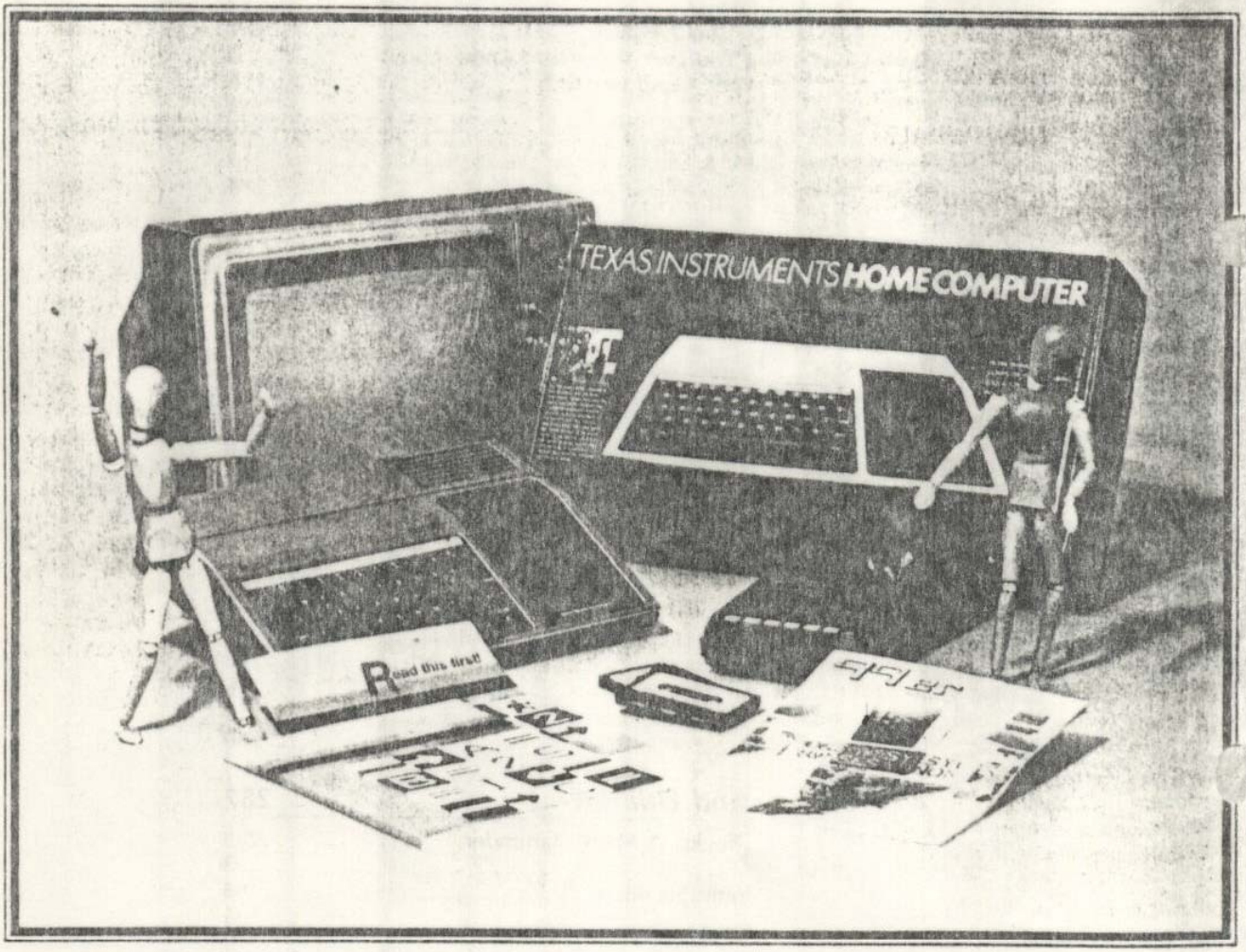
The Best of 99'er™
Volume 1

From the Editors
of
99'er Home Computer Magazine

Emerald Valley Publishing Co.
Eugene, Oregon

Contents

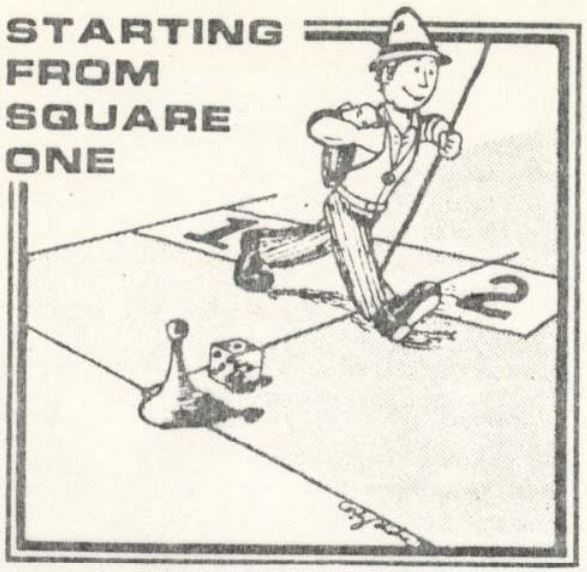
1
Starting Out



1

Starting Out

Taking the first steps on an exciting adventure!



How to Buy A Computer

". . . be assured that you are embarking on an exciting adventure . . . and realize that ownership is not only exciting but helpful and productive too."

In this article I will offer some suggestions to help those of you shopping for your first personal computer. I will not directly compare brand names, nor will I attempt a technical critique of the TI-99/4A home computer—but I will point out some of the TI machine's exceptional features.

What follows is a general discussion of computer shopping techniques written by and for the computer novice who is experiencing the bewilderment of trying to make a wise purchase in a market exploding with products. I offer these suggestions from the perspective of a writer who is not a computer professional. I have owned a TI-99/4 for a year and a half and also recently bought a competitive brand computer. In addition, I plan to purchase a third brand during 1982. Therefore, I am not dedicated to a single brand of computer although I am impressed with the 99/4A capabilities. All of my comments apply equally to the 99/4 and the new 99/4A unless otherwise noted.

My computers are used to develop computer-assisted instruction (CAI) for applications in the field of rehabilitation. The following suggestions result from the actual experiences of a beginner faced with the task of learning about computers—one who has spend literally hundreds of hours poring over manuals and magazines, and peering into a monitor screen.

Because my background is in psychology and counseling I can't resist beginning with some general, facilitative remarks. First of all, no matter which computer you eventually buy, you will later regret your choice at times. No one computer will have all the features you want; you'll have to compromise on some features—just remember that the grass always looks greener. . . so be aware that your buyer's anxiety may not totally disappear the instant you take possession of your new computer.

Secondly, regardless of how impressed you are with your new computer's gee-whiz features, you will quickly adjust your expectations upward. Whatever you buy now you will probably soon want to expand, with either more hardware (machinery/gadgets) or more software (programs).

Thirdly, start now! Don't wait for computers to come down to $9.98—they probably never will. The manufacturers will just keep on making them more sophisticated for about the same money.

And last of all, don't expect your friends, spouse, etc., to be as thrilled as you are about your computer. It is up to *you* to educate *them*.

Who Buys A Personal Computer

Rumor has it that someone once tried to profile the "typical" personal computer buyer for more effective marketing strategies. The survey data showed one shared factor: The majority of buyers wanted to become rich by writing and marketing a very successful program. In other respects, they are all different, and are using their computers for myriads of different purposes. So you're not alone when you go out to buy a computer—you may even find yourself in one of the following categories:

Type 1—The electronics amateur who is intrigued by all the technology. Fiddling with the equipment is enough reason for him to buy. We should be grateful to him: When he began buying kits and tinkering around with them a few years ago, he started the home computer craze.

Type 2—The aware parents who want the family to be up on "the latest." The family can play games and learn about computing as well as do the budget, and so on. The average family will want a computer that is flexible, versatile, inexpensive and "friendly" (easy to use). It should be expandable so that it can grow as the family's needs grow. This market has yet to peak.

Type 3—The small business owner or professional person who wants to automate the office. He will agonize over how much computer to buy. If it's not enough, it could well become merely a toy for his kids, but why buy a $10,000 system if a $3,500 package will do the job? This system will probably need both large amounts of data storage capacity and word processing capabilities.

Type 4—The educator interested in computer-assisted instruction (CAI). He or she will need a computer capable of displaying eye-catching color graphics and animations along with text, speech and sound.

Type 5—The scientist or engineer who will use the machine at work or home because it is easier than standing in line to get on the company's big main frame computer. Even companies that own big "braniac" computers are buying micros to spread around to key people.

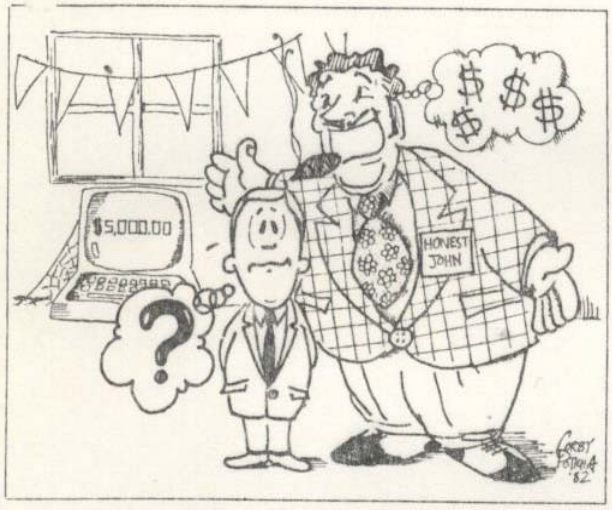
The list could go on and on, but I hope I have made my point that the "typical" microcomputer buyer is anything but typical. We all have one thing in common, however.

We have all been bitten by the computer bug and the only known cure is to take the plunge and get our very own microcomputer!

Types of Sellers

If you as a buyer are feeling overwhelmed by all the computer choices, pity the typical salesperson. He may be more at home with stereos and televisions, and entirely new to computers. Or he may be a programmer or technician entirely new to selling. Odds are that you'll meet the former more often at your local computer outlet; just as buyers exist at every level of sophistication, so do sellers. More important than knowledge of computer technology, though, is the willingness of the computer salesperson to help you learn. After all, we're all new to personal computing.

If you haven't already, you will shortly encounter at least one of the following salespeople:



Type 1—The sincere young man or woman who produces a nervous smile and confesses, "I only started in this department yesterday; let me see, where is the power switch on this little beauty. . ." Don't leave too soon, though. If you've the time and patience, you and the trainee can learn a lot about the computer in an hour.

Type 2—The equally sincere salesperson who introduces himself and says, "What can I show you. . .we have a 48K whiz-banger with a double DOS and CP/M on special. . ." This individual will joyously prattle on until your glazed eyeballs communicate either lack of interest or comprehension. (They are equivalent in the clerk's opinion.) You can then leave the store with a handful of pamphlets and a heart full of doubt—and possibly a car full of computer.

Type 3—The merchandising expert who moves computers the same way he used to move TV sets, stereos, etc. This type cannot refrain from knocking the competition by saying things like, "Brand X is almost out of business, that's why we don't handle 'em. . .what'd ya' say you do for work. . .I sell a number of Crunchy 100's to people in your neighborhood." This individual may be able to tell you a lot about his computer since he will be shrewd enough to read up on all the features of his machine; you may actually learn something if you have the confidence and patience to endure a barrage of irrelevancies.

Type 4—The skilled and sensitive sales professional who has developed a good knowledge of computing, or vice versa, the computer professional who has developed basic competence as a salesperson. This person will ask you right off what you want to do with your computer and help you with the answer if you aren't sure. You will appreciate this individual's patience and willingness to find out information for you. He or she will consult with a superior or even call the manufacturer without fear of appearing ignorant. When you meet people like this, respect their time and effort and show your appreciation. We don't want them to get discouraged and switch jobs. There will be little danger of this, however, since they will probably be making a lot of sales with many happy customers!

How To Shop

Be careful not to equate the amount of advertising you see for a computer with its technical sophistication or suitability for your needs. Take the time to go beyond mere advertising when you shop. Talk to computer owners, or visit a local computer club. But remember to expect some very prejudiced views, because people always try to convince themselves that *their* choices are best. Be cautious, too, of magazine reviews of various computers. Articles with extensive charts and diagrams may look impressive, but they are sometimes simply wrong. I have read articles which declared that the TI machine had no high-resolution color graphics or memory expansion capabilities. Well, TI has one of the best high-resolution color graphics capabilities on the market and can be expanded to a 48K system. I have noticed similar errors on other brands as well.

So visit several stores, read a few computer magazines, like *Home Computer Magazine*, and get your confidence up so the salesperson won't intimidate you. I am impressed with the TI-99/4A as I grow more familiar with it, but very little of this knowledge came from advertising or from salespeople: It came from *use* of the machine.

You may also need to know a little computer jargon, although the better salespeople will avoid trying to impress you with their vocabulary. If you don't already have one, pick up a glossary of terms while you are out for your first visit to the computer store. For starters, you should study the accompanying glossary for an understanding of its terms: With just a few of these terms tucked away in your memory banks, you can walk into the computer store with more confidence and less quiver in your voice when you ask to see the "Brainiac 3000" computer.

Ask to see a demonstration of each computer you think you can afford. But be aware that many demonstration programs you see are written in a program language other than BASIC—i.e., the language available to the user on most small computers. Consequently, the demo may be super impressive with lots of color graphics, animation, and sound, but find out if you can duplicate these effects readily with the BASIC programming language available to you. If you are interested in having good color graphics in your programs, ask the salesperson to enter some simple statements in BASIC to illustrate the computer's ability to perform the following:

A. Clear the screen.
B. Change the screen color.

C. Plot the color shapes on the screen. Try to place a "duck" or a "car" on the screen. Find out if the user can create his own shapes or is he limited to pre-defined shapes stored in the computer's memory.
D. Place a graphic shape and text on the same screen. Some computers can do one or the other without elaborate and difficult programming.

Happily, the 99/4A does all of the above with ease. You can program in 16 colors with simple, easy-to-use BASIC statements. If graphics are important to you, check out the TI Extended BASIC graphics capabilities. They are sensational and compete with computers costing as much as a thousand dollars more. If you want sound capabilities in your program, ask for a demo of the following:

A. Play a three note chord.
B. Play a simple scale.
C. Demonstrate the highest and lowest frequency programmable.
D. Demonstrate the loudest and softest volume of sound possible.
E. Create sound effects like a "choo-choo" or an "explosion."

Speech synthesis adds an exciting dimension to computing, especially in educational programs. Texas Instruments makes it easy to integrate speech into BASIC progams with its speech synthesizer and *Speech Editor* Command Cartridge or the *Terminal Emulator II* Command Cartridge. The *TE-II* will synthesize any English word typed into the computer; the *Speech Editor* will allow you to choose from a vocabulary of over 300 words. By all means get a demonstration of speech synthesis if you are interested in computer-assisted instruction—it is well worth the added cost.

The Editor

Regardless of the type of use you plan for your computer, you will definitely need a good editor. However, if you can think and type without errors, you can skip this section and not worry about editing.

Good, you are honest! I found out the importance of an editor the hard way. Not one salesperson mentioned this feature in any of my shopping except to say that I could correct errors. From this treatment of the subject, you might conclude that all editors are alike. The galaxy of differences between computer brands and their editing capabilities can make them either a joy or a pain to use.

So, what is an editor? Somewhere buried in all that fabulous circuitry is a component which interprets all of the instructions you type in. It turns your instructions—words—into the ones and zeros that the computer understands. It interprets the program for the computer. It will also edit or change, program statements after they have been entered into the computer. When you are writing and debugging (removing errors from) programs, you are bound to make typing errors. Typing the whole line over would correct these, but it is very time consuming and irritating, especially when there may only be one or two mistakes in 25-50 characters! If you could only correct the mistakes without disturbing the rest of the line.

You can: A good editor will permit you to modify a line of a program by inserting or deleting characters or words with a single keystroke, while displaying the changes on the monitor screen exactly as made in the program. A poor editor will require multiple keystrokes, and won't display the corrections as they are made. It will make you pound many keys and ultimately resort to retyping. The TI editor is far superior to my Number 2 computer's editor, and is equivalent to a good word processor in its correction capabilities. (I am writing this article on my 99/4 using a simple word processing program I wrote myself. It uses all the editing features resident in the computer and works very well for editing text.)

I cannot overemphasize the importance of the editor, and strongly recommend that you evaluate it carefully before you buy. Sit down at the keyboard and have the salesperson walk you through some editing. Don't let the clerk do it because he may pick simple tasks to make it look easy.

For instance, you might enter this program line;

```
100 PRINT "NOW IS THE TIME FOR ALL GOOD MEN TO COME TO THE AID OF THERE COUNTRY."
```

(If you are new to programming, let me point out that this BASIC statement will cause the words inside the quotes to be displayed on the monitor if you RUN the program.)

Notice that the word *THERE* is mispelled; so correct the spelling without retyping the entire line, then insert the word *BEST* before the *TIME*. If you can't accomplish this by the store's closing time, ask the salesperson to do it; if he can't do it with ease, give serious thought to buying another brand of computer.

While you are at this, ask the salesperson to demonstrate *resequencing* for you. Resequencing is a simple but valuable (and frequently unavailable) feature which permits you to renumber your program line numbers in order to insert additional lines into an existing program if necessary. For example, you might type in this simple BASIC program:

```
10 PRINT "HELLO"
11 PRINT "WHAT IS YOUR NAME?"
12 INPUT N$
13 PRINT "THANK YOU ";N$
14 END
```

Notice that you don't have any room between lines for additional lines. If you later decide to change the program, you either have to type the program over or resequence the line numbers to provide space. Normally, you don't intentionally get yourself into corners where it is necessary to resequence your programs, but it does frequently happen (courtesy of Murphy's Law). On the TI machine, resequencing is easily accomplished by typing RES and pressing the ENTER key. Presto! The program looks like this.

```
100 PRINT "HELLO"
110 PRINT "WHAT IS YOUR NAME?"
120 INPUT N$
130 PRINT "THANK YOU ";N$
140 END
```

Now you can add additional lines between the original ones. Many computers do not have the resequencing function built in so you have to load in a separate program from a disk or tape. This function is important enough that it should be built into the machine as it is in the TI-99/4A.

General Considerations

Regardless of the sophistication of the system, you should expect certain fundamental "creature comforts." First, it

is mandatory that the screen be clear and easy on the eyes. You may not fully appreciate this during a brief demonstration in the store, but spend an hour or two peering at the screen in your basement, and you'll know what I mean. Your 19-inch TV at home may not display characters as sharply as the store's 9-inch monitor. On a big screen, the characters may appear more ragged because the dots composing the characters are larger and more spread out. Instead of white characters on a black background or vice versa, the TI has exceptionally sharp black characters formed by an 8 × 8 dot matrix with a pale blue background. It's also possible to change the characters and background to any of 16 colors.

My only criticism of the TI display capabilities is that with TI BASIC it is limited to a line of 28 characters for text or 32 for graphics. [With TI's *Editor/Assembler* or TI-Writer Command Cartridge, you have a 40 character "window" which automatically scrolls horizontally across an 80-column "page." The Video Display Processor chip inside the computer actually has a 40-column "text mode," and the software produces the doubling effect.—Ed.] Some computers display fewer, but many display lines up to 80 characters or more. My Number 2 computer displays 40, but I see little practical difference between it and the TI machines. However, the 80-character display and lower-case characters are desirable if you plan to do extensive word processing (letters, reports, etc.). The TI-99/4A has a type of lower-case which is actually compressed upper-case; it works very well. You can do word processing with a 28 character format, but you won't be able to see the text on the screen exactly as it will appear on the printed page; with an 80 column format, however, you will. Your printer should have the capability of printing both upper- and lower-case characters with the proper program, so that you need not worry about having lower-case resident in your computer.

Another "creature comfort" to consider is the computer keyboard. The original TI-99/4 was criticised for having a keyboard smaller than a conventional typewriter. Actually it is very easy to use and one can touch-type on it very efficiently. But TI modified the keyboard on the TI-99/4A so it is more like a standard typewriter and added some function keys and a repeat key function to improve the computer's flexibility.

If you select a disk system for program and data storage rather than a cassette tape system, you will have the advantages of speed and convenience but you will sacrifice something, too. In addition to the higher cost of the disk system (maybe 10 times the cost of a tape recorder), you will also lose some of your program space (random access memory, known as RAM) inside the computer. Some systems will have a 2K overhead (2000 bytes) while others may require 10K or more. It is desirable to have a low overhead so that your valuable program memory space will be available for programs. The 99/4A disk system digests about 2K of your RAM leaving a nominal 14k for programs (on the standard 16K system). To put this into perspective, one page of typed, double-spaced material with liberal margins is equivalent to about 2K of information. If you buy a 16K computer which has a 10K overhead for the disk system, you would only have about 6K of program space after you turn on the disk system. And one of the very popular computer brands actually has a 10K overhead! So when you look at one in a store the salesperson will probably insist that you get (and pay for) at least 32K of RAM. Moral: 16K memory in computer "A" does not necessarily equal 16K memory in computer "B." Texas Instruments gives you a lot of memory for the money.

How much memory will you need in your computer? For most home use, a 16K computer is generally considered a satisfactory start. For business and educational applications you will probably need more memory—48K is satisfactory in most cases. That covers your program requirements inside the computer. For permanent storage of large amounts of data such as student grade records and inventory reports, you will use disk or tape. Such storage is relatively cheap. A diskette (called a "floppy disk" because it is flexible plastic) can store 90K or more of information on a 5¼inch surface costing a mere four or five dollars. You can store the equivalent of about 50 typed pages on one such disk. Cassette tape is okay for home use and for back-up copies of your disk data, but is generally too slow for serious business or educational applications.

Service

Check out the service policy on your computer before you buy. Some manufacturers will exchange defective components, and others want to repair and return the original unit. If *downtime* is critical to you, choose the system which can be replaced in the shortest time. My 99/4 developed intermittent problems after more than a year of very heavy use, and TI exchanged it for a factory rebuilt unit for only $45.00 with same day service and no questions asked. If trouble develops during warranty, the exchange charge is minimal. When I thought I had a defective disk system during the third month of ownership, the service center would have exchanged the entire disk system for about $3.50, but as it turned out, I had a bad diskette instead.

Where to Buy

Deciding where to buy your computer can be difficult. Should you buy from a local computer store, a department store, or perhaps from a mail-order outlet? You can get some terrific bargains from a mail-order firm. You'll see dozens of ads in any computer magazine and nearly all will accept credit cards, making it very easy to buy. I saved nearly 40% on my TI machine buying it from a firm in another city across the state; my Number 2 computer cost almost $500 less from out-of-state company than from the local store. The argument for buying from a local dealer and paying more is that you can count on better personal service if your machine goes on the blink. This may or may not be true depending on your dealer's integrity and quality of service. You could buy locally and still have problems with service. In my opinion, the overhead of the local computer store justifies the higher prices. If you can afford it and desire peace of mind, buy locally.

