264
59040	Load Current Level of Stack	264
59050	Write Updated Statistics to Disk	264
59060	Get Current Node if Not Present	264
59100-59160	Search for Key, KY$	264
59200-59250	Add Key and Data	264-265
59300-59320	Read Next Key (or Read First)	265
59400-59430	Read Previous Key (or Read Last)	265
59500-50568	Delete Key and Data	265-266
59600	Add Non-Unique Key	266

Index

Note: Italicized page numbers refer to definitions.

BASIC is not nearly as slow as most programmers think it is. *BASIC Faster and Better* shows you how to supercharge your BASIC, with 300 pages of fast, functions and subroutines.

You won't find any trivial 'checkbook-balancing' programs in this book – it's packed with *useful* programs.

Tutorial for the beginner, instructive for the advanced, and invaluable for the professional, this book doesn't just talk . . . it shows how! All of the routines are also available on disk, so that you can save hours of keyboarding and debugging.

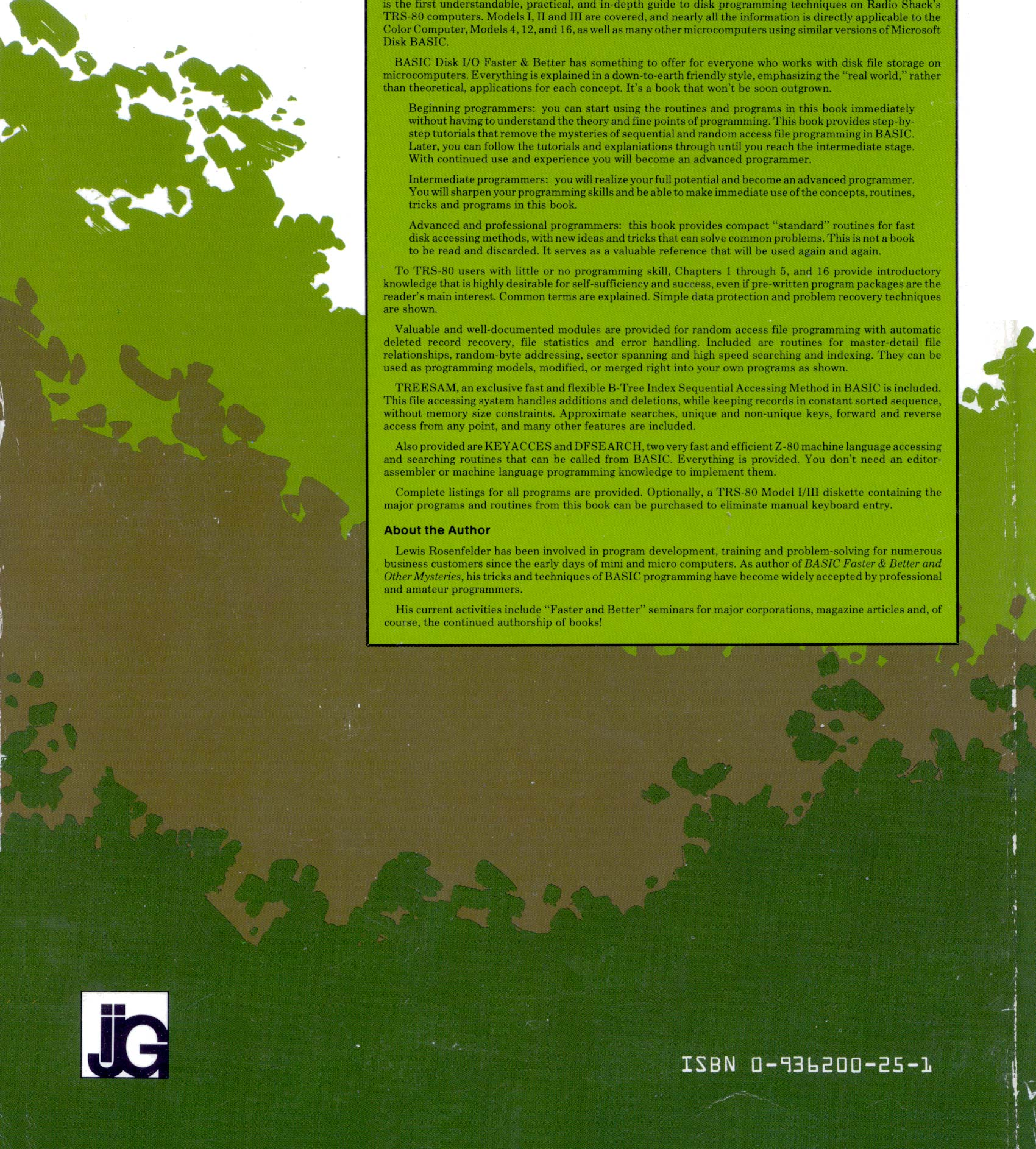
BASIC Faster & Better and Other Mysteries is $29.95, and the two program disks are $19.95 each. The #1 disk *BFBDEM* contains all the demonstration programs, and #2 *BFBLIB* has all the library functions.