In the case of TI computers, you can exchange the defective unit for a factory rebuilt unit at one of the exchange centers. It won't matter where you originally purchased the unit. You can check with your local dealer to see if a service center is near you.

Another point to consider is that we really should not abuse the local computer store owner's time by letting him educate us if we have no intention of buying locally. It is fair to expect him to compete with other dealers for our dollar by demonstrating his wares and services, but unfair

to sit through an hour or two of free demonstrations if we've already decided to buy through the mail. After all, we want the computer store to succeed, since it will advance personal computing in general.

Miscellaneous Points

Ask the salesperson if the computer you select can perform the graphics, sound, and text functions you desire just as it comes out of the box, or must you buy additional attachments or plug-in devices. You may find the demonstration you witnessed on a "loaded" floor model cannot be performed on a basic unit without adding several hundred dollars of additional equipment. On the other hand, you may find that most of the desirable features are built right into the basic computer.

It is also essential to have clear, concise, easily understood manuals which explain how to use your computer. You should not have to have any knowledge about computers to understand the basic introductory and tutorial manuals for your computer.

If you have not yet bought that first computer, be assured that you are embarking on an exciting adventure. The excitement and pride you'll experience when opening the box on the first day is like a dozen Christmas celebrations combined. Enjoy the experience, and realize that ownership is not only *exciting* but *helpful* and *productive* too.

In the meantime read all you can and shop carefully until you just can't stand it any longer. . .then take the plunge. Go out and get that computer!

Glossary of Terms

BASIC—Beginners All Purpose Symbolic Instruction Code is a program language developed at Dartmouth in the early 60's; it is the most common of all programming languages for small computers. BASIC is relatively easy to learn and is an effective and powerful language for most small computer applications.

bit—The smallest piece of information your computer deals with. It is equivalent to a circuit being turned either on or off. Like a light bulb, a computer logic circuit is either on or off; this equals one bit of information. Most home computers use an 8-bit microprocessor, but Texas Instruments and IBM have a 16-bit microprocessor. The advantages of the 16-bit configuration are too technical for this discussion, but we can generally say that more powerful and accurate computing can be accomplished. It has been predicted that the 16-bit microprocessor will be the future industry standard.

byte—The amount of memory necessary to code a character (a number/letter/punctuation, etc.) A byte has 8 bits in it. A computer which has 16K bytes of memory has 16 thousand bytes and can work with about 16 thousand characters of information in a single program.

chip—The circuits of the computer are fabricated on silicon chips. A chip is typically about 1/4 inch on a side. Today's chips are so sophisticated that the basic components of an entire computer can be fabricated on a single chip.

CRT (monitor)—The TV-like screen (cathode ray tube) to which the computer outputs information like numbers/letters/graphs, etc.)

disk drive—The accessory which stores and retrieves information on plastic (mylar) diskettes. The DOS (see below) controls the operation of the disk drive.

disk operating system—Sometimes called DOS and sometimes pronounced like "DOSS." It is the set of instuctions (software) which controls the storing and retrieving of information with the disk drive.

diskette—A plastic disk coated with an oxide upon which data and programs are stored using the disk drive under control of the DOS. Diskettes come in either of two sizes, 5 1/4 inch or 8 inch. The TI-99/4A uses the 5 1/4 inch.

firmware—Generally speaking, firmware is a chip in which a program has been stored permanently. It is "soft" in that it is a program (see software) but "hard" to the extent that it is an electronic chip rather than a diskette or tape. Hence it is "firmware." Firmware is used to store programs which are used repeatedly, and need not be changed or modified. (see ROM)

hardware—The actual physical machine, i.e., keyboard, CRT, printer, etc.

integrated circuit (IC)—If you look into the back of an old radio, you will see a lot of resistors, capacitors, and the like. Each component will be discrete—i.e., separate from other resistors, etc. which surround it. Integrated circuits, on the other hand, have many such individual components packed together or integrated in a small area. (See chip.) If you peer into a computer, you will see rows of little black boxes plugged into circuit boards. Each little black rectangle may have thousands of components integrated into it.

input/output (I/O)—Input is the data that goes into the computer via the keyboard as well as disk drives, tape recorders, etc. Output is what comes back out of the computer to the monitor screen, disk drive, tape recorder, and printer. (Throughput is what happens in between).

microcomputer—All computers used to be very large and esoteric and were called "mainframes." But miniaturization with integrated circuits has resulted in very powerful computers of small size coming into being. That is, you could pack a lot of computer into a very small box. These computers were initially called "minicomputers." But as the reduction in size continued, small desktop-size computers were produced with sufficient computing capacity to still be very useful. These are called "microcomputers." The difference in power between the mini and the micro is diminishing rapidly, so that it will soon be difficult to tell a mini from a micro. For now, all home computers are considered microcomputers.

modem—A device that connects your computer to the telephone so you can communicate with other computers. It works by MOdulating and DEModulating a sound tone.

peripherals—All those hardware devices which plug into your computer such as disk drives, tape recorders, printers, and modems.

printer—A peripheral device which will print a copy (called hardcopy) of your computer's output. Very handy to have for correspondence and for program debugging.

program—The set of coded instructions which directs the activities of your computer. Without a program, your computer is just so much metal and silicon junk. (See software and BASIC.)

RAM—Traditionally, the abbreviation for random access memory. But the name is a little misleading. Both RAM and ROM memory are random access. More accurately, RAM should be described as read and write memory (contrast with ROM). RAM is the memory you are using when you program a computer. It is also the memory to which your computer salesperson is referring when he says, "This one has 16K memory." The more RAM you have, the bigger programs you can run. When you turn your computer off, all the contents of RAM is erased. So if you wish to avoid having to type in hundreds of program lines everytime you use your computer, you must save programs on tape or disk for future use.

ROM—This is read only memory. That's right, you cannot "write" anything to a ROM; you can only "read" it. This means that you cannot change the contents of a ROM memory like you can a RAM memory. ROM contents are usually not changed; therefore they are used for *firmware*.

RS-232C—A common interface specification used to define the link between the computer and some other device like a modem or a printer.

software—It is not the physical machine (hardware) and usually not the permanent programs stored on chips (firmware) that instructs the computer on how to perform a task. It is the program stored on disk or tape. You can see that a tape or plastic disk is not as much a part of the computer as a chip (not as "firm"); therefore the programs stored on tape or disk are called "software."

NOW WHAT?

Congratulations, you're the new owner of a TI-99/4A Home Computer!! Now what? You have it all unpacked and need to know what to do with it, right? Fortunately, you have *The Best of 99'er*, and we'll give you a few ideas to start you on your way.

Of course you can plug in a variety of Command Cartridges that can teach you exercise, challenge you to a chess game, help with your finances, or do a multitude of other things. But the real fun and challenge is making that machine do what *you* want it to do.

When I got my computer, many of my friends asked, "Well, what can it do?" And the next questions were: "Can you balance your checkbook with it?" "Can you file names and addresses?" "Can you keep track of other things such as household inventories?" "Can you do your income taxes?"

The TI-99/4A is so versatile that you *can* do all of these home applications plus a myriad of business and professional applications. You'll soon be "hooked" on your computer and be one of those computer nuts who stay up all night saying, "I'll just make *one* more change in this program, and then . . . ."

Let me just give you a few ideas for programming and then you'll be on your own.

Most households own a calculator. Now, with a calculator you just punch in numbers and symbols and get an answer. Your computer can manipulate numbers too, but it can also interact with you, using words. And it can do the same process over and over again. You can also save your program and the data and use it again a month or a year later. You'll soon find your computer is a valuable household addition.

To make an interactive program you'll need to use PRINT or DISPLAY and INPUT. PRINT and DISPLAY do the same thing on the screen in TI BASIC. You probably have discovered how to PRINT messages, so let me just give you one hint here. A colon in a PRINT (or DISPLAY) statement means, "Go to the next line." The screen will be much easier to read if you have a few spaces here and there rather than all the printing jammed up. You may use more than one colon in the statement to get more blank lines. Here's an example:

```
100 CALL CLEAR
110 PRINT "THIS IS A SAMPLE."
120 PRINT : "HELLO" : "HOW ARE YOU?" ::::
130 PRINT "START SPACING HERE."
140 PRINT : "I SKIPPED ONE LINE."
150 PRINT :: "I SKIPPED TWO LINES."
```

I usually start a program by clearing the screen. Line 110 prints a message. You'll notice the line actually prints then moves up one line. The first colon in Line 120 says, "Go to the next line," then print HELLO. Another colon—so HOW ARE YOU? starts on the next line, then you "go to the next line" four times. The number of blank lines is the number of colons at the end of the line, minus one. If the colons are at the beginning of a statement, the number of lines is equal to the number of colons. Don't get confused—just RUN this program and experiment a little to learn how to use the spacing effectively.

INPUT is how you enter something from the keyboard while the program is running. You may PRINT a message and then INPUT like this:

```
100 PRINT "WHAT IS YOUR NAME?"
110 INPUT NAME$
```

Remember, string variables need $ at the end of the variable name; numbers do not. This program will print the message, then print a question mark on the next line, blink the cursor and wait for the user to enter something.

INPUT also allows a prompting message:

```
100 INPUT "WHAT IS YOUR NAME?":NAME$
```

This time the cursor will blink in the space immediately following the prompt message and print your response there as you key it in.

When programming responses, you generally use INPUT. However, on a one-stroke answer I like to use CALL KEY. The user will just have to press one key (won't have to press ENTER), *and* you can block out unacceptable answers. For example, suppose you need a yes or no answer.

```
400 PRINT "ANSWER Y OR N"
410 CALL KEY(0,KEY,STATUS)
420 IF KEY = 78 THEN 500
430 If KEY < > 89 THEN 410
440 Continue here for "Yes" answer
.
.
.
500 Continue here for "No" answer
```

Only Y and N are accepted; any other key pressed is ignored. Another example is:

```
400 PRINT "CHOOSE 1, 2, 3, OR 4"
410 CALL KEY (0,K,S)
420 IF K<49 THEN 410
430 IF K>52 THEN 410
440 ON K-48 GOTO 1000,2000,3000,4000
```

Only 1, 2, 3, or 4 will be accepted, then the program will branch to the appropriate section. Remember that the K value in the CALL KEY statement is the ASCII code number of the character pressed.

Now you are armed with some basics of interactive programming. Let's try some specifics and answer those questions above.

Checkbook Balancing

Ah-ha! That's already in your TI-99/4A *User's Reference Guide,* page III-22. Just key that program in and add your own embellishments to make it *your* program. I like to take advantage of TI's color and sound to enhance a program, so let's add a little color at the beginning. Add:

```
115 GOSUB 500
```

This means go down to Line 500 and do some stuff then come back. Now add Lines 500 to 660. A complete modified listing follows. Try it. Then adapt it to what you want.

```
100 REM  CHECKBOOK BALANCE
110 CALL CLEAR
115 GOSUB 500
120 INPUT "BANK BALANCE: ":BALANCE
130 DISPLAY :"ENTER EACH OUTSTANDING"
140 DISPLAY "CHECK NUMBER AND AMOUNT."
150 DISPLAY
160 DISPLAY "ENTER A ZERO FOR THE"
170 DISPLAY "CHECK NUMBER WHEN FINISHE
    D."
180 DISPLAY
200 INPUT "CHECK NUMBER: ":CNUM
210 IF CNUM=0 THEN 250
220 INPUT "CHECK AMOUNT: ":CAMT
230 CTOTAL=CTOTAL+CAMT
240 GOTO 200
250 DISPLAY "ENTER EACH OUTSTANDING"
260 DISPLAY "DEPOSIT AMOUNT."
270 DISPLAY
280 DISPLAY "ENTER A ZERO AMOUNT"
290 DISPLAY "WHEN FINISHED"
300 DISPLAY
320 INPUT "DEPOSIT AMOUNT? ":DAMT
330 IF DAMT=0 THEN 360
340 DTOTAL=DTOTAL+DAMT
350 GOTO 320
360 NBAL=BALANCE-CTOTAL+DTOTAL
370 DISPLAY :"NEW BALANCE= ";NBAL
380 INPUT "CHECKBOOK BALANCE: ":CBAL
390 DISPLAY "CORRECTION= ";NBAL-CBAL
400 END
500 CALL CHAR(96,"0")
510 CALL CHAR(97,"0000FF")
520 CALL CHAR(98,"8618433402CC61")
530 CALL COLOR(9,13,11)
540 CALL HCHAR(10,6,96,18)
550 CALL VCHAR(11,6,96,6)
560 CALL HCHAR(11,7,98,5)
570 CALL HCHAR(11,12,96,4)
580 CALL HCHAR(11,16,97,8)
590 CALL HCHAR(12,7,96,17)
600 CALL HCHAR(13,7,97,17)
610 CALL HCHAR(13,18,96,2)
620 CALL HCHAR(14,7,97,17)
630 CALL HCHAR(15,7,96,17)
640 CALL HCHAR(16,7,96,9)
650 CALL HCHAR(16,16,97,8)
660 RETURN
```

As the program is written in the manual, there may be a few problems. There is no DIMension statement, so if you have more than ten outstanding checks or deposits you will get an error. Because there is really no need to even worry about subscripts, delete (N) in Lines 200, 210, 220, and 230 and (M) in Lines 320, 330, and 340. You may then also delete Line 190 and change Line 240 to GOTO 200; and delete Line 310 and change Line 350 to GOTO 320.

Remember what I said above about spaces, and insert a colon before the first quote mark on Lines 130, 250, and 370 to make the screen easier to read. You may wish to add SOUND and red lines if the balance or correction is negative. Try your own ideas.

Name and Address File

Another easy solution—find Issue 2 of *99'er Magazine* and use the *Electronic Home Secretary* program. [Reprinted in this volume.—Ed.] What? You haven't keyed it in yet?? I thought *everyone* grabbed his issue of *99'er* and immediately keyed in *all* the programs!!

You can probably use this program as is, or adapt it to your needs to make your address file, phone list, Christmas list, or even a wedding invitation list. You can add a printer to print address labels if you want.

Recipe Conversions

How about recipes? Some people cook with a dab of this, a glug of that, enough flour until it looks right, and cook it until it's done. But a computer is more precise and will give you exact amounts. Try this program to convert a recipe.

```
100 REM  RECIPE CONVERSION
110 CALL CLEAR
120 DIM AMT(20),M$(20),ING$(20)
130 PRINT "ENTER NUMERICAL AMOUNT"
140 PRINT "(USE DECIMAL IF FRACTION)."
150 PRINT "THEN ENTER MEASURE (C,TSPP.
    )"
160 PRINT "THEN ENTER INGREDIENT."
170 PRINT :"ENTER '0' TO END RECIPE."
180 INPUT "AMOUNT: ":AMT(I)
190 IF AMT(I)<=0 THEN 250
200 INPUT "MEASURE: ":M$(I)
210 INPUT "INGREDIENT: ":ING$(I)
220 I=I+1
230 PRINT
240 GOTO 180
250 CALL CLEAR
260 FOR J=0 TO I-1
270 PRINT AMT(J);M$(J);" ";ING$(J)
280 NEXT J
290 PRINT :"MULTIPLY BY WHAT NUMBER"
300 INPUT "OR DECIMAL FRACTION? ":F
310 CALL CLEAR
320 PRINT F;"TIMES ORIGINAL RECIPE"::::
330 FOR J=0 TO I-1
340 PRINT F*AMT(J);M$(J);" ";ING$(J)
350 NEXT J
360 PRINT :"CONVERT AGAIN? (Y/N)"
370 CALL KEY(0,K,S)
380 IF K=89 THEN 290
390 IF K<>78 THEN 370
400 END
```

Let's show an example of this program. Key it in then RUN. Remember you need to use decimal fractions.

AMOUNT: 2
MEASURE: CUPS
INGREDIENT: SHORTENING

AMOUNT: 2
MEASURE: CUPS
INGREDIENT: SHORTENING

AMOUNT: 2
MEASURE: CUPS
INGREDIENT: SUGAR

AMOUNT: 2
MEASURE: (just press ENTER)
INGREDIENT: EGGS

AMOUNT: 1.5
MEASURE: TSP
INGREDIENT: ALMOND EXTRACT

AMOUNT: 2
MEASURE: TSP
INGREDIENT: BAKING POWDER

AMOUNT: 4
MEASURE: CUPS
INGREDIENT: FLOUR

AMOUNT: 4
MEASURE: DOZ.
INGREDIENT: ALMONDS

AMOUNT: 0

If you want to triple the recipe, you would next enter 3. Answer CONVERT AGAIN? (Y/N) with Y, and this time try .5 and the recipe will be halved. While someone is keying in this program, another member of the family can try this recipe. It's Grandpa's Almond Cookies. Mix the ingredients together in order (except the almonds), roll in balls, and flatten slightly on cookiesheets. Press one almond on top of each cookie and brush with egg. Bake at 375 for about 10 minutes.

You may use this program as part of a larger program that retrieves the recipe from a file, then asks if you want to convert the recipe. You may want to READ the recipe from DATA statements rather than using INPUT. You can get fancy and print the title and instructions and draw pictures. [Also check out *Micro Bartender* in this book—a program that can be adapted for any recipe file.—Ed.]

Inventory

There are many ways to approach an inventory program. Ten programmers will come up with ten different programs. One possibility is to use the *Electronic Home Secretary* program. Here is one method for a household inventory. Use DATA statements and enter each item in the following order: room number, item, cost.

```
100 REM  HOUSEHOLD INVENTORY
110 FOR I=1 TO 4
120 READ R$(I),C(I)
130 NEXT I
140 CALL CLEAR
150 CALL SCREEN(4)
160 PRINT TAB(9);"INVENTORY"
170 PRINT :::"CHOOSE:"
180 PRINT :"1. LIVING ROOM"
190 PRINT :"2. KITCHEN"
200 PRINT :"3. STUDY"
210 PRINT :"4. WHOLE HOUSE"
220 CALL KEY(0,K,S)
230 IF K<49 THEN 220
240 IF K>52 THEN 220
250 CALL CLEAR
260 CH=K-48
270 CALL SCREEN(C(CH))
280 PRINT "INVENTORY OF ";R$(CH)::::
290 TOTAL=0
300 RESTORE 460
310 READ ROOM,ITEM$,COST
320 IF ROOM=9 THEN 380
330 IF CH=4 THEN 350
340 IF ROOM<>CH THEN 310
350 PRINT ITEM$,"$";COST
360 TOTAL=TOTAL+COST
370 GOTO 310
380 PRINT "TOTAL","$";TOTAL
390 PRINT "DIFFERENT ROOM?(Y/N)"
400 CALL KEY(0,K,S)
410 IF K=78 THEN 430
420 IF K=89 THEN 140 ELSE 400
430 END
440 DATA "LIVING ROOM",15,KITCHEN,8
450 DATA STUDY,10,"WHOLE HOUSE",12
460 DATA 3,COMPUTER,1000
470 DATA 3,DESK,129.50
480 DATA 2,"MICRO. OVEN",450
490 DATA 1,PIANO,5900
500 DATA 3,PRINTER,225.50
510 DATA 2,REFRIGERATOR,425
520 DATA 1,SOFA,425
530 DATA 2,STOVE,525
540 DATA 1,TELEVISION,475.50
550 DATA 3,TELEVISION,150
560 DATA 3,TYPWRITER,300
570 DATA 9,ZZZ,9
```

Only a few items in a few rooms are shown here to illustrate the logic of the program. You will probably want to include more rooms, the year purchased, and perhaps depreciation, replacement value, and a few other remarks. And don't forget to add titles to make the information more meaningful. You can use this program idea for any kind of inventory from food storage to retail products. Extended BASIC allows nice formating of output (with PRINT USING or IMAGE) so the numbers line up. It is also possible in regular BASIC by testing the length or the size of the numbers and printing accordingly.

I entered the DATA items in alphabetical order so they will be listed alphabetically, but you could use a sort routine to alphabetize the items or list the items according to cost. Following is a basic sorting routine:

```
100 REM  SORTING
110 REM  N=NUMBER OF ITEMS
120 DIM A(50)
130 CALL CLEAR
140 INPUT "N=":N
150 IF N>1 THEN 180
160 PRINT :::"N MUST BE LARGER THAN ONE
    , TRY AGAIN PLEASE.":::
170 GOTO 140
180 IF N<51 THEN 210
190 PRINT :::"N MUST BE LESS THAN OR EQ
    UALTO FIFTY (50). TRY AGAIN.":::
200 GOTO 140
210 PRINT
220 FOR I=1 TO N
230 INPUT "A"&STR$(I)&" = ":A(I)
240 NEXT I
250 PRINT :::
260 LIM=N-1
270 SW=0
280 FOR I=1 TO LIM
290 IF A(I)<=A(I+1)THEN 350
300 AA=A(I)
310 A(I)=A(I+1)
320 A(I+1)=AA
```

```
330 SW=1
340 LIM=1
350 NEXT I
360 IF SW=1 THEN 270
370 FOR I=1 TO N
380 PRINT A(I);
390 NEXT I
400 PRINT
410 END
```

You can use this interchange sort algorithm to arrange a list of numbers in *ascending* order. In this example, the user inputs the numbers of items in the list, N, and then enters each number (in any order). For this example, N is limited to 50. The maximum execution time for 50 numbers is about 50 seconds.

Within a FOR-NEXT loop, each number is compared to the next number. If the first number is larger than the second number, those two numbers in the array are switched and SW is set equal to 1 to indicate a switch is made. If the first number is smaller than or equal to the next number, the loop returns to the next pair of numbers to compare.

If SW = 1, at least one switch has been made and the process is repeated with SW reset to zero and the limit LIM of the loop set to the place a switch was made (the numbers after the last switch will be in ascending order with the largest number of the orginal list situated last in the series.)

To change this algorithm to rearrange a list of numbers in *descending* order, simply change the "less than" sign in statement 230 to "greater than." More efficient (and complex) sorts are available for large sets of numbers, but this algorithm is sufficient for smaller sets of numbers.

The alphabetizing algorithm is the same as this interchange sort algorithm with the list of variables changed to string variables. Just change all occurences of A to A$ and AA to AA$. In a regular program the INPUT and PRINT formats would be different from this example.

In the inventory application, we have three variables for each item: room, item name, and cost. These could be read in as arrays and the sort routine would need to interchange all three items. For example, let A(I) be the cost that you are sorting. You would need to add:

```
242 RR$ = R$(I)
244 II$ = ITEM$(I)
252 R$(I) = R$(I + 1)
254 ITEM$(I) = ITEM$(I + 1)
262 R$(I + 1) = RR$
264 ITEM$(I + 1) = II$
```

This coding ensures that the variables associated with each cost are interchanged in the same order as the costs are interchanged. You could also combine the room number and item name into one string variable to be interchanged with the cost variable.

Income Tax

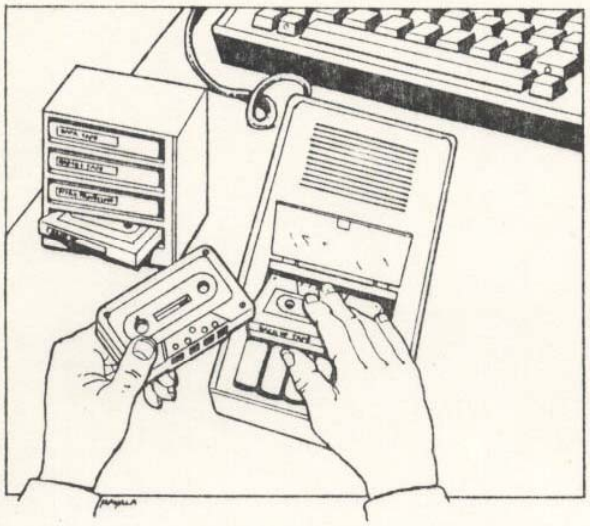
Probably the most common use for the computer when helping out at income tax time is keeping track of expenditures in different deduction categories. You can use the same program idea discussed above in the inventory section, but with a slightly different data structure. Instead of room number, you would use category (medical, interest, contributions, etc.). You would probably still use item and cost, and possibly add the date of expenditure. Your DATA statement would look like this:

```
500 DATA 1, "DR. PAYNE",25.50,"MAY 9"
```

for a medical expense of $25.50 to Dr. Payne on May 9.

I have suggested several ideas to help you get started writing your own programs for your own home, business, or professional applications. Now you just need to DO IT!

Recently someone asked me for a special income tax program— one that would indicate zero taxes to be paid. Hmmmm. I'm still thinking about that program . . . .



A Beginner's Guide

To Cassette Operation With A Home Computer

You bought your TI-99/4A Home Computer because the plug-in Command Cartridges looked like a quick and easy way to get started. You played the games and typed in the programs that you found in the *User's Reference Guide*. Now comes the moment of truth—*What to do next?* The answer, fellow 99'ers, is easy: Learn how to use a cassette tape recorder with your computer so that you can begin to build up a program library by recording and saving the many excellent software programs that are printed in *99'er Home Computer Magazine*.

TI manufactures a special cassette recorder (PHP-2700) for use with its computers that comes supplied with a dual cassette cable. If you cannot locate this recorder, things may get a little complicated. Finding a recorder that provides satisfactory results is not as easy as you'd think. To explain why, I will have to give you a quick background on how a computer talks to a tape recorder and vice versa.

What the Recorder Records

In order to do the wonderful things your computer is capable of doing, *bits* (the "offs" and "ons" that computers use) have to be arranged into patterns. This is true not only for numbers, but also for letters. For example, if you type the letter "A" on the keyboard, your TI-99/4A really sees a pattern that looks like this: off-on-off-off-off-off-off-on. If we think of an "off" as a zero and an "on" as a one, the pattern looks like this: 01000001. Remember that everything your TI-99/4A does is based on groups of *binary* numbers like that. A group of 8 bits is called a byte.

Learning to count in binary is beyond the scope of this article, but there are a number of books or articles around that can teach it to you. What you should know for now is that each letter and character has its own pattern of zeros and ones (its own binary value). For example, the 65th pattern (a *byte* value = 65) represents the letter "A" in the ASCII character coding system used by the TI-99/4A and most computers. This means that 65 is the ASCII value of letter "A." That is why the computer will give you back an answer of 65 if you ask for the value of ASC("A").

In order to read data from a tape recorder, your computer will have to be able to read in bytes of data. That means that it will have to understand "offs" and "ons" when listening to the tape. But, the TI Home Computer doesn't listen to the cassette tape for "off" and "on" sound. Rather it listens to two different frequency tones that represent the two states.

Not all Recorders Are Equal

It appears generally true that it takes more power for a cassette tape recorder to produce or reproduce a high frequency than it does to produce or reproduce a lower frequency tone. If the volume is not high enough during either recording or playback, your computer won't hear anything, or it might not be able to hear the higher frequency tone. In order to help the TI-99/4A hear the high frequency tones properly, the tone control knob on the recorder should be set at or near the maximum level. Even if this is done, some tape recorders cannot handle the high frequency. If your recorder doesn't have a tone control, there's a good chance it was meant to handle only the frequencies of human speech and won't be mechanically able to handle the high frequency tone at all.

Since it is possible that your recorder cannot reproduce the high frequency tones properly, your computer has to be sure that it has *read* all the data. How can it be sure that nothing was lost? Your computer counts the number of "ons" that it heard. After every so many bytes, it expects to read a number on the tape. This number tells the computer how many "ons" it should have read. If the two numbers don't match, a *parity error* has occurred and the computer will tell you that you have a problem.

Now suppose that the volume is set high enough to reproduce the high level tones, but is up *too* high? Well, too much volume causes distortion in a tape recorder. This distortion will mean that some of the tones will not be heard accurately by the computer at all. It's just as if someone screamed in your ear. You know something was said, but you don't know what it was.

A Remote Possibility

There is one additional problem that may crop up even with tape recorders that satisfy the above criteria: Almost all cassette recorders have a remote control jack which allows you to stop the recorder by pressing a button or switch located on the microphone. Unfortunately, since this jack is meant to work with the manufacturer's own microphone, there is no guarantee that the jack is hooked up the same way in each tape recorder. In fact, there is a 50-50 chance that a non-TI tape recorder model you may buy or already own will not be compatible with the Home Computer system. This means that the drive motor of your recorder might not be capable of being turned on and off *automatically* by the computer when the plug on the TI cable is inserted into the recorder's remote jack.

Luckily, if this is true for your recorder, Emerald Valley Publishing Co. sells an inexpensive adapter (called "TEXSETTE") which is used between your recorder and the TI cable. If you don't want to spend the money for this adapter, you can get by without it by *manually* starting and stopping the tape [except if you intend to use cassette data files, in which case the automatic operation is necessary.—Ed.].

The conclusion you can draw from all this is that your TI-99/4A requires a tape recorder with specific attributes in order to consistently guarantee good results. If you do not already own a recorder, I strongly suggest that you buy the model PHP2700 tape recorder from Texas Instruments. If you do have a recorder, you should try it out before incurring the expense of purchasing a new one.

Plugging In!

Now that we have discussed why some recorders won't work at all or won't work with the remote control jack plugged in, let's get down to business. Shut off your machines and plug the wide connector (with 9 holes in it) into the back of your computer. The other end of the cable has two cords. One cord has three plugs attached (labeled plug #1), and the other (plug #2) has only two. The tape recorder that you connect to plug #1 will be called "CS1" by the computer. If you are lucky enough to have a second usable tape recorder, you can hook up that one to plug #2. It will be called "CS2" by the computer. Just follow the installation instructions printed on the card that came with the TI cassette cable. If your tape recorder does not have a remote control jack, just ignore the instructions to insert the black plug. Note that CS2 does not have a playback plug. You can only record on CS2.

Plug the tape recorder into an electrical outlet and you are now ready to check out your system. [A battery-operated tape recorder is usually too unreliable for recording and playing back data for your computer because of the possible fluctuations in speed and amplifier gain over the life of the battery.—Ed.] Load a high-quality (remember we have to record those high tones accurately!) C-10, C-15, or C-30 blank tape into the tape recorder. The number part of the tape code gives the number of minutes of recording time available on both sides of the tape. A C-10 tape has 5 minutes of recording time on each side. You can use a tape as long as a C-60, but never anything longer. This is because longer tapes are thinner, stretch more, and may not maintain proper speed in the recorder. For this first test, make sure the tape is completely blank. Turn on your computer and select TI BASIC. Key in the following 4-line program:

```
100 PRINT "HELLO"
110 I=30
120 PRINT "MY VALUE IS";I
130 END
```

Turn up the volume on your TV (or monitor) by a few notches so that you can hear a slight hum. Set the volume control on your tape recorder midway between the lowest and highest settings. Set the tone control (if there is one) up to maximum. [Or, if you are using the TI PHP2700, follow its manual's setup instructions.—Ed.] Now type in SAVE CS1 and press the ENTER button. Follow the instructions that the computer gives you to rewind the tape and begin recording. When you press "record" on your tape unit and then press the ENTER button on the computer, the tape should start moving.

If the tape doesn't start moving, you have a non-compatible remote control jack. If this is the case, wait for the computer to leave recording mode and print the "VERIFY (Y/N)" message. When it does, type in an "N". Now remove the plug from the remote control jack and begin the recording process all over again (by typing SAVE CS1 and pressing the ENTER button). When you are told to record, you should now see the tape moving.

Getting Adjusted

After a short pause, you will actually hear your program being recorded onto the tape. The recording consists of an initial long phrase of a single tone, followed by bursts of sound with a very short pause between bursts. The initial tone is used to tell the computer on playback that data is coming. This tone is recorded before each program and each block of data (which we will talk about later). When the recording is over, you will get the verify message (see above). Type in a "Y" (you don't have to press the ENTER button). Follow the instructions about rewinding the tape. When you play back the tape, listen to the sounds that it is making. Note that the volume is much louder than when you recorded. If that initial tone does not sound pure (if it seems to warble, with the tone going higher and lower), you are probably using a recorder that won't work well consistently. If the tone does seem pure, you're halfway home!

When the tape goes silent, the program has finished loading. You should get a message that says either "DATA OK" or "ERROR IN DATA". If no message prints, then the volume setting was too low and your computer is still waiting for the first recognizable byte of data. It will eventually get tired of waiting and give you a "NO DATA FOUND" error. Just wait for this message to appear, or shut off your computer and start all over again.

If you got the "DATA OK" message, you are home free! Relax and go on to the next paragraph. If you were unlucky enough to get a "NO DATA FOUND" error, turn up the volume one notch. Write down the latest notch on a piece of paper. In either case, respond to the computer question by entering an "R" to re-record. The computer will guide you in another recording session. Keep repeating the process until you can't change the volume any further, or the "DATA OK" message appears or the error message has changed (i.e., from "NO DATA FOUND" to "ERROR IN DATA"). If you can't change the volume any further, your recorder just isn't good enough. Don't aggravate yourself any longer—go out and find somewhere to buy the TI recorder. If the "DATA OK" message has appeared, you are in good shape. If the message has changed, back off your last change by half a notch. For example, if moving the control from 6 to 7 made the "ERROR IN DATA" message appear, try the recording process again at 6½. If that doesn't work, try it at ¼ notch intervals. If *that* doesn't work, forget it. Buy a different recorder.

After you get the "DATA OK" message, mark the volume setting in some way. I usually dip a toothpick in white paint (a light nail polish will also work) and dab

a line on both the recorder and the control so that I can easily see that the volume setting is correct. You now have a functioning cassette tape system and are ready for bigger and better things.

Better Safe than Sorry

When you entered the SAVE CS1 command, you told the computer to copy the bytes that represented your program inside the computer onto a tape. The entire program is saved each time. Your program is still in the computer, however. If you agree to verify your tape, TI BASIC will read the data from the tape and compare it in a byte-for-byte manner with the program still residing in memory. Unless the two match perfectly, your 99/4A will issue a warning that you have a bad tape. ALWAYS VERIFY ANY SAVEs BEFORE ENDING A PROGRAMMING SESSION!

The tape version of the program is saved in a "machine image" format that is meaningful only to TI BASIC. You cannot, however, write a TI BASIC program that will read this tape. The only way to get your program back into the 99/4A is via the OLD CS1 command. This will load the program back into the machine. Anything that may have been in the computer before the OLD CS1 will be lost. By the way, you can SAVE CS2 (if you have a recorder hooked up to cable #2) and then read in the tape by entering OLD CS1. Of course, you have to move the tape over to the recorder attached to cable #1 first!

The instructions built into the TI-99/4A whenever you enter the SAVE CS1 or OLD CS1 command assume that you have only one program per side of tape. A long program will require about 3-4 minutes of recording time. This means that it is possible to save about 4-5 programs on each side of a C-30 tape. If your recorder has a tape counter, just keep track of where the next free space on the tape is located. Then, when the computer tells you to rewind the tape, just fast-forward to that next free spot on the tape instead. Make sure to keep a log of what programs are recorded on tape and where they are located. [If you don't want to be bothered by this, and want maximum reliability, it is better to use C-10 cassettes and record only *one* program per side.—Ed.]

A cassette tape recorder will usually have the ability to record a new program directly over an old one. It is good to get into the habit of completely erasing a tape, however, when you no longer need it. This ensures the best possible recording the next time you use the tape.

Filing Data

The cassette recorder also makes a handy data storage device for use in your computer programs. Suppose that you have written a program to keep track of the bowling scores and figure out the handicap of each member of your bowling league. You don't want to re-enter this information each time you run your program. What you need is a way of saving the data when you are through with it so that it can be read in the next time around. Some people do this by entering the information in DATA statements each time before SAVEing the program. A better way of doing this is to write out a small *file* of data onto tape. Your program can then read in this data file the next time it runs. TI BASIC has an easy way of doing this by using the INPUT # and PRINT # statements.

Before you can read or create a file, you must tell the computer a little about your file. This is done by the OPEN statement. Your reference manual does a pretty good job of explaining this statement, so I'll just go over the parts specifically dealing with cassette tape files.

Unlike the SAVE command which writes out your entire program as a large "chunk" of data, BASIC data files can only handle small chunks of data, called *records*, at a time. Each file can contain 1 or more records. All cassette records in a file must be of the same size. They can all be 64 bytes (characters) long, 128 bytes long, or they can all be 192 bytes long. You can specify other lengths as part of the OPEN statement, but TI BASIC will boost the number up to either 64, 128, or 192. If a record you want to write is shorter than the length that you specify, TI BASIC will add enough blanks at the end of the record to make it the right length.

Each record can contain as much data as you can fit in a record of that size. When you have a statement that uses PRINT # and ends with a semi-colon, BASIC will add that data to the record, but will *not* write anything out to the tape. When BASIC sees a statement with PRINT # that *doesn't* end with a semi-colon, it will write out everything in a record (including this last piece of data) to the tape. When the record is written to tape, it is preceded by the steady high-pitch tone that starts off a SAVE. That means that BASIC uses a lot of tape to write a single record. In fact, if you use records that are only 64 bytes long, it is possible that more room is spent on the tape for the start tone then is used to record the data! Remember that more room on the tape means slower reading by the computer. That's why I usually use 192 byte records and try to fit as much data as possible into each record. Doing this will cut down on the number of records written to tape, and make the program run faster.

Because TI BASIC only writes to tape when you tell it to, the computer must have total control of the cassette recorder so that it can start and stop the recorder as needed. This means that the black remote-control plug must be inserted (and functional!). If your remote jack is not compatible with the TI-99/4A, you will *not* be able to use the recorder for saving and reading data under program control.

You can store in two different formats. DISPLAY format means the data is saved just the way it would look in a DATA statement. INTERNAL format saves the data in the same way that the computer stores the information internally. Numbers require 8 characters (bytes). Strings (i.e., names) require 1 byte (for the length) plus the data itself. I usually save my data in INTERNAL format so that I know the length needed for numbers no matter how big or small they are.

THE BASICs of Record Keeping.

Let's write a part of a program that will save each bowler's name, his pin average and his handicap. Pretend that we have 60 bowlers in our league. If we restrict each bowler's name to a maximum of 45 characters, we will need a total of 62 bytes per bowler (45 bytes + 1 = 46 for the name + 8 for the average + 8 for the handicap = 62). We can therefore fit the data for 3 bowlers into one 192 byte record. (See Listing 1.) If you have not

filled up a record by the time the program hits the CLOSE statement, TI BASIC will fill out the record with blanks and write it out. You do not have to worry about writing out a last record that is partially full. Just remember always to program in a CLOSE statement. To read the data file into your program, you need program instructions that almost duplicate the write program (see Listing 2, below).

When your program executes the OPEN statements, the computer will issue commands about rewinding the tape and pressing ENTER. When INPUTing from tape, the screen will scroll up one line to indicate that it has begun processing the tape just before it reads the first record.

Once you have these basic components working and understood, you will probably wish to embellish them with things like update capability, printing of bowlers' statistics, etc.

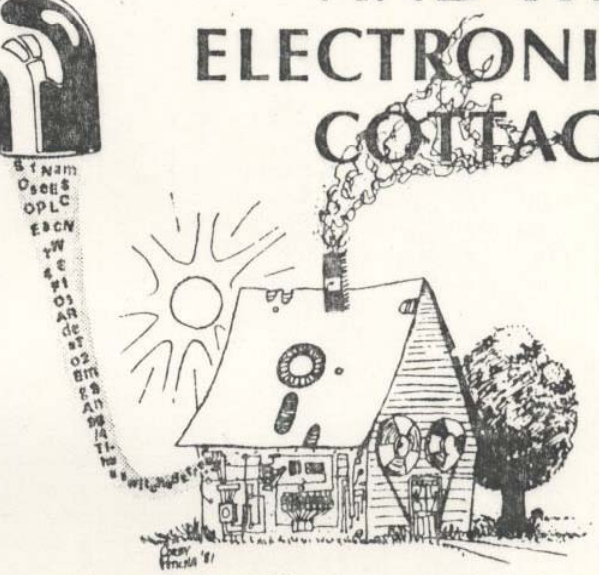
I have often been asked why TI provides the CS2 plug. I have to admit that most manufacturers do not provide dual cassette support. It is useful if you must process more data in your program than the computer can handle inside its memory. You would need *two* recorders hooked up, and would read in as much data as possible (for example, as file #1) on CS1, then do whatever you have to, and finally write the updated data out on CS2 (as a different file number). You would then go back and read in the next batch of data from CS1, update it, and write it out. You repeat this until there is no more data on CS1. This allows a *small* computer to handle very *large* files.

At this point you should have the basic knowledge for choosing a cassette recorder, and getting it to work with your computer. Keep in mind that tape storage transforms your Home Computer into a very powerful and versatile machine. And once you get familiar with the few simple procedures and precautions, each occasion of saving and loading programs and data files will become second nature. . .one might even say, "filled with memories . . . ."

```
100 REM  *********************
110 REM  *  LIST 1 CASSETTE OPER.  *
120 REM  *********************
130 REM  ROOM FOR 60 BOWLERS NAMES, A
    VERAGES, HANDICAPS
140 DIM B_NAME$(60),B_AVG(60),B_HANDI(
    60)
150 REM  ****** INITIALIZE NAME ARRAY
160 FOR I=1 TO 60
170 B_NAME$(I)="EMPTY
    "
180 NEXT I
190 REM  ****** INPUT BOWLER DATA LOO
    P
200 REM
210 FOR I=1 TO 60
220 CALL CLEAR
230 PRINT "INPUT DATA  FOR BOWLER #";
    I::::::
240 INPUT "NAME?  ":NAME$
250 REM  IF BLANK NAME, EXIT LOOP
260 IF NAME$="" THEN 390
270 REM  SET NAME LENGTH TO 45 CHARAC
    TERS
280 NAME$=NAME$&"
                                    "
290 B_NAME$(I)=SEG$(NAME$,1,45)
300 PRINT
310 INPUT "AVERAGE?  ":B_AVG(I)
320 PRINT
330 INPUT "HANDICAP?  ":B_HANDI(I)
340 PRINT
350 NEXT I
360 REM  ***** END OF INPUT FROM KEYBO
    ARD
370 REM  **********************
380 REM  ***** BUILD RECORDS AND WRITE
     TO TAPE
390 CALL CLEAR
400 PRINT "  <<< BUILDING RECORDS >>>":
    ::::::
410 REM  OPEN FILE FOR OUTPUT
420 OPEN #1:"CS1",OUTPUT,INTERNAL,SEQU
    ENTIAL,FIXED 192
430 FOR I=1 TO 60 STEP 3
440 REM  PRINT THREE SETS OF DATA PER
     RECORD
450 PRINT #1:B_NAME$(I);B_AVG(I);B_HAN
    DI(I);
460 PRINT #1:B_NAME$(I+1);B_AVG(I+1);B
    _HANDI(I+1);
470 PRINT #1:B_NAME$(I+2);B_AVG(I+2);B
    _HANDI(I+2)
480 NEXT I
490 CLOSE #1
500 REM  ***** END OF WRITE TO TAPE SE
    CTION
510 END
```

```
100 REM  *********************
110 REM  *  LIST 2 CASSETTE OPER.  *
120 REM  *********************
130 REM  ROOM FOR 60 BOWLERS NAMES, A
    VERAGES, HANDICAPS
140 DIM B_NAME$(60),B_AVG(60),B_HANDI(
    60)
150 REM  ****** INPUT BOWLER DATA F
    ROM TAPE
160 REM
*170 CALL CLEAR
180 PRINT "  <<< READING RECORDS >>>":::
    ::::
190 REM  OPEN FILE FOR INPUT
200 OPEN #1:"CS1",INPUT,INTERNAL,SEQU
    ENTIAL,FIXED 192
210 FOR I=1 TO 60 STEP 3
220 REM  READ THREE RECORDS FROM THE
     TAPE
230 INPUT #1:B_NAME$(I),B_AVG(I),B_HAN
    DI(I),
240 INPUT #1:B_NAME$(I+1),B_AVG(I+1),B
    _HANDI(I+1),
250 INPUT #1:B_NAME$(I+2),B_AVG(I+2),B
    _HANDI(I+2)
260 NEXT I
270 CLOSE #1
280 REM  ****** END OF READ FROM TAPE
    SECTION
290 REM  **********************
300 REM  ***** DISPLAY ON SCREEN SELEC
    TED BOWLER
310 REM
320 CALL CLEAR
330 PRINT "INPUT BOWLER NUMBER":::
340 INPUT A
350 IF (A<1)+(A>60)=-1 THEN 320
360 CALL CLEAR
370 PRINT "BOWLER'S NAME --"
380 PRINT B_NAME$(A)
390 PRINT :::"AVERAGE --";B_AVG(A)
400 PRINT :::"HANDICAP --";B_HANDI(A):
    :::
410 INPUT "ANOTHER? (Y/N) ":B$
420 IF B$="Y" THEN 320
430 END
```

INFORMATION UTILITIES AND THE ELECTRONIC COTTAGE



". . . we find more and more companies that can be described . . . as nothing but 'people huddled around a computer.' Put the computer in people's homes, and they no longer need to huddle."