Get the book and/or disks from your local *IJG* dealer and B. Dalton bookstores.

IJG

1953 West
11th Street
Upland,CA
91786 (714)
946-5805

TRS-80 DISK & OTHER MYSTERIES
MICROSOFT BASIC DECODED & OTHER MYSTERIES
MACHINE LANGUAGE DISK I/O & OTHER MYSTERIES
BASIC DISK I/O FASTER AND BETTER & OTHER MYSTERIES
BFBDEM
BFBLIB
GETTING STARTED ON THE SHARP 1500 & RADIO SHACK PC-2
THE CUSTOM TRS-80 & OTHER MYSTERIES
BASIC FASTER AND BETTER & OTHER MYSTERIES
GAMES and GRAPHICS FOR THE TRS-80 a BEGINNER'S GUIDE
TRSDOS 2.3 DECODED & OTHER MYSTERIES
HOW TO DO IT on the TRS-80
RED pencil
BLUE pencil
ionary-proofing
michael shrayer's ELECTRIC pencil
easy to learn easy to use
word processing system



BASIC Disk I/O Faster and Better & Other Mysteries by Lewis Rosenfelder

The author of the best selling *BASIC Faster and Better & Other Mysteries* has done it again! This 432 page book is the first understandable, practical, and in-depth guide to disk programming techniques on Radio Shack's TRS-80 computers. Models I, II and III are covered, and nearly all the information is directly applicable to the Color Computer, Models 4, 12, and 16, as well as many other microcomputers using similar versions of Microsoft Disk BASIC.

BASIC Disk I/O Faster & Better has something to offer for everyone who works with disk file storage on microcomputers. Everything is explained in a down-to-earth friendly style, emphasizing the "real world," rather than theoretical, applications for each concept. It's a book that won't be soon outgrown.

> Beginning programmers: you can start using the routines and programs in this book immediately without having to understand the theory and fine points of programming. This book provides step-by-step tutorials that remove the mysteries of sequential and random access file programming in BASIC. Later, you can follow the tutorials and explaniations through until you reach the intermediate stage. With continued use and experience you will become an advanced programmer.
>
> Intermediate programmers: you will realize your full potential and become an advanced programmer. You will sharpen your programming skills and be able to make immediate use of the concepts, routines, tricks and programs in this book.
>
> Advanced and professional programmers: this book provides compact "standard" routines for fast disk accessing methods, with new ideas and tricks that can solve common problems. This is not a book to be read and discarded. It serves as a valuable reference that will be used again and again.

To TRS-80 users with little or no programming skill, Chapters 1 through 5, and 16 provide introductory knowledge that is highly desirable for self-sufficiency and success, even if pre-written program packages are the reader's main interest. Common terms are explained. Simple data protection and problem recovery techniques are shown.

Valuable and well-documented modules are provided for random access file programming with automatic deleted record recovery, file statistics and error handling. Included are routines for master-detail file relationships, random-byte addressing, sector spanning and high speed searching and indexing. They can be used as programming models, modified, or merged right into your own programs as shown.

TREESAM, an exclusive fast and flexible B-Tree Index Sequential Accessing Method in BASIC is included. This file accessing system handles additions and deletions, while keeping records in constant sorted sequence, without memory size constraints. Approximate searches, unique and non-unique keys, forward and reverse access from any point, and many other features are included.

Also provided are KEYACCES and DFSEARCH, two very fast and efficient Z-80 machine language accessing and searching routines that can be called from BASIC. Everything is provided. You don't need an editor-assembler or machine language programming knowledge to implement them.

Complete listings for all programs are provided. Optionally, a TRS-80 Model I/III diskette containing the major programs and routines from this book can be purchased to eliminate manual keyboard entry.

About the Author

Lewis Rosenfelder has been involved in program development, training and problem-solving for numerous business customers since the early days of mini and micro computers. As author of *BASIC Faster & Better and Other Mysteries*, his tricks and techniques of BASIC programming have become widely accepted by professional and amateur programmers.

His current activities include "Faster and Better" seminars for major corporations, magazine articles and, of course, the continued authorship of books!

ISBN 0-936200-25-1