—The Third Wave
By Alvin Toffler

In his book, *The Third Wave*, Alvin Toffler presents a powerful argument that ". . . our biggest factories and office towers may, within our lifetime, stand half empty . . . . This is precisely what the new mode of production makes possible: a return to *cottage industry* on a newer, higher, electronic basis, and with a new emphasis on the home as the center of society." Toffler goes on to single out many powerful socio-economic forces that are presently fueling this transition and points to the software production industry which has already set an early example as the fastest growing cottage industry of the 1980s.

Within the last three years, the microcomputer community has been witnessing the unfolding of an extraordinary event. I say "extraordinary" not because of what has already happened, but rather, for what it portends for the future. What is this event, and what great significance does it hold? Quite simply, the event has been the birth and maturation of "information utilities"—a significant event because of their awesome potential to speed up Toffler's timetable and change the way most of us live and work *within this current decade!*

There's certainly nothing mysterious about utilities. All of us are already familiar with telephone, electric, water and gas utilities. These are necessary and valuable resources delivered to and consumed in the home. If we now add *information* to this list, we create an "information utility"—a service that brings information to a place where the general public can access it and put it to use. . .and where the cost of packaging and delivery is *shared* by the consuming public. And what better, more convenient place is there for the general public to consume this information than in the home—the forthcoming "electronic cottage."

The New Timesharing

Timesharing, the foundation of all information utilities, is certainly not new. It was originally developed to serve the needs of business by providing companies with access to computer power without them having to buy expensive data processing equipment. Custom programming and technical assistance were available at extra cost to those who couldn't use the "canned programs."

What these information utilities have done is add a new wrinkle to the traditional timesharing concept. Using the famous "baking soda technique"—whereby a producer of this unglamorous age-old product continually dreams up and advertises new uses for it—they have repackaged timesharing to make it palatable to a much greater potential market. But lest you jump to the wrong conclusion, I should point out that these utilities are *not* simply pushing an old service to a new market. Rather, what we really have here is the creation of an *entirely new dimension* to timesharing—an attempt to satisfy a mass audience with extremely diverse needs and wants. . .and do it at an *affordable* price.

Information Services for the Masses.

To provide you with some appreciation for the great diversity of presently available information services, let's take a brief look at one of the largest, fastest growing utilities, *The Source* (a service mark of Source Telecomputing Corporation, a subsidiary of The Reader's Digest Association, Inc.) At present, The Source offers over 1,200 services in areas such as:

(1) computer-based message services
(2) proprietary databases
(3) business and professional applications packages
(4) personal and corporate services
(5) consumer purchasing
(6) entertaiment
(7) education
(8) "classical" timesharing

All these services enter a subscriber's home or business through existing telephone lines (using the *packet-switching*

networks of Telenet and Tymnet.) A local number is available in over 360 U.S. cities for accessing The Source. A subscriber types in (on a computer terminal connected to the telephone line, or a self-contained microcomputer with appropriate software to emulate a terminal) his or her private ID account number, and then chooses from a menu of services. Since subscribers can command the "host" computer in plain English (in a somewhat abbreviated form), very little instruction is necessary to do meaningful things—an extremely important attribute of any information utility.

Although an information utility such as The Source hopes, in the not-too-distant future, to be able to feed millions of inexpensive computer terminals in U.S. households, its present subscriber base is drawn from the business community and a small segment of the vast consumer community—the segment which presently owns microcomputers.

It's not surprising that businesses of all types are attracted to very inexpensive services such as electronic mail, travel arrangement, applications software packages, programming access to mainframes, and business/industry news. It does, however, take some stronger incentives to lure the consumer segment of the microcomputer community—the present-day pioneers who purchased their micros for home use. It's to this group that information utilities like The Source must ultimately cater if they hope to eventually reach the economy of distribution and substantial return-on-investment that are possible in a mass market.

To this end, consumers with microcomputers are presently being wooed with a rapidly expanding array of personal services (such as bookkeeping, correspondence, travel arrangements and keeping track of investments), educational programs, home economics assistance, plus activities and information that the *whole* family can use—especially games, movie and product reviews, news and sports reports.

The TEXNET Turn-On

If having the services and activities of The Source in your home isn't exciting for you, how about having it together with the following package of special enhancements: color graphics and animation, music and sound effects, a software exchange with hundreds of free programs plus state-of-the-art synthetic speech—*actually "spoken" to you!* No, all this isn't just a "wouldn't-it-be-great-if" speculation of things to come, but rather embellishments to the basic Source menu.

The special services and enhancements I've been describing are available to users of the Texas Instruments TI-99/4A microcomputer, and come under the TEXNET (a service mark of Texas Instruments, Inc.) umbrella. Besides the microcomputer, the only additional items that are needed to take advantage of *all* of the special TEXNET features are an RS232 Interface and modem (for establishing a compatible telephone connection), a plug-in *Terminal Emulator II* Command Cartridge (the software for the micorcomputer), and the plug-in Solid State Speech Synthesizer—the Texas Instruments peripheral that "voices" the synthetic speech. The synthesizer won't be necessary if speech capability isn't desired.

Just how, exactly, are TEXNET and The Source related? According to Craig W. Vaughan (President, Software Sorcery, Inc.), a systems support consultant to Source Telecomputing Corporation and Texas Instruments, TEXNET *appears* to encompass The Source totally. That is to say, TEXNET subscribers have access to everything Source subscribers do *plus* additional special services that require the Texas Instruments Home Computer for access and use. Graphically, it would appear like this, with the outer ring of TEXNET including everything within The Source's inner ring, and expanding its own outer ring of special services over time. This is only an *appearance,* however, as Vaughan pointed out; "In reality, TEXNET users will be running a *shell program* . . .on The Source system."

Services on TEXNET fall into two major groups: (1) directory or lookup textual information, and (2) interactive or transfer services. In this first group there will be a product and technical newsletter (TI News), TI Software Directory, TI User Groups, TI Service Centers, and TI Phonetic Dictionary (helpful when programming with text-to-speech). The second group of services is really what TEXNET is all about. First, there are the *transfer* services. Sophisticated error-checking software in the *Terminal Emulator II* Command Cartridge will permit any of hundreds of user programs from the TI Software Exchange to be *downloaded* correctly into another user's system. Eventually, we can expect to see on TEXNET the capability for *direct* uploading and downloading between users. The TI Graphics Library and TI Music & Sound Library will work the same way: A TEXNET subscriber will be able to download the color graphics, musical scores, and sound effects into his own system for later use in his own programs.

The *interactive* services on TEXNET are really speech enhancements of services already available on The Source. For example, the electronic mail service—probably the most highly used service, and reason enough for many to be Source subscribers—is made even more intriguing when you mail is "read" to you by your machine's electronic voice. And if "electronic voice mail" intrigues you, wait till you experience TI Voice Chat: TEXNET users will be able to participate in "spoken"interactive communication, CB-style. Well almost . . . . What actually happens is that one user types in something, and the words get converted back into synthetic speech on the other end; the typed-in reply gets sent back, and then also gets converted to speech. So what we actually wind up with is a real-time verbal conversation *between two speech synthesizers!*

There's one short paragraph in the latest Source brochure that perfectly sums up what's presently happening in the world of information utilities:

> *"This brochure is obsolete. By the time you read this brochure, new information and communication services will have been added to The Source. Old data bases will have been updated, and streamlined "userfriendly" access procedures introduced."*

Without a doubt, it's an exciting time to be living and learning along the new information frontier.

For more information on TEXNET and the Source, see your TI dealer or contact The Source Telecomputing Corp., 1616 Anderson Rd. McLean, Virginia 22102.

DATA COMMUNICATIONS & the TI-99/4A

If you have invested in an RS232 interface and a modem in addition to your TI-99/4A system, you have the possibility of tapping a vast information network through existing and planned computer time-sharing services. A variety of information services such as news, financial information, computer games, various data bases, and program exchange, to name just a few, are provided through information utilities such as *The Source* (by Source Telecomputing Corporation). TEXNET, a collaboration between Source Telecomputing Corporation and Texas Instruments, will enhance data base services with the addition of text-to-speech, color graphics, and music. This service is available exclusively to users of the TI-99/4A. TEXNET and The Source are covered in another article. [See "Information Utilities & the Electronic Cottage."—Ed.] This article is an examination of basic data communications between the TI-99/4A and other computers.

Data Communications Concepts

A number of *coding schemes* have been devised to represent characters in order to input information into a computer. The most widely used code is the American Standard Code for Information Interchange—more commonly known as ASCII code. It is a 7-bit code which can represent 128 character configurations. Figure 1 illustrates the bit patterns associated with each of the characters. An eighth bit, called a *parity bit*, is commonly included in the ASCII code. The parity bit used to detect errors in the bit stream which might be due to the reading or transmission of the data. Parity of a ASCII coded signal can be odd or even. An ASCII code with even parity must contain an even number of ones; for an odd parity the number of ones must be odd (i.e., 1, 3, 5, 7). The Texas Instruments *Terminal Emulator II (TE-II)* Command Cartridge enables you to tailor your TI-99/4A to fit the characteristics of the remote computer system. With the communications device menu, you can specify the parity of the received or transmitted signal—odd, even or none (no parity bit)—and set the number of data bits at 7 or 8.

Figure 1 **ASCII**

MOST SIGNIFICANT OCTAL DIGIT	LEAST SIGNIFICANT OCTAL DIGIT 000	001	010	011	100	101	110	111
0 000	NUL	SOH	STX	ETX	EOT	WRU	RU	BEL
0 001	BS	HT	LF	VT	FF	CR	SO	SI
0 010	DLE	DC1	DC2	DC3	DC4	MAK	SYN	ETC
0 011	CAN	EM	SUB	ESC	FS	GS	RS	US
0 100	SP	!	"	#	$	%	&	'
0 101	(	)	*	+	,	-	.	/
0 110	0	1	2	3	4	5	6	7
0 111	8	9	:	;	<	=	>	?
1 000	@	A	B	C	D	E	F	G
1 001	H	I	J	K	L	M	N	O
1 010	P	Q	R	S	T	U	V	W
1 011	X	Y	Z	[	\	]	↑	←
1 100	_	a	b	c	d	e	f	g
1 101	h	i	j	k	l	m	n	o
1 110	p	q	r	s	t	u	v	w
1 111	x	y	z	{	|	}	~	DEL

NUL	Null or tape feed (control shift P)
SOH	Start of heading (control A)
STX	Start of text (control B)
ETX	End of text (control C)
EOT	End of transmission (control D)
WRU	Enquiry (control E)
RU	Acknowledge (control F)
BEL	Ring bell (control G)
BS	Backspace (control H)
HT	Horizontal tab (control I)
VT	Vertical tab (control J)
FF	Form Feed (control L)
CR	Carriage Return (control M)
SO	Shift out (control N)
SI	Shift in (control O)
DLE	Device link escape (control P)
DC1	Device control 1 (control Q)
DC2	Device control 2 (control R)
DC3	Device control 3 (control S)
DC4	Device control 4 (control T)
NAK	Negative acknowledge (control U)
SYN	Synchronous idle block (control V)
ETC	End of transmission block (control W)
CAN	Cancel (control X)
EM	End of medium (control Y)
SUB	Substitute (control Z)
ESC	Escape (control shift K)
FS	File separator (control shift L)
GS	Group separator (control shift M)
RS	Record separator (control shift N)
US	Unit separator (control shift O)
SP	Space
DEL	Delete, rub out.

The actual number of bits transmitted is larger than the number of bits in the code. "Housekeeping" bits are added both before and after the bits which represent the character code. The additional bits are called *start* and *stop* bits. A single bit is added at the front of the code as a signal to advise the receiving device to start sampling the incoming signal. Stop bits, added after the character code, indicate when the code is finished, and reset the device for recognition of the next start bit. For an ASCII coded character 11 or 12 bits are typically transmitted.

In data communications terminology, a *full duplex* channel implies that information can flow in two directions simultaneously. On a *half duplex* channel, the information can flow in both directions, but not simultaneously. If you select the half duplex mode from the *TE-II* communications device menu (and set the modem accordingly), the characters you send will be "echoed" back to your monitor or TV set, and appear on the screen. The echoed or extra character does not occur if full duplex is selected.

The public telephone network can provide means of communication from your TI-99/4A to another computer or information service. The information or bit stream that your computer sends and receives, travels serially through the network. That is to say that the bits making up a character are sent and received one after another.

There are a variety of modes of serial data transmission. Your modem transmits data *asynchronously*. This means that any set character is sent independently of any other character, and that the character bits are preceded by a start bit and followed by at least one stop bit. *Synchronous*

transmission requires that both the sending and receiving modems are synchronized by a clock signal. The rate at which data is transmitted (or received) is termed the *baud rate*. The formal definition of a baud is the reciprocal of the length of the shortest pulse used to create a character. Since all the bits of the ASCII code are equal in length, the terms "bits per second" and "baud" can be used interchangeably. A baud rate of 110 requires a minimum of 2 stop bits; at 300 baud and higher a minimum of 1 is required. The *TE-II* software allows you to choose between two baud rates (110 or 300), and your modem usually limits your use to either 110 or 300. The RS232 interface (without the *TE-II*) also allows you to use baud rates of 1200, 2400, 4800 or 9600. The higher rates can be used to output data to a printer or to send data to another TI-99/4A connected directly to your system.

The function of your modem is to convert the binary pulse train (1s and 0s) from your computer to some form of analog signal (tones) that can be transmitted over a telephone line. You will note that in the transmit mode your modem emits a continuous tone. This tone is called the carrier signal. When sending data from your TI-99/4A, the modem's function is to *modulate* (vary the amplitude or frequency) this carrier signal. It also works in the opposite direction by *demodulating* the carrier, so that the ASCII code sent to your TI-99/4A can be properly interpreted. Thus, the term "modem" is derived from the two words which describe its function: MOdulation and DEModulation. A common modulation technique is called frequency shift keying (FSK). This technique converts the binary pulses from the computer to two tones of different frequency. For example, if the carrier signal has a frequency of 1500 Hz, a 1 might be transmitted at 2000 Hz and a 0 at 1000 Hz.

Terminal Emulator II Command Cartridge

The *TE-II* Command Cartridge implements all 128 characters of the standard ASCII code, which is illustrated in Figure 1. It's also possible to send any standard ASCII *control characters* (used for signaling a remote computer or device to perform a predefined function), and display lines containing more than 40 characters by "wrapping" the extra characters onto a second line. The most powerful feature of the *TE-II* is the ability it gives users to store received data on tape or disk. You can review this data after logging off the remote computer, and can also send it to a printer or another computer.

Listing 1

```
100 REM  ** DISPLAY FILE **
110 REM  READS IN A DISPLAY FILE AND O
    UTPUTS IT TO THE TV
120 INPUT "ENTER FILE NAME: ":FNS
130 OPEN #10:"DSK1."&FNS,FIXED 80,DISP
    LAY
140 REM  USE <LINPUT> BELOW IF EXTENDE
    D BASIC
150 INPUT #10:XS
160 PRINT XS
170 IF EOF(10)THEN 190
180 GOTO 150
190 CLOSE #10
200 GOTO 200
```

The format of the data stored by the *TE-II* is ASCII (display format) and is of fixed record length of 80 bytes (characters). In order to make further use of the information, it is necessary to write programs using BASIC. A simple example of such a program is shown in Listing 1. Line 130 opens a saved disk file using the OPEN statement. The following line inputs an ASCII character string; if the record denotes the end of file (EOF), the program ends. Otherwise, the program returns to the INPUT statement (line 140) and continues to read the data file until an EOF is detected.

Data Communications Using BASIC Programs

Display format files can be sent from your TI-99/4A to another computer under control of the BASIC listing shown in Listing 2. The program assigns file number 1 to the indicated disk filename, and file number 2 to port 1 of the RS232 interface. Each record or character string is input from the disk and then transmitted to the remote computer. Of course, this assumes that a means of recording this data is resident on the remote computer. This program can be used, for example, to efficiently transmit a pre-recorded message, or text file to another Home Computer.

The program listings in Listing 1 and 2 have a common flaw: If a display format file contains commas, the character string will be terminated by the first comma encountered. This is due to the fact that the BASIC INPUT statement interprets a comma as a separator between character strings or data items in display format data. (See page II-126 of the *User's Reference Guide.)* This flaw can be overcome by using the LINPUT statement from Extended BASIC.

Listing 2

```
100 REM  ** TRANSMIT FILE **
110 REM  TRANSMIT FILE TO REMOTE COMPU
    TER
120 REM
130 CALL CLEAR
140 INPUT "ENTER FILE NAME: ":FNS
150 REM  THE FOLLOWING <OPEN> MUST BE
    MODIFIED TO MATCH THE STRUCTURE OF
     YOUR FILE
160 OPEN #1:"DSK1."&FNS,VARIABLE 192,D
    ISPLAY
170 REM  THE FOLLOWING <OPEN> MUST BE
    MODIFIED TO MATCH THE COMPUTER TAL
    KED TO.
180 OPEN #2:"RS232.DA=8.NU",VARIABLE,O
    UTPUT,DISPLAY
190 REM  USE THE <LINPUT> BELOW IF EXT
    ENDED BASIC
200 INPUT #1:XS
210 PRINT #2:XS
220 IF EOF(1)THEN 240
230 GOTO 200
240 CLOSE #1
250 CLOSE #2
260 END
```

TEXT-TO-SPEECH ON THE HOME COMPUTER

Go ahead—shout something at your Home Computer. . .but don't be surprised if it answers you back! Welcome to the exciting world of talking computers—a world in which synthetic speech will very soon cease being a novelty, and will instead become instrumental in the everyday interactions between humans and machines.

If you have a Texas Instruments Home Computer, you're one jump ahead of everyone else in taking advantage of this revolutionary communications tool. All you need additionally is the Speech Synthesizer peripheral, and the plug-in *Terminal Emulator II* Command Cartridge. Text that you type on the console keyboard will be converted to synthetic speech, and "spoken" through your TV set or monitor. There's no fixed vocabulary to constrain you, and personal phrases can be called up under program control through the TI BASIC computer language.

But this is only the beginning. . . . If you connect TI's RS-232 interface and a modem to this configuration, you can have access (through your telephone) to the electronic mail, database, entertainment, and computing facilities of the SOURCE and its offspring, TEXNET. The TEXNET service allows TI-99/4A users to access all the menu selections from its parent information utility plus some additional features *with the enhancements of text-to-speech, sound effects, music, and color graphics!* Imagine a weather report with a color graphic representation of a bright sun being blotted out by ominous looking rain clouds, while "Stormy Weather" is being played in the background, and the temperature, wind, humidity and other vital statistics are flashed on the screen and recited to you by your speech synthesizer—an exciting prospect at the least.

Linear Predictive Coding (LPC)

When Texas Instruments made the first single-chip speech synthesizer in 1978, its original application was in their *Speak & Spell* learning aid. The chip, A TMS5100, is essentially an electronic model of the human vocal tract (a mathematical model implemented as a filter network) that produces speech through a technique known as *Linear Predictive Coding* (LPC). There have been other approaches to speech storage-methods employing digitized speech and pulse-coded modulation—but these result in very high data rates (64,000-100,000 bits per second). And the higher the data rate, the fewer words of speech the available memory can hold.

The value of TI's LPC technique is its modest memory requirement: It provides speech quality nearly comparable to these other methods, at a much lower data rate (1,200 bits per second). For example, a speech reproduction of the words "Texas Instruments" requires approximately 90 times as many bits using digitized speech techniques as it requires with LPC.

What is the secret to LPC's economy of storage? The "P" in the middle that stands for "Predictive." Here's how it works: A speech waveform is originally sampled and encoded. This data is used to calculate the coefficients of the linear equations of the digital filter network that will control the "shape" of this synthetic vocal tract. When excitation noise (a chirp function and white noise generator) is applied to this filter network, the circuitry produces a simulation of the resonant effects of the mouth and nasal cavities.

Allophones

This is fine if a user doesn't mind being confined to a pre-stored vocabulary. A pre-stored vocabulary is, however, under-utilizing the synthesizer's capability to produce *any* spoken work *on demand* as long as it has the appropriate input data. This is where TI's recently unveiled *allophone stringing technique* comes into play. TI linguists have chosen 128 separate sounds called "allophones" that can be linked together to sound out any word in the English language. Allophones are variations of a particular "phoneme" (the smallest unit of speech that can distinguish one utterance from another) that are modified by the environment in which they occur. For example, the aspirated (followed by a puff of air) "p" in "pin" and the non-aspirated "p" in "spin" are allophones of the phoneme "p." These allophones represent the sound more accurately than the phoneme.

A total of 128 allophones are grouped in a library occupying only 3 kilobytes of memory storage. Each allophone is identified with a numerical code (indicating the parameters for setting the filter characteristics in the LPC synthesizer). When a word is entered into the computer, its ASCII representation is identified; the computer then searches through a set of rules (contained in 7 kilobytes of memory storage) to pick out the appropriate allophones and string them together in the proper sequence (*concatenation*) to represent the keyed-in words.

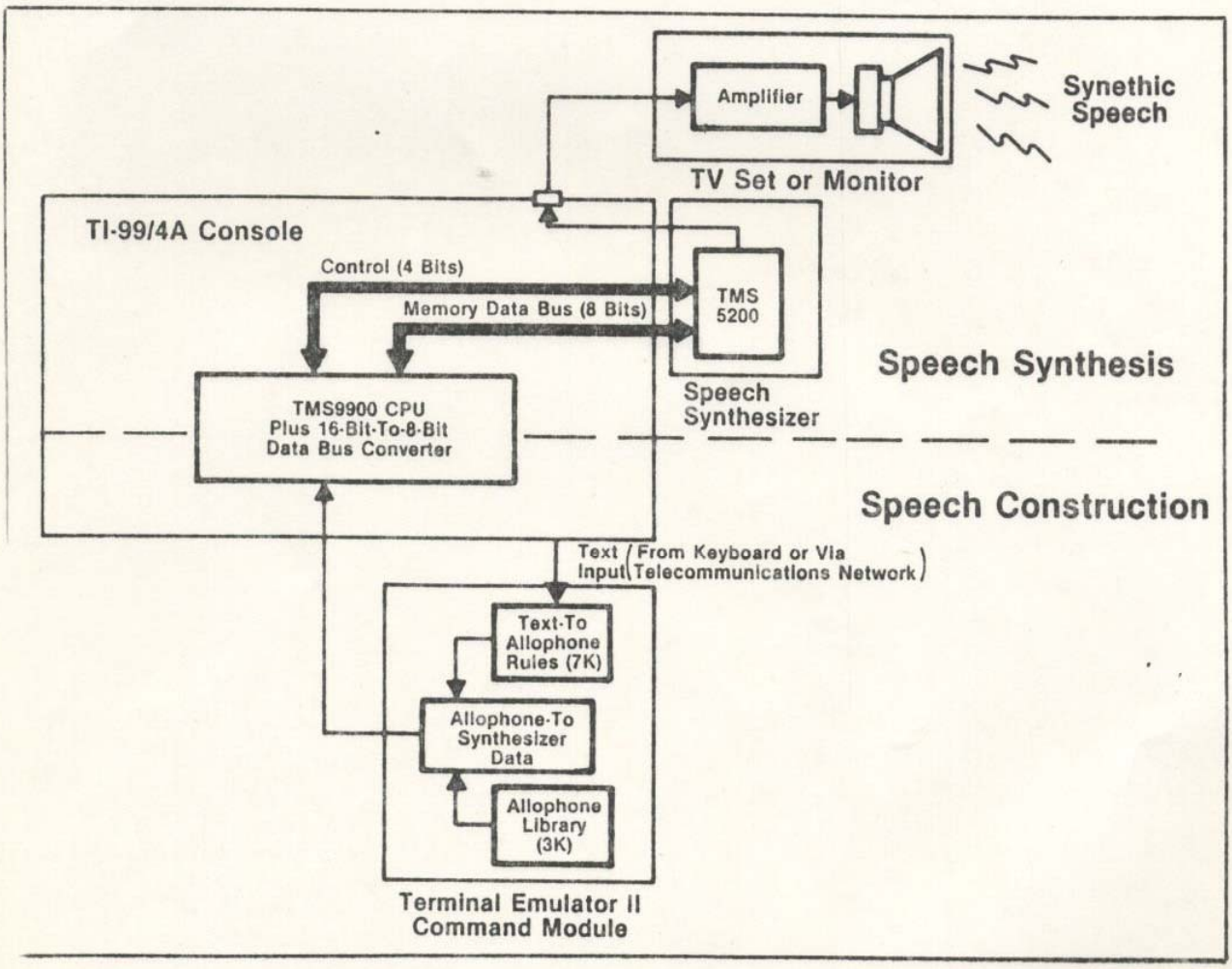
The rules (about 650 presently) overcome most of the many pronunciation exceptions and irregularities in the English language; they're able to select both phonemes and allophones correctly over 90% of the time. However, speech scientists have found it impossible to achieve 100% accuracy in a text-to-speech system of this type since there are too many silent letters and incongruities that humans perceive, but that the computer cannot discern. To get around this problem, some words must by typed into the computer phonetically or entered allophone-by-allophone.

If this were all the text-to-speech software did, the quality of the speech would sound monotonous and unnatural. It therefore provides its software with the ability to produce more lifelike inflection: Users can add stresses to certain syllables, and required pitch patterns to particular points in a sentence. Questions then sound like questions, and commands like commands!

Text-to-Speech with the TI-99/4A

The chip used in the speech synthesizer peripheral that attaches to the TI-99/4A is a TMS5200—a second generation of the TMS5100 (used in the Speak & Spell). It has the following added features : (1) "Speak External" input which allows the chip to accept speech data from a source other than a Speech ROM (read-only memory), (2) an internal buffer to store chunks of data (freeing the computer for other tasks), and (3) a memory data bus allowing it to work with any standard 8-bit microprocessor. (The 16-bit data bus of the TMS9900 micropressor is converted to 8 bits for use with all TI-99/4A peripherals.)

The text-to-speech production by this configuration is a two part process : (1) the *speech construction phase* in which letters are translated into a digital representation of component sounds and are concatenated (strung together), and (2) the *speech synthesis* phase in which the LPC circuitry "voices" the spoken words through a simulation of the mouth and nasal cavities. As seen in the accompanying diagram, speech construction is handled by the software resident in the *Terminal Emulator II* Command Cartridge, and speech synthesis by the TMS5200 chip within the separate speech peripheral.



Just the sound of the name Walt Disney conjures up images of all those fantastic animated movie classics spanning over a quarter century of entertainment for young and old alike. But recently, the celluloid magic of Disney Studios has taken on a new dimension with the release of their eagerly awaited science-fantasy, *TRON*—an incredible computer graphics extravaganza in which fantastic vistas of texture and light are generated artificially by computer. As movie-goers worldwide continue to be awed by *TRON's* video warriors and computer programs fighting for survival in an electric universe, a new awareness of computers—and in particular, the mind-boggling possibilities of computer-generated imagery—permeates the consumer cosmos. With one wave of Disney's digital wand, the glass of Cinderella's slipper has been magically transformed into the cathode ray tube of a video monitor.

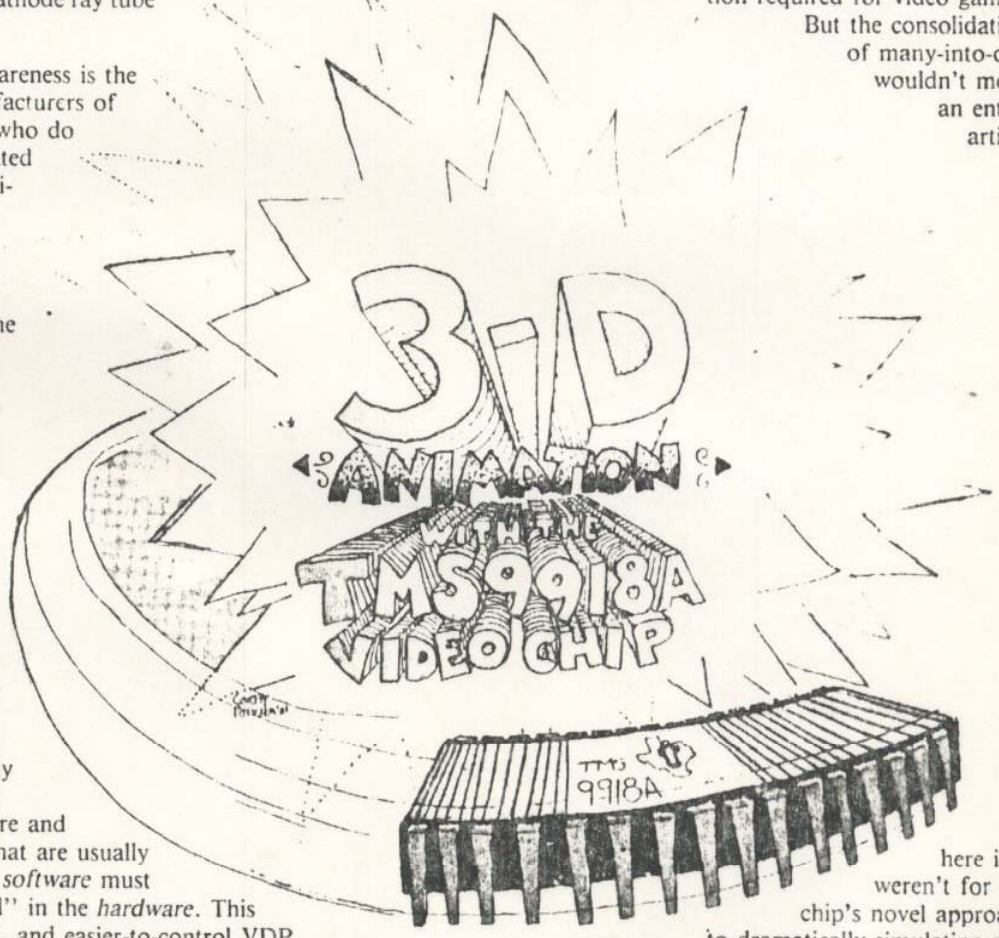
This heightened awareness is the death knell for manufacturers of consumer computers who do not provide sophisticated color graphics and animation capabilities. Fueled by *TRON* (and the horde of video clones that are destined to follow), the public's demand for, and expectation of, more visually spectacular video games and educational displays will surely take quantum leaps. How can computer manufacturers and software houses ever hope to satisfy this demand? That's one tough technical question that some of the finest design teams in the world are currently tackling. One thing is obvious, though—more and more special effects that are usually implemented through *software* must instead be "integrated" in the *hardware*. This means more powerful, and easier-to-control VDP (video display processor) chips—the silicon workhorses responsible for the displays.

The easier-to-control requirement doesn't necessarily mean easier for highly-trained, professional *programmers* to control. There will have to be a way for people such as as artists and "graphic gurus" with fantastic imaginations to interact directly with the display system—a way that requires only a bare minimum of "programming" experience to implement sophisticated visual effects.

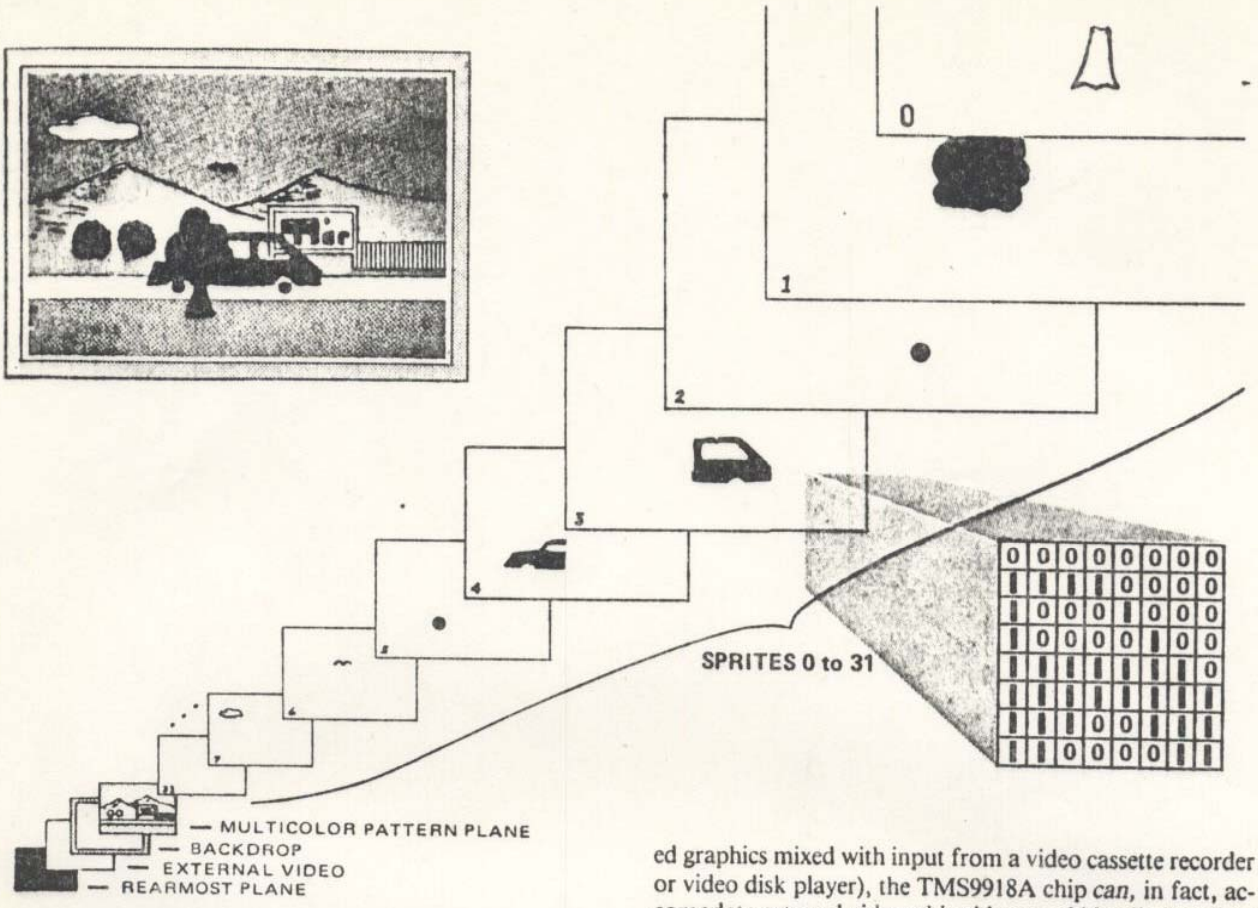
To anyone familiar with the interactive graphics capability of the Texas Instruments 99/4A Home Computer, it is obvious that TI has already made great strides toward this design goal—great enough, in fact, to cause at least two other well-known computer manufacturers to attempt to emulate TI with their "newly-discovered," smoothly moving graphic patterns now known universally as *sprites*. Color sprites as implemented on the TI-99/4A, however, have yet to be equaled in their versatility and ease of use in a multi-language environment. (Extended BASIC, TI LOGO, UCSD Pascal, 9900 Assembly Language and TI PILOT).

A Flat, Yet 3-D Sandwich

The wonder VDP chip behind sprites and other video affects that the 99/4A is capable of producing is called the TMS9918A. This complex LSI (large-scale integrated) chip represents the next generation beyond the many small- and medium-scale integrated circuits that formerly had to be assembled to achieve a display with a minimal level resolution required for video games. But the consolidation of many-into-one wouldn't merit an entire article here if it weren't for the chip's novel approach to dramatically simulating a 3-dimensional animated graphic display: It does this by creating nearly three-dozen flat, "stacked" geometric planes that are sandwiched one on top of the other onto the picture tube of your TV or color monitor.



On *each* of the first 32 planes (numbered 0 to 31), we can define the image of *one* sprite, give it one of the 15 standard colors (the 16th is transparent), and then set it in motion quickly and smoothly. We do *not* have to redefine the imagery over the screen to simulate motion, because once set in motion, a sprite can continue to move without further program control. When a sprite on a *lower* numbered plane (closer in the foreground) comes into contact with another sprite on a *higher* numbered plane, it progressively



blots the second one out and creates the illusion of passing in front of it.

For example, in the figure shown here, the moving car that is composed of four sprites set in motion together on plane numbers 2-5 will pass *behind* the stationary tree (composed of 2 sprites on plane numbers 0 and 1) and *in front of* the billboard which is drawn on the plane immediately behind the rear-most (number 31) sprite plane. By the same design rules, the cloud (plane 7) will mask the color of the sky behind it, and a bird (plane 6) both mask the sky behind, and appear to fly *in front of* the cloud. And since sprites move in a transparent surrounding, the scenery in the background behind the car may be seen *through* the "windows" of the moving vehicle! The entire scene has the appearance of depth and simulates a 3-D animated color movie.

The Multicolor or Pattern Plane is used for textual and fixed-graphics images. It is this plane (containing the sky, mountains, bushes, billboard, fence, roadway and grass) that the sprites on the remaining 32 planes appear to pass directly in front of.

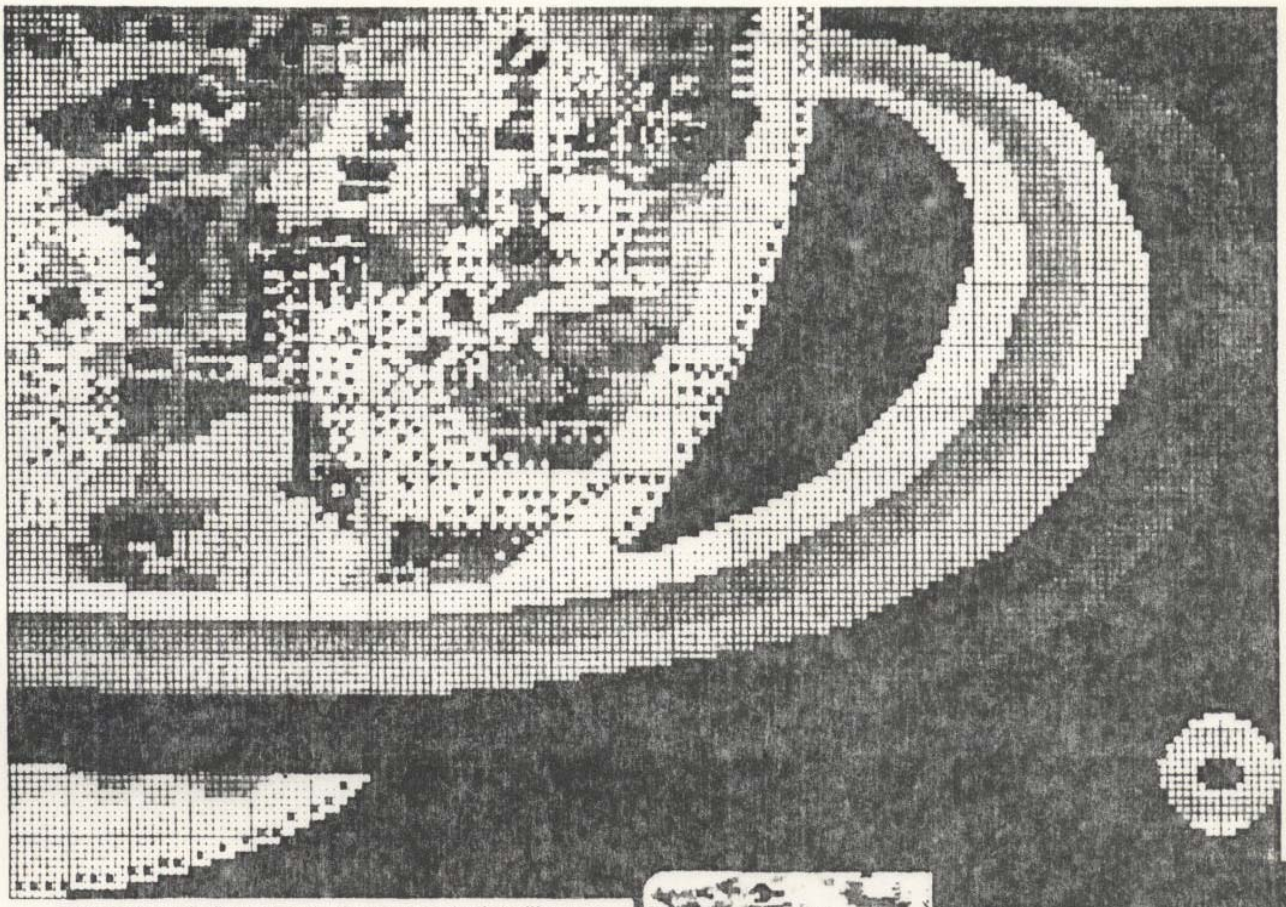
Immediately behind the Multicolor Plane is the Backdrop Plane—solid-colored and slightly larger than the other 33 planes in front of it, so that it forms a rectangular rim around the other elements on the display.

The rearmost plane is pure black, so that when the other planes are set to transparent, the screen appears to be black. Although there is no provision in the current version of the TI-99/4A for *simultaneous* on-screen mixing of external video with computer-generated graphics (e.g., sprites or fixed graphics mixed with input from a video cassette recorder or video disk player), the TMS9918A chip *can*, in fact, accomodate external video; this video would be displayed on the rearmost plane with part or all of it masked by computer-generated graphics until needed (e.g., as subtitles for the deaf or foreign language translation or perhaps a "real-life" video-taped space movie scene viewed through a scanner screen of a computer-generated starship command center). Add to this the capability of chaining together *multiple* 9918A chips, and you have the potential for a visual gaming or educational environment (in future versions of the TI Home Computer) that is simply mind-boggling!

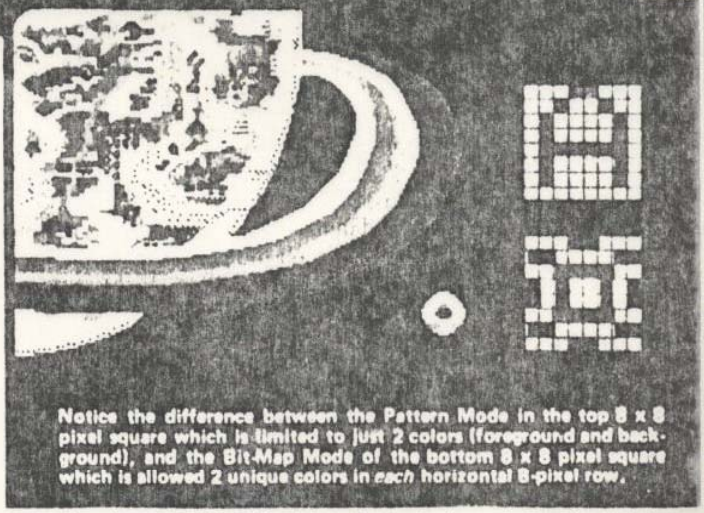
Those Magical Sprites

When sprites are on the screen, the 9918A chip organizes the display into a high resolution pattern of 256 by 192 little boxes or picture elements called "pixels"—the smallest controllable elements on the display. Each one of these 49,152 pixels represents a possible address for a sprite to reside at, or pass through when moving across the screen.

The shape of a regular or standard sprite is defined by an 8 × 8 bit pattern stored in memory. Each of these 64 bits correspond to one of the 49,152 screen pixels mentioned previously—with each being a single color whenever the bit pattern contains a 1 (is thereby "turned on"); a zero designates transparency ("turned off"). We can specify a larger sprite by either (a) using a 16 × 16 bit pattern ("a double-sized unmagnified sprite"), (b) magnifying the existing sprite by a factor of four, ("single-sized magnified"), or (c) using both techniques together to create a sprite sixteen times normal size ("double-sized and magnified"). This size feature allows screen objects to grow and shrink at will—with virtually *none* of the programming effort that would be required in more conventional VDP systems.

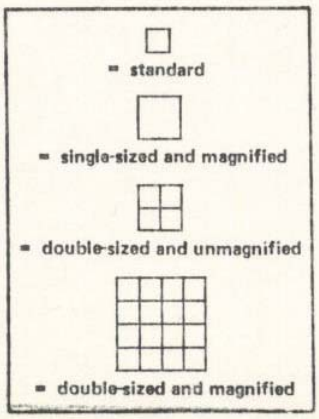


This high-resolution scene is shown over-sized to illustrate the color combination possibilities in Bit-Map Mode. Notice the color blending and shadings that the eye perceives when viewing the *same* scene on the screen-size display of the lower figure.

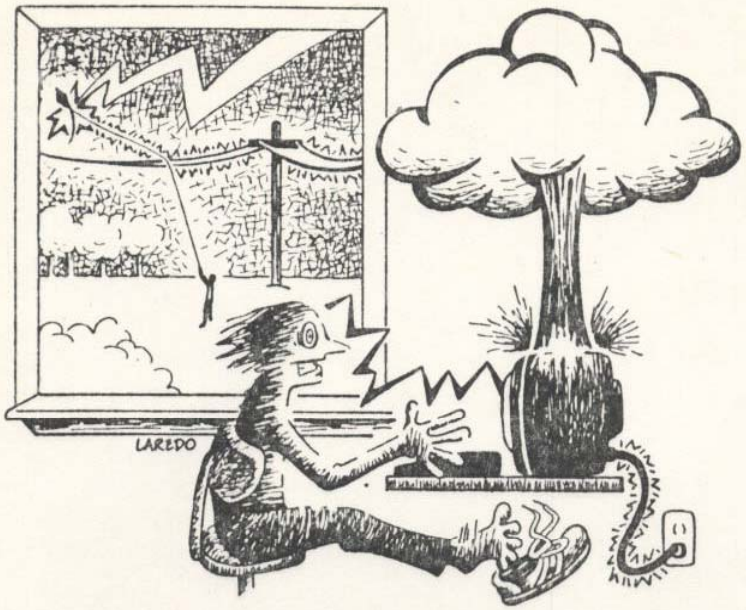


Notice the difference between the Pattern Mode in the top 8 x 8 pixel square which is limited to just 2 colors (foreground and background), and the Bit-Map Mode of the bottom 8 x 8 pixel square which is allowed 2 unique colors in *each* horizontal 8-pixel row.

Each sprite carries four attributes: The first two specify its horizontal and vertical position; the third defines its shape "name" (according to the bit-pattern concept described above); and the fourth specifies its color. Moving a sprite is simply a matter of changing its position indicators; it will continue moving smoothly on its own. The high-speed, smooth motion of a sprite compared with a conventional moving-graphic element is due to the smaller, more precise "steps" (higher resolution) that the sprite can take while moving. Animated secondary motion—for example, rotating wheels or an asteroid tumbling through space—is achieved by defining ("naming") several similar looking sprites in different secondary positions (e.g., states of rotation). Then swapping the sprite names causes what appears to be a *single* sprite to move smoothly across the screen.



The TMS9918A VDP chip has four modes of operation: (1) Graphics 1 or Pattern Mode, (2) Graphics 2 or Bit-Map Mode, (3) Text Mode, and (4) Multicolor Mode. The Pattern Mode consists of a 32-column by 24-row grid of 8 × 8 pixels in each 2-color grid square. Bit-Map Mode (shown in the above figures) allow *each* of the 8 horizontal rows within an 8 × 8 grid square to have 2 *unique colors*. In Text Mode the screen is a 2-color *single* plane (so sprites aren't available) of 40 columns by 24 rows composed of 6 × 8 grid squares. This allows an ASCII character set with each character formed from a 5 × 7 pixel grid, with 2 pixels between characters and rows. Multicolor Mode [see the "Super Crayon" article in this book for more information] divides the pattern plane into an unrestricted 64-column by 48-row color-square display, with *each* 4 × 4 pixel square allowed to take on any of the 15 colors or be made transparent.



POWER LINE PROBLEMS IN PERSONAL COMPUTERS

Although glitches, crashes, errors, false printouts, memory loss, and other forms of erratic microcomputer operation are usually blamed on software and hardware, most of these annoying problems actually come to you courtesy of your ordinary 120-volt powerline! These problems are directly traceable to three general causes: (1) processor-memory-peripheral interaction, (2) power line noise/hash, and (3) transient voltage surges. Fortunately, serious computer users don't have to live with these problems, because many types of corrective devices are available.

Powerline Coupling

The fact that microcomputer systems are so easy to hook up—just plug the computer and peripherals into the wall socket, and connnect the components with a few convenient male/female preassembled cables—makes them susceptible to power line noise. Connecting them to powerline strips that are integrated with RFI (radio-frequency interference) filters will effectively isolate the computer and peripherals from each other and from the power line—thus providing a convenient solution to the problem.

Hash

Hash is another problem altogether. When your favorite space-war game gets fouled up by "glitches," or your previously-proven-to-be-faultless program "blows up" or creates erroneous printout, externally created hash is the probable cause. Elimination of hash at the source is the most desirable solution. But with hundreds of potential sources (arcing in tools, motors, appliances, and other small electrical devices, plus loose, defective, or corroded light sockets, wall sockets, line-cord plugs, or wire connections), pinpointing the offender is often most difficult. That's where hash filters are most effective. They often can completely eliminate the interference.

An alternate approach to the hash problem is first to make certain that all equipment covers and shields supplied by the manufacturer are securely fastened in place. If that doesn't work, you might try building and installing your own shield. Also don't forget to make sure that you have an adequate grounding system with *direct ties* to a good ground rather than *ground loops* (which often provide a home for system hum that can induce glitches).

Transient Voltage Surges.

Transient voltage surges (transients) are certainly not friends of microcomputer circuitry. Semiconductor components are easily damaged by these momentary spikes—often 5 or 10 times the normal AC line voltage. And industry studies indicate that some transients have pulses up to 5,600 volts!

Common causes of destructive powerline transients include (1) demand power load switching by utility companies, (2) nearby lightning strikes, (3) static discharge, and (4) on/off switching of inductive motors, power supplies, air conditioning and refrigeration units. Any of these can cause a Differential Mode powerline surge—one in which short surges of extremely high voltage are developed *between* the AC lines. Anything connected to the AC lines will get a dose of this damaging voltage. The resulting "domino effect" could wipe out large sections of microcomputer memory.

A Common Mode surge occurs when *both* AC lines are brought to a very high voltage—a situation usually caused only by lightning. This high voltage may cause arcing between conductors and ground, destroying the insulation of power transformers (rendering the units worthless) and cables. Damage to switches and controls is also a frequent occurrence in this situation.

Besides the surge damages that are immediate and permanent, there are some harder-to-detect damages as well: deteriorated performance and shortened life-spans. These damages can be the most irritating since equipment will require repeated servicing and will often seem to be falling apart.

Fortunately, a large measure of surge protection is possible with clamping devices that can be placed across the AC line and between each AC line and ground. These devices are frequently built into special AC line cords, and thus, like the other protective devices mentioned, can be attached without altering any equipment.

Murphy's Law and the Home Computer

> "ANYTHING THAT CAN GO WRONG WILL GO WRONG"
> — Murphy's Law
>
> "And at the worst possible time."
> —Pincus's Corollary to Murphy's Law

The significance of Murphy's Law is etched into the hearts and minds of all professional programmers. As an outgrowth of their crucial need to prevent costly errors and protect their sanity, these professionals have developed a set of rules for minimizing the expected problems inherent in any programming project. In this article, I'll acquaint you with a few of these "tricks of the trade" that *you* can adopt when working with your own home computer.

Rule #1—Name that Variable

Most beginning programmers select simple two-letter variable names like X1, X2, X3, etc. The major problem with this kind of naming scheme is that you tend to forget what each one represents. Fortunately, both of TI's BASIC languages allow you to use long variable names. Unfortunately, using long names tends to slow down a BASIC language program. The solution is to pick a 4- or 5-letter name that adequately identifies the variable it represents. For example a program that needs a loop counter could have the name "LOOP" for the counter. [When picking names for variables, you should also be careful to avoid reserved words used in BASIC. If in doubt, check your *User's Reference Guide*.—Ed.] Keep a running list or chart of variable names and their meanings in front of you while you program. Avoid the temptation to add a variable to your program without updating the chart.

Rule #2—Save that Program

Get used to SAVEing your program after typing in about 50 program lines if your storage medium is a floppy disk, or 75 lines if cassette tape. Keep your labels up-to-date or else you may get confused in case of trouble. I always place the version number "V.XX" on the cassette before recording on it. Flipping the tape over and scratching out the old version number as soon as the program has been SAVEd makes sure that I don't accidently record over my last SAVE. For disk users, I suggest that they SAVE their programs under the names V01 and V02 *alternately* so that they don't fill up a disk.

Rule #3—Walk, Don't RUN

Computer-induced heartburn is one experience you'll definitely want to avoid. Yet how many of you court this malady by meticulously entering the last 50-75 lines of your program, then typing in RUN? If you're guilty of this practice, don't be too surprised if Murphy pays an unexpected visit and causes your computer to "freeze" on you—effectively wiping out everything in memory. The best way to avoid this is to always "back up" your *entire* program (on tape or disk) *before* you RUN it the first time. Then verify your recording. The next step is to *plan* what you want to test: Don't expect to have the program execute successfully the first, second, or even third time that you try it out. Instead, take a blank sheet of paper and write down what you want to test and how you will do it. Then RUN your program following your plan. If you notice a problem that does not halt your program, write it down on your test sheet and keep going. Don't stop to figure out what program line caused the problem; there will be time for that later. And above all, avoid the temptation to correct the error right away. Doing this will cause the following:

- If you change a program line, TI BASIC resets all the variable data (that you entered during your test) to zero or spaces. Subsequently, you will have to retype all that data back into your program.

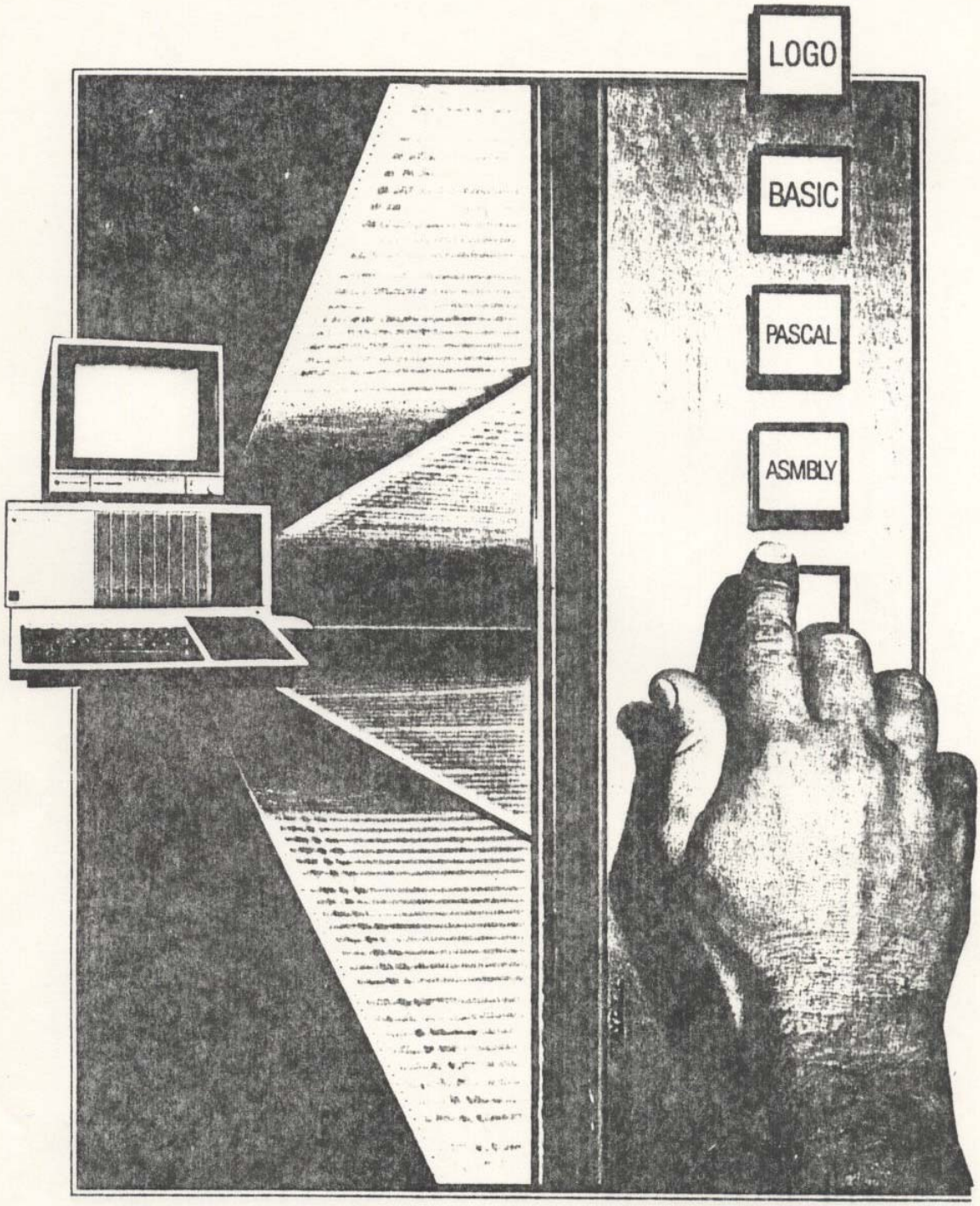
- You may lose your train of thought when you stop your test in the middle. It's possible to miss testing a *major* item because you spent your time fixing a *minor* problem.

After you have done as much of your test as possible, stop the program and begin fixing all the "bugs." Don't forget to mark down the statements that you change. This will come in handy if you must restore a previous version because of a problem in making corrections. There are various way to correct (debug) a program, some of which you'll encounter as you read through this book. One of the best, however, is Rule #4.

Rule #4—Test, Test, TEST !

Any professional programmer will tell you that it's extremely difficult to produce a bug-free program—maybe even virtually *impossible* to do with very complex programs. So even after extensive testing, *your* program may have minor flaws in it. Therefore, the only way to produce a relatively clean program is to test it as much as possible. This brings up another corollary to Murphy's law: "The only programs without bugs are the ones not yet written." This says it all, and should be reason enough to TEST, TEST, TEST !

2
Programming Techniques and Languages



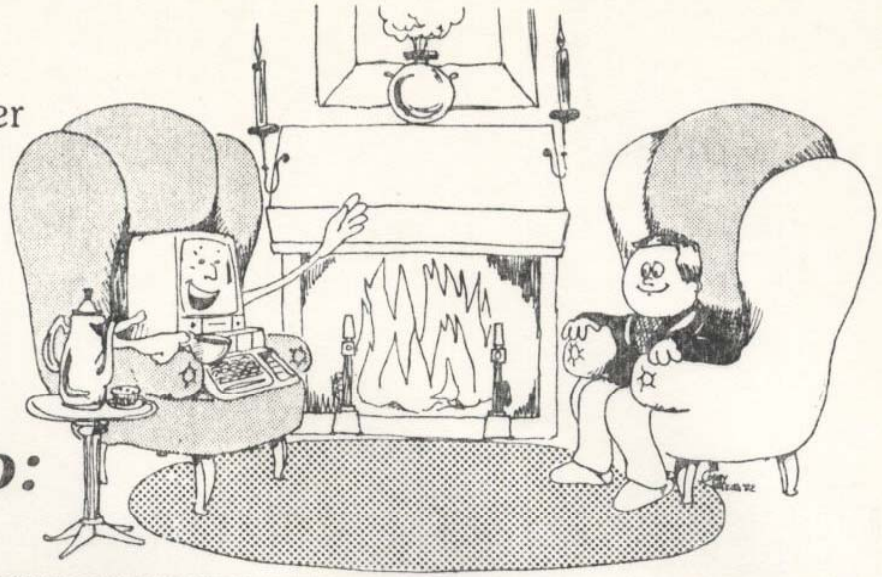
2

Programming Techniques and Languages

How to talk to a computer.

Languages for
the Home Computer

Chatting with Your Micro:



Home Computers are indeed wonderful machines. They have been carefully designed to allow beginners to do meaningful tasks, act as educational tools, and provide hours of inexpensive family entertainment.

All of this is made possible by the availability of "user-friendly" software—Command Cartridges, cassette tapes, and floppy disks that have been pre-recorded with programming instructions the computer can understand and carry out.

Users of this software need not concern themselves with how this programming was actually produced—unless, of course, they get smitten with that highly contagious human germ known as "curiosity," and want to understand something about the process.

"Programming" the Home Computer is *not* some mysterious rite that is meant to be practiced by a select few in secrecy. Rather, it is simply a means of communicating with a machine in a language that *both* humans and human-designed electronic circuits can understand—nothing more elaborate than basic, down-to-earth communication.

Languages, whether human-to-human or human-to-machine, differ widely in their complexity. Depending on the language, varying amounts of memorization and practice are required before a "speaker" can communicate effectively. The levels of computer language complexity run the gamut from conversational English phrases, to the switching on and off of electric current that the machine "understands" and transforms into various actions.

Before a user can begin communicating with a computer, however, one of three conditions must be met: (1) The user must be able to communicate in the computer's language; (2) the computer must be able to communicate in the user's language (i.e., English, German, Spanish, etc.); or (3) some common intermediate language must be established, understood, and used by *both* parties. By definition, the closer this intermediate language is to the machine's natural "electrical" language, the *lower* its level. And conversely, the closer to the human's language, the *higher* the level.

Machine Language

First, let's take a look at the lowest level of common intermediate language—referred to as "machine language." Since electricity can either be on or off—one of *two* possible conditions—machine language can only be constructed from *two* "words." This binary language is often expressed by humans with the two digits 1 and 0, with 1 representing the "on" state (presence of electricity), and 0 representing the "off" state (absence of electricity). Absolutely *shocking* in its simplicity, isn't it?

Figure 1.

"MACHINE LANGUAGE SAMPLE"

0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0
0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	1	1	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0
0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0

Figure 1 represents six machine language "sentences." It's not easy for a human to understand, is it? Yet when communicating this way, more explicit control of the machine is possible, because there can be nothing "lost in the translation."

TMS9900 Assembly Language

Human difficulty in communicating in a binary language led to the next step in the evolution of higher-level languages—an easier-to-remember ("mnemonic") way of expressing these binary "sentences." This was done by assigning combinations of alphabetic letters to represent operations formerly only expressable by binary sequences, and assigning a *full range* of characters (including numbers) to represent the things actually "operated" on.

This easier, alphanumeric way of communicating is called *Assembly Language* because these newly created scores of symbols must eventually be translated back (*assembled*) to their binary equivalents for the machine to carry them out.

Figure 2

```
            REF     VMBW,INPUT
LINE 1      TEXT    'HI, I AM THE TI-99/4A'
LINE 2      TEXT    'HOME COMPUTER '
LINE 3      TEXT    'WHAT'S YOUR NAME?'
BUFFER      BSS     32
LINE 4      TEXT    'NICE TO MEET YOU, '
GREET       LI      R0,0
            LI      R1,LINE1
            LI      R2,32
            BLWP    @VMBW
            LI      R0,64
            LI      R1,LINE2
            BLWP    @VMBW
            LI      R0,128
            LI      R1,LINE3
            BLWP    @VMBW
            LI      R0,BUFFER
            BLWP    @INPUT
            LI      R0,256
            LI      R1,LINE4
            BLWP    @VMBW
            LI      R0,288
            LI      R1,BUFFER
            BLWP    @VMBW
            END     GREET
```

Note: Keep in mind that the use of this and other sample program segments that follow are for comparison purposes *only*, and do *not* indicate the true power of any of the languages. Note also that the reference to a routine called INPUT doesn't imply the existence of that routine (as this is only an example).

In Figure 2 we are showing you part of an Assembly Language program that causes the computer to print several English language messages on the screen, and allows it to accept and acknowledge human response via the keyboard. The screen dialog goes like this:

HI, I AM THE TI-99/4A
HOME COMPUTER
WHAT'S YOUR NAME?

You type in your name

NICE TO MEET YOU,

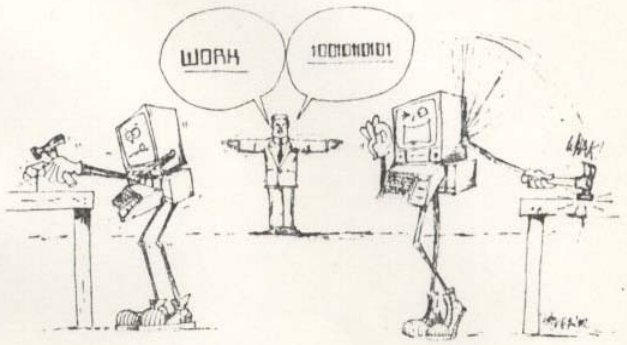
Your name appears here

Observe in Figure 2, the left to right sequence of symbols that must be followed if the program is to be assembled correctly. As an example of the proverbial "before and after"—Assembly Language lines that have been assembled back to binary—take a look at the last seven lines of symbols in Figure 2. The machine language that results from the assembling of these symbols appears as the entire sequence of left-to-right binary sentences shown in Figure 1.

Higher-Level Building Blocks

Although some very important programming is still done at the Assembly Language level, the majority of programs written are in higher-level languages. These languages are closer to human languages such as English than to machine language. To generate these higher-level languages, we must take ordered groups of Assembly Language statements and equate *each group* with a *single word* of the new, higher-level language we are generating. Each word of this new language is much more powerful than any single Assembly Language symbol: With *one* new higher-level word, we can make the computer do *several* things. This is a powerful technique, indeed, and has been the basis for all computer languages that have evolved.

For the computer to understand one of these new English-like languages, the language must first be translated into machine language.



Compiling & Interpreting Down

When translation of *all* the high-level language statements in a program takes place *before* the computer acts on the statements, the language is said to be *compiled*. The binary sequences that result from this compilation are then saved and later used directly any number of times.

On the other hand, the language is said to be *interpreted* when the computer acts on *each* statement immediately after that statement's translation. Therefore, *every* time an interpreted language program is "run" (all statements followed step-by-step to completion), the program *must* be re-translated. Because of this basic difference in translation technique responsible compiled language programs are faster than interpreted ones.

This is not to say that interpreted languages do not have compensating advantages. Ease of use is a case in point:

Additional Terms You'll Want to Know

Command Cartridge—A plug in plastic cartridge from Texas Instruments with integrated circuits that contain a computer program (software).

floppy disk—A mass storage device using a flexible mylar disk to record information. It is a more sophisticated alternative (quick random access) to cassette tape storage (sequential access).

Home Computer—The Texas Instruments TI-99/4A console with either home television or TI Color Monitor.

Integrated circuit(IC)—Integrated circuits have many individual components packed together or integrated in a small area. The circuits of the computer are fabricated on silicon chips. A chip is typically about 1/4 inch on a side. Today's chips are so sophisticated that the basic components of an entire computer can be fabricated on a single chip.

mnemonic—Assisting or intended to assist the memory.

screen—The home television or TI monitor to which the computer outputs information like numbers/letters/graphs, etc.

Speech Synthesizer—A peripheral device built by Texas Instruments for use with the Home Computer and used to reproduce the human voice electronically.

TMS9900—A very sophisticated integrated circuit (called a "microprocessor") containing all the most basic components of an entire computer. Designed and built by Texas Instruments, it is the heart of the Home Computer.

Just as soon as we finish writing the last program statement in an interpreted language, we can *immediately* run the program—without having to go through an additional intermediary step such as compilation. The translation in an interpreted language is therefore invisible or hidden from us.

Furthermore, in many interpreted languages such as the BASIC language that comes built into your Home Computer, statement misuse or errors of spelling in language vocabulary are checked for *right at the time the statements are typed in*. Appropriate error messages (if needed) will appear on the screen; the person doing the programming can then make immediate corrections.

TI BASIC

Because TI BASIC is a high-level and interpreted language, it is easy to learn and use. The sample program segment that follows (Figure 3) will cause the computer to carry on a dialog similar to the one previously shown in Figure 2. Notice how much easier the TI BASIC version is to understand.

Figure 3

```
100  PRINT "HI, I AM THE TI-99/4A"
110  PRINT "HOME COMPUTER"
120  PRINT "WHAT IS YOUR NAME?"
130  INPUT NAME$
140  CALL CLEAR
150  PRINT "NICE TO MEET YOU,"
160  PRINT NAME$
170  END
```

TI Extended BASIC

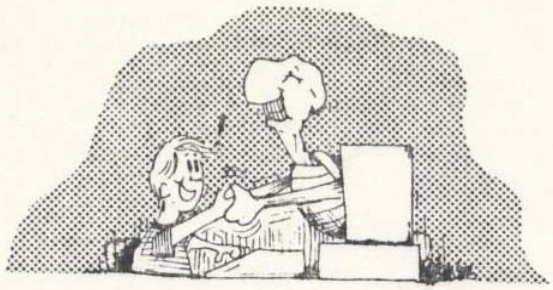
TI Extended BASIC, one of the higher-level languages that you can add to your Home Computer by plugging in the separate Command Cartridge for the language is similar to the regular built-in BASIC. It gives you everything that the regular BASIC does *plus* many special additional features such as arcade-style animated graphics (known as "sprites"), commands to control the Speech Synthesizer, as well as more precise control of on-screen text messages (demonstrated in Figure 4).

Figure 4

```
100  DISPLAY AT(2,1):"HI, I'M THE TI-99/4 HOME
     COMPUTER"
110  DISPLAY AT(6,1):"WHAT IS YOUR NAME?"
120  ACCEPT AT(8,5)VALIDATE
     (UALPHA)SIZE(15):NAME$
130  CALL CLEAR : : PRINT "NICE TO MEET YOU,
     ";NAME$
140  END
```

TI LOGO

Another interpreted language very popular with children and educators is TI's unique implementation of LOGO. It contains the previously mentioned sprites and features "Turtle Graphics." These line drawings generated by a "pen" attached to a simulated "turtle" object (that is moved about the screen with only simple heading and distance commands) are both enchanting and instructive—contributing to the wonder of discovery that children experience with the computer. See Figure 5 for a sample TI LOGO program (known as a *procedure*).



THE "USER-FRIENDLY" LOGO TURTLE

Figure 5

```
TO GREET
CLEARSCREEN
PRINT [HI, I AM THE TI-99/4A]
PRINT [HOME COMPUTER]
PRINT [WHAT IS YOUR NAME?]
CALL READLINE "N
PRINT "HELLO,
PRINT :N
END
```

TI PILOT

Whereas LOGO is a high-level language with great depth—i.e., the built-in vocabulary can be "customized" by the user—TI PILOT, another high-level language, is much more abbreviated. With a fixed vocabulary of only 15 major commands, the interpreted TI PILOT language still allows access to sprites, color graphics, and sound. Each command is represented by one or two letters followed by a colon. The program segment in Figure 6 illustrates a dialog in PILOT.

Figure 6

```
D:  R$(15)
T:  HI, I AM THE TI-99/4A
T:  HOME COMPUTER
T:  WHAT IS YOUR NAME?
A:  R$
T:  HI THERE, $R$
E:
```

With PILOT, you can develop effective educational programs even if you've had little or no programming experience. For this reason, PILOT is favored by educators as a language highly suitable for producing computer-assisted instruction (CAI) courseware.

UCSD Pascal

Currently, the only high-level *compiled* language available for the Home Computer is University of California at San Diego (UCSD) Pascal. This version of Pascal includes functions for accessing all the special Home Computer features. The language is more appropriate for professional programmers or users who wish to delve into more sophisticated programming. Although not as difficult as Assembly Language to master, UCSD Pascal is, nevertheless, much more difficult than other high-level languages on the Home Computer.

This compiled language also happens to be highly *structured*—i.e., it restricts programs to modular organi-

zation according to sets of specific construction rules known as *syntax*. Because it is both compiled and structured, programs written in UCSD Pascal are faster and easier to modify than most other high-level languages. This makes it a suitable language for large business and scientific programs. See Figure 7 for a very simple (almost trite!) example of our now-familiar man-machine dialog as written in Pascal.

Figure 7

```
PROGRAM GREET;
VAR     NAME: STRING;

BEGIN
  WRITE(OUTPUT,'HI,I AM THE TI-99/4A ');
  WRITELN(OUTPUT,'HOME COMPUTER.');
  WRITELN(OUTPUT,'WHAT IS YOUR NAME?');
  READLN(INPUT,NAME);
  WRITE(OUTPUT,'NICE TO MEET YOU, ');
  WRITELN(OUTPUT,NAME');
END.
```

ASPIC

Early in this article we implied that higher-level languages are constructed from other languages. This means that *you* have the opportunity to design your own personal languages for communicating with your Home Computer. You can do this by defining both the syntax and each word of your *new* language in terms of the statements and commands of an *existing* Home Computer language. The new ASPIC language is a case in point. Constructed from TI BASIC, ASPIC was created to simplify a child's manipulation of color graphics on the Home Computer. Figure 8 shows an example of a typical program segment. [See ASPIC article in this book for a complete discussion of this new language—Ed.]

Figure 8

```
10    CLEAR
20    MAKE +
30    MAKE X
40    COLOR SCREEN RED
50    COLOR + BLACK
60    COLOR X GRAY
70    LET R1 = 5
80    LET C1 = 5
90    LET R2 = 21
100   LET C2 = 13
110   REPEAT 9
120   DRAW + IN ROW#R1 COL#C1
130   DRAW X IN ROW#R2 COL#C2
140   LET C1 = C1 + 1
150   LET R2 = R2 - 2
160   END
```

TI FORTH

If you want to modify a language to fit your own particular needs, TI FORTH may be for you. FORTH is much like LOGO, in that the basic language implementation consists of a small number of built-in primitives (called *definitions*) from which you may construct new definitions. The primitive definitions and the new definitions which you create are entries in FORTH's *dictionary*. Once you've entered your new definitions in the dictionary, however, they're a permanent part of *your* FORTH implementation. Programmers who feel the need to simplify their work with custom modifications—software developers, for instance—will be most interested in FORTH.

Now it's up to *you* . . . Go ahead and strike up a conversation with your new-found electronic friend. Who knows? New respect for and long-lasting TIes with your Home Computer may be the result.

COMPARISON OF LANGUAGES FOR THE TI-99/4A HOME COMPUTER

	Minimum Relative Cost of System Components Needed[1]		Ease of Use (1-10)[4]	Execution Speed (1-10)[4]	Color Graphics Supported	Sprites Supported	High Speed Turtle Graphics Supported	Speech Supported (with additional Synthesizer peripherals)	Music Supported
	To Run Programs	To Write Programs							
ASPIC	1.0	1.0	9	1	yes	no	no	no	no
9900 Assembly Language	1.3 Note 2	4.7 Note 3	1	10	yes	yes	no[5]	yes	yes
		1.3 Note 3A							
TI BASIC	1.0	1.0	7	3	yes	no	no	no[6]	yes
TI Extended BASIC	1.3	1.3	6	4	yes	yes	no	yes	yes
TI LOGO	2.9 Note 8	2.9 Note 8	8	4	yes	yes	yes	yes	yes[7]
UCSD Pascal	3.3 Note 8	7.3	4	7	yes	yes	no[5]	yes	yes[7]
TI PILOT	5.4	7.5	10	5	yes	yes	no	yes	yes[7]
TI FORTH	1.3	4.7	4	7	yes	yes	no	yes	yes

1. Relative cost ratio with price of a TI-99/4A taken as unity.
2. Cost for running assembled program in Mini Memory cartridge.
3. Cost for writing Assembly Language programs using *Editor/Assembler*.
3A. Cost for writing *very* small Assembly Language sub-routines & programs using Mini Memory cartridge and separate *Editor/Asembler Manual*.
4. Number 1 represents worst case; 10 represents best case. Please note these values are subjective and are *not* based on any laboratory test data.
5. With the proper background and experience, the user can use this language to write his own turtle graphics support routines.
6. If the *Speech Editor* cartridge or *Terminal Emulator II* are added to the system, speech can be supported by the language.
7. The new TI LOGO II, TI PILOT, and USCD Pascal have enhanced music commands.
8. Minimum configuration based on cassette storage.

HOW TO WRITE YOUR OWN PROGRAMS

Using Flowcharts To Outline a Solution.

Sitting down in front of your TI-99/4A and running packaged software may stimulate your desire to try some programming of your own. If you have taken courses in computer programming or have had some practical on-the-job training, you can probably type some lines and have the computer do what you want. However, if you haven't had this experience, you may soon find the frustration of not knowing where to start too much to bear. Well, take heart! Here we present some basic information on how you can begin programming on your own.

A Framework for Writing Programs

Computer programming is an exercise in reasoning and logic. Before programmers develop software to do specific jobs, they plan their attack on the individual elements that are inherent to those jobs or problems. It is helpful to have in mind a general framework for solving each problem.

This general framework could take a number of different forms. Most will, however, contain similar steps. These steps can be described as follows.

1. Define the Problem

Initially, it is necessary to have a good understanding of exactly what you want the program to do. If it is possible, try to express the problem in a simple thought or sentence stating the intended outcome of the programming effort. Defining the problem in this manner may not only save you time, but may also help focus your efforts.

2. Outline the Solution

This step is the primary purpose of this article. We'll get back to examine this step in more detail later.

3. Select the Algorithm

Many problems requiring a computer for solution depend on certain mathematical algorithms that are required in the calculation of the desired solution. For those who are puzzled by the word "algorithm," mathematicians and math teachers use this word to refer to the specific method of solving a certain kind of mathematical problem. For example, you may have been taught to subtract whole numbers by placing the larger number on top, the smaller on the bottom, and to borrow when necessary. This is but one possible algorithm for subtraction. In general, you can either locate those algorithms that are necessary from published sources, or design your own. In either instance, the simpler the algorithm, the better.

4. Writing the Program

Many people believe the writing or "coding" of the program is what computer programming is all about. Actually, this is just *one* in a series of steps. Prior planning (as detailed in steps 1-3, above) is absolutely essential before the actual writing of the program can begin. And inherent in the writing of the program must be a reasonable understanding of the computer language you will be using.

5. Debugging

Once you have typed the program into the computer, it is necessary to run it to determine if and what difficulties exist. You will seldom write an error-free program on the first draft. Trying to locate and correct those "bugs" can be frustrating. This is where some of the 99/4A's built-in features help tremendously.

6. Validating the Program

In this step, you intentionally try to locate situations in which the program yields inaccurate or undefined solutions.

7. Documentation

It is a good idea to record the characteristics of the program such as its intent, algorithms, and specifications. Some day, in the future, when you decide to modify this program you will be very happy that this documentation exists. Incidentally, when buying a program, the author's documentation (or lack thereof) can often be a good indication of the quality of the program.

Outlining the Solution

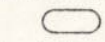
The development of an adequate outline (Step 1 above) is the most critical step in writing a program. Many of us dabblers in the art of programming seem to fail in developing an adequate outline. My intention here is to demonstrate to you some elementary outlining techniques—in the hopes that we, the dabblers, may be able to improve our lot in the somewhat puzzling world of bits, bytes, and bugs.

Flowcharting

There are a number of methods available for outlining a solution to a problem. Of those used in computer programming, flowcharting is one of the simplest and easiest.

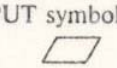
To introduce you to the flowcharting method, let's first look at some of the symbols used.

1. START and END symbol



This symbol is used to indicate both the beginning and the end of a program.

2. INPUT – OUTPUT symbol



The input-output symbol is used to indicate where the user of the program will need to supply a piece of data or where a calculation will be printed out for the user.

3. COMPUTATION or ASSIGNMENT symbol

This symbol is used to indicate where computations or assignments of values to variables will occur.

4. DECISION symbol

This decision symbol is used to indicate where a "yes" or "no" or "true" or "false" decision point is located.

5. STOP symbol

This symbol is used in some programs to indicate a termination point if this point is different from the end point of the program.

There are other symbols that can be used according to your needs. Also, remember that no rules exist to stop you from developing your own symbols.

Toward a Workable Technique

The outline strategy that works best for me is to start off simple and then increase the complexity of my outline until it does what I want it to do. My approach includes: (1) writing a sentence that defines the problem I want to solve, (2) preparing an informal outline, (3) developing a more complex flowchart, and then (4) writing the program. To demonstrate how this approach leads you to developing a better program, let's take a look at some examples.

EXAMPLE 1

For our first example, let's write a program that will add two numbers together and print their sum. We will design the program so that we may input two numbers from the console. This is called an *interactive* program, in that the user must input the values to be assigned to the variables. Following the approach presented, we first define the problem:

Step 1. Definition of the Problem:

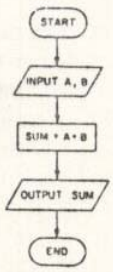
The program will take two numbers being input from the console, add them together and print the sum.

Step 2. Informal Outline:

1. Start
2. Input two numbers, A and B
3. Add A and B
4. Output the sum of the numbers

Step 3. Flowcharting:

Using the flowcharting symbols, the solution is further developed:



Explanation of the flowchart

The "flow" is evident in the continous line running from the initial start symbol to the final end symbol. The input symbol shows that the two values are requested, with the first input value being assigned to the variable A and the second to B. The addition of the two numbers and the assignment of their sum to a variable occurs inside the computation symbol. The value of the sum is then output, and the program ends. The algorithm necessary for the solution is shown.

Now that the problem has been outlined, we proceed to write or code the program.

Step 4. Coding:

```
100 REM **ADDITION PROGRAM**
110 INPUT A,B
120 LET S=A+B
130 PRINT S
140 END
```

Explanation of the program.

The program shows how the original intent is followed.

Line 100 contains a REM statement allowing us a means of identifying the program.

Line 110 allows the user to type in the two numbers to be added.

Line 120 assigns the value of A plus B to the variable S.

Line 130 prints the value of S.

Line 140 ends the program.

Since my primary intent here is to explain how an outline is developed and used, I will not explain the TI BASIC command statements but assume that readers of this article have already read most of the TI *Beginner's BASIC*, the book that came with their computer.

EXAMPLE 2

For a more complex example, let's develop a program that will select and print the larger of two input values.

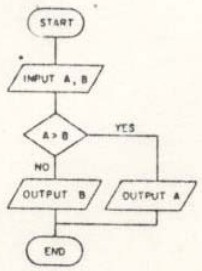
Step 1. Defining the Problem.

Given two numbers, the program will select the larger of the two and print it.

Step 2. Informal Outline:

1. Start.
2. Input two numbers, A and B from the console.
3. Compare number A with B. If A is larger than B, print A. If A is not larger than B, print B.
4. End.

Step 3. Flowcharting:



Explanation of the flowchart.

The flowchart begins with the start symbol. The two values are then input. In the decision box, a comparison of the value A with B takes place. If the statement A>B is true, the computer is instructed to bypass the output B box, and output the value of A. If the statement is false, the computer continues down the chart to output the value assigned to B. The program then ends.

Step 4. Coding:

```
100 REM **PRINTS LARGER OF TWO NUMBERS**
110 INPUT A,B
120 IF A>B THEN 150
130 PRINT B
140 GOTO 160
150 PRINT A
160 END
```

Explanation of the program.

Line 100 is a REM statement used to identify the program.

Line 110 allows the user to input the two numbers to be compared.

Line 120 is used to compare the two numbers. If the statement *A is greater (>) than B* is true, the computer is *then* instructed to go to line number 150 to print A. If the statement in line 120 is false the computer continues to the next line.

Line 130 prints the value of B, as it must be the larger.

Line 140 is used to direct the computer to go to line 160. Without this line, the computer would print the value of B, then the value of A. This is, of course, not what we wanted.

One difficulty exists with this program. If A and B are equal, the program will not be able to distinguish the two. (If this arises, B will be printed.) This difficulty could be corrected for this situation by allowing another step where value A and value B could be displayed.

EXAMPLE 3

Let's take a look at one more simple example. This time we'll try writing all the squares of the integers between 1 and 99, inclusive of the two boundaries.

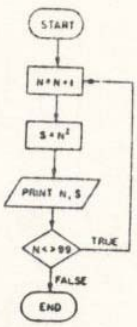
Step 1. Defining the Problem:

The program will make a list of all the squares of the integers between 1 and 99, inclusive.

Step 2. Informal Outline:

1. Start.
2. Let N be a variable whose initial value is 1.
3. Compute the value of N^2, and let the result be the value of S.
4. Print N and S on one line of the screen.
5. If the value of N is 99, then end the program. Otherwise go to step 6.
6. Add 1 to the value of N and then go back to step 3.

Step 3. Flowcharting:



Explanation of the flow chart.

After the program starts, the variable N is increased by 1. As the TI BASIC will automatically set the initial value of N to zero, using the statement N = N + 1 will set the first value of N to 1. Next, the square is calculated. Both the integer and its square are then printed. The next step checks to see if N is equal to the upper boundary of 99. If N is equal to 99, the computer is instructed to end the program. If N is not equal to 99, the program loops back to add 1 to the value of N and continues.

Step 4. Coding:

```
100 REM **SQUARES**
110 LET N=N+1
120 LET S=N^2
130 PRINT N,S
140 IF N<99 THEN 110
150 END
```

Explanation of the program.

Line 100 is the REM statement.
Line 110 adds 1 to the variable N.
Line 120 computes the square.
Line 130 prints the integer N and its square S.

Line 140 determines if the value of N is 99. If N is equal to 99, the computer goes to line 150 and ends the program. If N is not equal to 99, the computer returns to line 110.

Line 150 ends the program.

Now that we have seen the use of the outlining technique in some rather elementary program examples, let's get serious and try something more challenging.

EXAMPLE 4

Let's try writing a program to test our recall of a series of digits. With each correct matching of a digit, we'll instruct the computer to add another digit to the series.

Step 1. Defining the problem:

The program will display a series of digits of increasing length and ask the user to recall the correct order of the digits.

(It might be helpful to place some limits on the program to further qualify what we want it to do. This can be done in the informal outline.)

Step 2. Informal Outline:

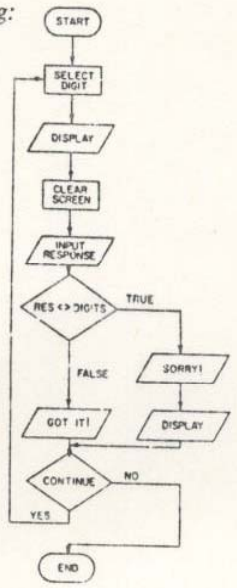
1. Start.
2. Have the computer select a random digit.
3. Display the series of digits for a short time.
4. Clear the screen.
5. Ask for a response from the user.
6. Compare the response to the series of digits.

7A. If the response is correct, congratulate and ask if the user wants to continue.

7B. If the response is incorrect, show the correct series of digits and ask if the user wants to continue.

8. If the user wants to continue, have the computer select another digit and add it to the end of the previous series.
9. If the user does not want to continue, end the program.

Step 3. Flowcharting:



Explanation of the flowchart.

After the program starts, a random digit is selected. The series of digits is displayed, and the screen is cleared. The user is then asked to respond. If the response is correct, the computer offers congratulations with a GOT IT! message, then asks if the user wants to continue. If the response is incorrect, the computer says SORRY!, then displays the correct response. The user is then asked if he wants to continue. If the answer is yes, the computer loops back to the random digit selection box, tacks on an extra digit to the string, and continues. If the answer is no, the program ends.

Step 4. Coding:

In coding the program, we'll use the RANDOMIZE and RND statments from TI BASIC to get a better selection of digits. Take the number returned by RND, multiply it by 10, then take the integer portion: The algorithm is INT(RND*10). In order to display, compare and add digits to the series, we'll translate them into a string using the STR$ function.

Explanation of the program.

Lines 100-170 are REM statements.

Line 180 is the RANDOMIZE statement.

Line 190 begins the selection of the random digit. With this statement, the computer will display a series of digits starting with a single digit and extending to an upper limit of 25 digits maximum.

Line 200 is the algorithm for selecting the random digit and assigning its value to the variable A.

Line 210 translates the digit selected to a numeric string. The line will also function in adding each digit selected to the end of the previous series of digits.

Line 220 clears the screen.

Lines 230-250 present the series of digits and tell you how much time you are allowed to study the series.

Lines 260-270 time the digits being displayed. Going through the FOR. . .NEXT loop takes about five seconds.

Line 280 clears the screen.

Line 290 directs the computer to jump to line 320. The line is intended to get us out of the FOR I. . .NEXT I loop without disrupting it.

Line 300 continues the FOR I. . .NEXT I loop.

Line 310 ends the program.

Lines 320-330 prompt the user to respond.

Line 340 compares the response to the series of digits. If the response is incorrect, the THEN condition directs the computer to line 380, which is the SORRY! comment. If the response is correct, the computer goes to the next line (line 350).

Line 350 clears the screen.

Line 360 congratulates the user.

Line 370 directs the computer to go to line 390, bypassing the SORRY! comment.

Line 390 asks if the user wants to continue.

Lines 400-410 check to see if the user is interested in continuing.

Line 420 returns the computer back into the FOR I. . .NEXT I loop.

Line 430 ends the program.

```
100 REM  *********************
110 REM  ***                 ***
120 REM  ***NUMBER MATCH    ***
130 REM  ***                 ***
140 REM  *********************
150 REM
160 REM
170 REM
180 RANDOMIZE
190 FOR I=1 TO 25
200 LET A=INT(RND*10)
210 LET MSG$=MSG$&STR$(A)
220 CALL CLEAR
230 PRINT "HERE IS THE NUMBER"::
240 PRINT MSG$::::
250 PRINT "YOU HAVE FIVE SECONDS":"TO
    STUDY IT"
260 FOR DELAY=1 TO 1500
270 NEXT DELAY
280 CALL CLEAR
290 GOSUB 320
300 NEXT I
310 GOTO 430
320 PRINT "TYPE THE NUMBER"
330 INPUT RES$
340 IF RES$<>MSG$ THEN 380
350 CALL CLEAR
360 PRINT "GOT IT!"
370 GOTO 390
380 PRINT "SORRY! THE NUMBER WAS:";MSG
    $
390 PRINT "DO YOU WANT TO CONTINUE.
    "Y" OR "N"."
400 INPUT ANS$
410 IF ANS$<>"Y" THEN 430
420 RETURN
430 END
```

FINAL COMMENTS

Once you've had a chance to use this approach—defining the problem, doing an informal outline, flowcharting, and then coding—in a project of your own, programming your computer will no longer be as forbidding and mysterious as you first thought.

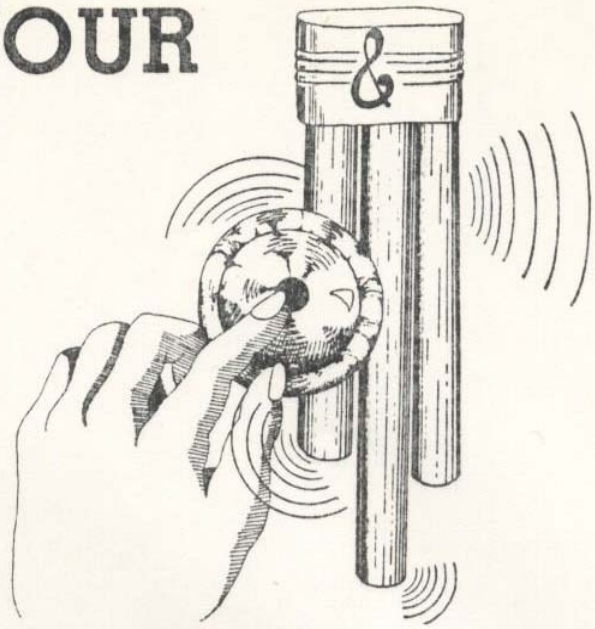
Before attempting programs of your own, you may want to try a little exercise. Add the following features to the previous program:

(1)Allow the user to choose how much time the digits are displayed on the screen.

(2)If the response is correct, play a 3-note chord.

(3)If the response is incorrect, play one note of noise and print a screen message that tells how many digits were contained in the largest number correctly guessed.

.IVENING UP YOUR CALL SOUNDs



he CALL SOUND subprogram in TI BASIC commands an amazing integrated circuit in your TI-99/4A called the SN76489 Sound Generation Controller. On ngle chip, TI has squeezed in three programmable fre- ncy dividers, a programmable noise generator, four pro- mmable attenuators (volume controls), and eight registers hold the data that control the tones, noise, and their ume levels. In effect, the tones and noise are synthesized our specifications from a frequency of 3.58 megahertz; is also the frequency that carries the color information m your computer to your color monitor or video dulator.

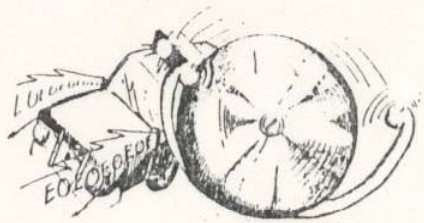
f you have used CALL SOUND only to produce scellaneous beeps, noise, and music, read on. I'm going give you some "mini programs" that demonstrate the iety of other sounds your Home Computer is capable producing.

or the first example, let us try to re-create the sound a door bell of the type associated with the once popular von Calling" commercial. This is an example of an ob- that is struck a sharp blow and allowed to vibrate at resonant frequencies. The following characteristics are ded to recreate this sound: 1) the fundamental frequen- of the two tones, 2) the overtone frequencies, and 3) radually decaying volume. Those of you with a sense of olute pitch would immediately recognize the two fun- nental frequencies, but in my case, I actually measured dimensions of the sounding bars and their points of sup- t and determined with a magnet that the bars were prob- y steel. From a textbook, *Acoustical Engineering* by rry F. Olson, I obtained the formula and values of the stants needed to calculate the resonant frequencies of bars. The calculated frequencies came out to be very se to 698 and 554 cycles per second (F and C# above high . The book also told me that the two closest overtones re 2.756 and 5.404 times the fundamental frequency. The rs were supported on rubber mounts close to the oretical nodes (points of minimum vibration) for the fun- mental and the first overtones, but were located near nts of maximum vibration for the second overtone. I refore assumed that the second overtone would be mpened out, so I omitted it from the CALL SOUND cification for each tone. The decaying volumes for the es were obtained by including each CALL SOUND in FOR – NEXT loop as follows:

```
100 REM DOOR CHIMES
110 FOR A=0 TO 30 STEP 5
120 CALL SOUND(-99,698,A,1924,A)
130 NEXT A
140 FOR A=0 TO 30 STEP 5
150 CALL SOUND(-99,554,A,1527,A)
160 NEXT A
```

If you are wondering about the significance of the 99 for the durations, it is simply an easily keyed number larger than the 50 milliseconds needed to make the steps sound continuous. The minus sign indicates that the sound generator will be updated as soon as the new value for A is determined; the duration specified need only be long enough to cover the time between updates.

Next, let us try a sound in which the frequency varies with time. A siren is an example which can be characterized by a slowly rising and falling frequency. Apparently, this is a sufficient clue to the brain for us to recognize it as a siren. Try varying the frequency range step in the following program to see how far it can be varied and still be recognizable as a siren.



```
170 REM SIREN
180 N=1
190 FOR F=700 TO 900 STEP 5
200 CALL SOUND(-99,F,0)
210 NEXT F
220 FOR F=900 TO 700 STEP -8
230 CALL SOUND(-99,F,0)
```

```
240 NEXT F
250 N=N+1
260 IF N=4 THEN 270 ELSE 190
270 END
```

N=4 on line 260 limits the siren to 3 up-down frequency sweeps.

In the next example, let us vary both the frequency and the volume as a function of time. Imagine a large "killer" bee buzzing around you, with the frequency of the buzz proportional to the rate of the beating wings, and the volume proportional to the closeness of the bee.

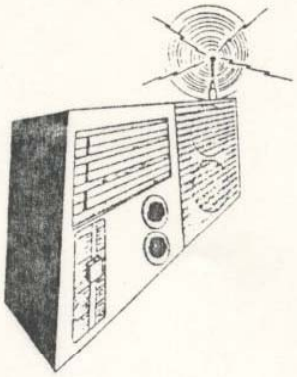
```
280 REM BEE
290 N=1
300 CALL SOUND(-99,RND*8+110,RND*10)
310 N=N+1
320 IF N=75 THEN 330 ELSE 300
330 END
```

Unlike the previous examples, where the variations in frequency and volume were obtained by using a FOR-NEXT loop, the variations in this case were obtained by using the RND statement. It is interesting to note that this routine will not sound the same in TI Extended BASIC—the bee sounds very sluggish. This is one case in which the TI BASIC runs faster than the Extended version.

For the next sound, imagine that you are tuning a shortwave radio receiver. The background static is simulated with the noise type (-8), and the random signal is simulated with frequency #3. The random volume on frequency #3 simulates varying signal levels with the noise volume formulated to be high when the signal level is low and vice versa.

```
340 REM SHORTWAVE RECEIVER
350 N=1
360 F=RND*15000+110
370 A=RND*30
380 CALL SOUND(-99,111,30,111,30,F,A,-8,30-A)
390 N=N+1
400 IF N=100 THEN 410 ELSE 360
410 END
```

Frequencies #1 and #2 are "do nothing frequencies" since their volumes are set to the minimum and are inserted so the program will recognize frequency #3, from which noise type -8 is derived. The 111's therefore were picked for the ease of inputting.



Next, imagine that the radio of the previous example is now tuned to a "pre-ASCII" teleprinter signal which uses an 850 cycle-per-second frequency shift to differentiate between a mark and a space.

```
420 REM RADIO TELEPRINTER
430 N=1
440 CALL SOUND(22,2975,0)
450 FOR D=1 TO 5
460 S=850*INT(RND*2)
470 CALL SOUND(22,2125+S,0)
480 NEXT D
490 CALL SOUND(31,2125,0)
500 N=N+1
510 IF N=30 THEN 520 ELSE 440
520 END
```

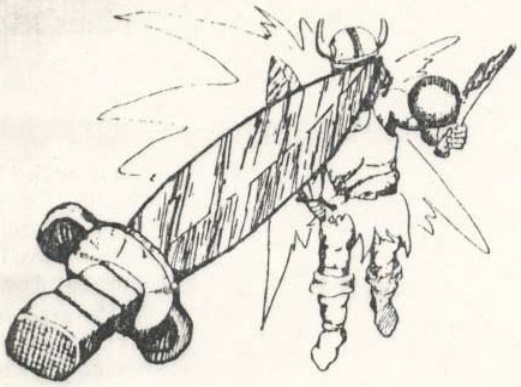
One character consists of a 22 millisecond (ms) start pulse, followed by a five-bit code for the character with each bit 22 ms long, and a 31 ms stop pulse. Line 440 generates the start pulse, which is always a space. The FOR – NEXT loop in lines 450-480 randomly generates a mark or space pulse for the five data bits, and line 490 generates the stop pulse, which is always a mark. Line 510 limits the number of characters generated to 29. Like the "bee" sound, this will not come out well in Extended BASIC. In general, data communications signals are easy to imitate because they are well defined by standards.

For a change of pace, try the following sound:

```
530 REM FOOTSTEPS
540 N=1
550 X=INT(RND*5)
560 IF X=2 THEN 620
570 CALL SOUND(5,-3,5)
580 CALL SOUND(30,-7,20)
590 CALL SOUND(500,-7,30)
600 N=N+1
610 IF N=30 THEN 640 ELSE 550
620 CALL SOUND(60,-7,20)
630 GOTO 590
640 END
```

The CALL SOUND on line 570 is the heel contacting the floor, followed by the sole contact on line 580. The CALL SOUND on line 590 is the delay between steps. Lines 550, 560, and 620 add a shuffle about once every 4 steps to make the footsteps sound a little more natural. Changing the noise type on line 580 from -7 to -5 will make the shoes squeak.

The sound of a sword fight can be re-created by recognizing that the sword blade is a resonator like the door chimes, except that instead of being essentially free, it is clamped at the handle—thus creating overtones at different ratios than the chime bars. Also, the amplitude decays faster, since the collision of the two blades would have a dampening effect.



```
650 REM SWORD FIGHT
660 N=1
670 FOR A=0 TO 30 STEP 15
680 CALL SOUND(-99,1000,A,3250,A,6750,A)
690 NEXT A
700 FOR D=1 TO RND*200
710 NEXT D
720 N=N+1
730 IF N=30 THEN 740 ELSE 670
740 END
```

Lines 700 and 710 add a random delay between sword clashes.

For the final example, let us try to simulate the sound of an internal combustion engine starting, accelerating, and then decelerating to a stop.

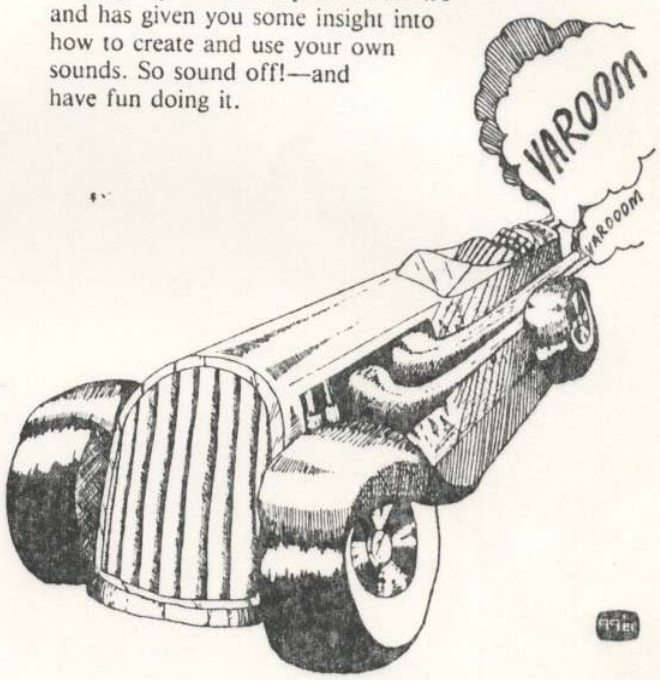
```
750 REM ENGINE
760 FOR N=1 TO 8
770 CALL SOUND(60,220,8,-5,0)
780 CALL SOUND(60,220,8,-5,5)
790 NEXT N
800 CALL SOUND(80,220,8,-5,0)
810 FOR F=1000 TO 5000 STEP 20
820 CALL SOUND(-99,111,30,111,30,F,30,-8,0)
830 NEXT F
840 FOR F=4000 TO 800 STEP -50
850 CALL SOUND(-99,111,30,111,30,F,30,-8,0)
860 NEXT F
870 END
```

Lines 760 through 800 simulate an electric starter motor. The accelerating and decelerating engine sound is made by sweeping noise -8 up and down in a FOR-NEXT loop.

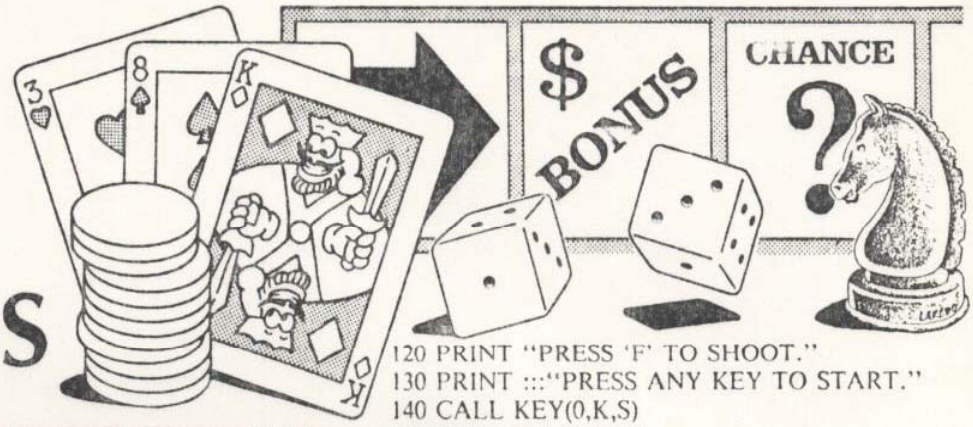
Now that you're convinced that your computer can produce a wide variety of sounds, you are probably wondering how one uses these sounds. If you are an adventure game programmer, suppose that the player is confronted with a door with a knocker and a bell button. Wouldn't it be more interesting if the player heard the bell upon pressing the bell button—before getting the usual textual message? Or if you are dynamically simulating a race car, you could use line 820 in the engine sound example in a CALL KEY loop where the F parameter would depend on the accelerator pedal setting. The duration in the CALL SOUND would have to be increased if you are updating other parameters in the loop for the sound to be continuous.

One nice thing about sounds is that the listener will make up the visual image that fits, which is why the radio programs of years past were so effective. The bee sound, for instance, immediately conveys the situation, whereas a screenful of color graphics would be hard-pressed to evoke the same feeling. Thus, for the programmer of interactive fiction, sound should be a very effective way to make a story come alive. If you could collect enough sounds, you could even write a sound effects program where a given sound could be accessed on cue for stage plays.

Hopefully, this article has opened your ears to the sound-making capabilities of your TI-99/4A and has given you some insight into how to create and use your own sounds. So sound off!—and have fun doing it.



FUN & GAMES



Psst! I've got a little secret for you, gang: *Designing* and *programming* your own game can be just as much fun as *playing* games produced by others. And best of all, it's really not as hard as you might think. . . .

Pick an Idea

You can have a maze, a game using dice, a card game, a memory-type game, a board game, a popular sport, a game involving logic, a game using skills or reaction time, some form of hide-and-seek, an adventure, or a myriad of space and shooting games. Still don't have a game plan? Walk through a video arcade to get some ideas.

Use the Computer

Now this sounds silly, doesn't it, since we're talking about writing computer games. Let me explain. If you write a game of tic-tac-toe or Othello for two players, you're really only utilizing graphics—the game could just as well be played on paper or on the board. But, if you write the game for one person against the computer, then you *are* using the computer to help go through a logic process. Another use of the computer is doing anything with random numbers.

Write Your Program

Of course, you may just sit at the console and begin programming your game and hope you can remember all the logic. Some programmers like to draw a flowchart. On logic games you may like "tree diagrams"—i.e., if the player does one option you branch one way; then depending on the next choice, you branch again and so forth. Other programmers prefer a structured approach—each process of the game is in a subroutine and the main program calls the subroutines in order. This type of program is easy to evaluate and easier for other programmers to follow than a program that has GOTO statements all over the place.

Include Instructions

Many players are anxious to play the game and won't read anything that comes with the game program, so it is wise to include simple instructions within your program. Players who are playing the game a second time, however, won't want instructions, so you must try to satisfy everyone. One method is to print the instructions on one screen with "PRESS ANY KEY TO START" at the bottom of the screen. The player can then look at the screen as long as he wants or immediately press any key to start the game.

```
100 PRINT "PRESS ARROW KEYS TO GO"
110 PRINT "LEFT OR RIGHT."
120 PRINT "PRESS 'F' TO SHOOT."
130 PRINT :::"PRESS ANY KEY TO START."
140 CALL KEY(0,K,S)
150 IF S<1 THEN 140
160 Program continues for game.
```

Another method is to ask the player if he needs instructions:

```
100 PRINT "NEED INSTRUCTIONS? (Y/N)"
110 CALL KEY(0,K,S)
120 IF K=78 THEN 150
130 IF K<>89 THEN 110
140 Program prints instructions.
150 Program continues for game.
```

If the player presses Y, instructions will be printed; if he presses N, the game starts. Any other key pressed is ignored.

Be sure the instructions are as clear and concise as possible. Use enough blank lines to make the instructions easy to read. Make sure words are not divided at the ends of lines, be sure to spell correctly, and use correct grammar.

"Dummy-Proof" Your Game

A nicer way of saying this is make your program "user-friendly." This means consider all possibilities of input. You never know what some other player will try to do. If he has to answer "yes" or "no," can he just press Y or N, or does he need to spell out and ENTER the answer? Pressing one key makes for less chance of error than using INPUT. What if the game asks for a *number*, and a *letter* is pressed? What if the game asks for a choice of 1 through 4, and the number 7 is pressed? If the player needs to use the arrow keys, is there a default value if he hits the wrong key, or is that key ignored—or worse yet, does the program crash?

Check for Speed and Captivation

You don't want the player to fall asleep between moves. If you have moving objects in your game, he wants them to be as fast as possible. The main hints here are to make the moving object just one character and to minimize the logic between moves. Remember, the more objects you have to move, the longer it will take. And if you don't need to worry about scrolling (lines moving up the screen), PRINTing characters is faster than CALL HCHAR or VCHAR.

Look Through your Listing

If you use the same group of lines several times, use a GOSUB and place the subroutine near the beginning of the program. For example, a subroutine to print a message M$ on Row X starting in Column 1 is

```
180 FOR J=1 TO LEN(M$)
190 CALL HCHAR(X, J,ASC(SEG(M$(J,1)))
200 NEXT J
210 RETURN
```

Within the program, the message and row numbers are defined, then the subroutine called:

```
1740 M$="BLUE WINS THIS TIME."
1750 X=22
1760 GOSUB 180
```

Check for unnecessary statements. I have seen a few listings that contain some coding that can never be executed or is superfluous, or a subroutine that is never called. Other cases may occur because of editing. For example:

```
900 GOTO 920
910 X=25
920 GOTO 980
```

Or:

```
900 GOTO 910
910 Z=Z+1
```

Or:

```
900 IF X=A THEN 700 ELSE 910
910 GOTO 980
```

Test Your Game

Again, check all possibilities. If you say your spaceship can move to the right and to the left, be sure to check *both* directions. Make sure positive and negative numbers work correctly in your calculations (you may want to use the ABSolute function). Check the scoring to see if it is adding correctly. Test the possibility of hitting the wrong key. Test moving objects at the edges of the screen.

Specific Game Coding

Random Numbers

Be sure to use the statement RANDOMIZE before using RND so each game played will be different. If random numbers are computed in several different places, consider using RANDOMIZE before each RND to ensure total randomization throughout the game. Sometimes a single RANDOMIZE statement at the beginning of the program does not work.

A simulation of rolling the dice would need a random number between 1 and 6:

```
100 RANDOMIZE
110 D1=INT(RND*6)+1
```

In a space program or skill-type game you may want to place obstacles at random positions. If you have several objects, DEFine a few functions at the beginning of the program, so they can be used more easily in the coding later:

```
100 DEF RX=INT(RND*24)+1
110 DEF RY=INT (RND*29)+2
120 CALL CLEAR
130 RANDOMIZE
140 FOR I=1 TO 5
150 CALL HCHAR(RX,RY,65)
160 NEXT I
170 CALL VCHAR (RX,RY,66)
180 STOP
```

The DEFinition statements must be numbered lower than the statements in which the functions are used. Lines 140-170 place five A's and one B in random X and Y positions for X from 1 to 24 and Y from 2 to 30.

Another use of random numbers is choosing a random message or procedure. For example,

```
500 PRINT A$(INT(RND*9)+1)
```

chooses one of nine messages previously stored in the A$ array. For random subroutines, the coding would be

```
510 ON INT(RND*5)+1 GOSUB 220,250,300,350,400
```

Games using a deck of cards may use an array to keep track of which cards are dealt. You may use C$(52) for the 52 cards or a two-dimensional array C(13,4) where the first parameter is the number chosen and the second is the suit. An example for choosing ten cards follows. The values in the card array are initially zero. As a card is chosen, the corresponding C element is set equal to 1. In the following example I printed the card values, but you really should use the TI graphics to *draw* the cards.

```
100 REM  CARDS
110 CALL CLEAR
120 DIM C(13,4),A$(13)
130 DATA ACE,2,3,4,5,6,7,8,9,10,JACK,Q
    UEEN,KING
140 FOR J=1 TO 13
150 READ A$(J)
160 NEXT J
170 SUIT$(1)="HEARTS"
180 SUIT$(2)="CLUBS"
190 SUIT$(3)="DIAMONDS"
200 SUIT$(4)="SPADES"
210 PRINT "TEN CARDS CHOSEN:"::
220 RANDOMIZE
230 FOR I=1 TO 10
240 N=INT(13*RND)+1
250 S=INT(4*RND)+1
260 IF C(N,S)=1 THEN 240
270 PRINT TAB(6-LEN(A$(N)));A$(N);TAB(
    7);"OF";TAB(10);SUIT$(S)
280 C(N,S)=1
290 NEXT I
300 STOP
```

One more use of RND is for choosing random sounds. The CALL SOUND statement requires a frequency between 110 and 44733. Of course, most people cannot hear frequencies above 15,000; however, your dog may enjoy the higher frequencies. This statement plays a sound frequency between 110 and 2109:

```
300 CALL SOUND (200,INT(RND*2000)+110,0)
```

You may wish to use random sounds while you're placing objects randomly on the screen.

Sound and Noise

A lot of the fun in programming games is choosing the sound effects to fit your game. The following is a program that demonstrates the "noises" available on the TI-99/4A:

```
100 REM NOISE
110 FOR I=-1 to -8 STEP -1
120 CALL SOUND(4000,I,0)
130 CALL CLEAR
140 CALL SCREEN(ABS(I)+2)
150 PRINT "NOISE NUMBER";I
160 NEXT I
170 GOTO 110
180 STOP
```

Listen to these noises and choose what you need for your game. You can make crashing noises, explosions, airplane

or car motors, splats, bounces, rocket boosters, missile fire, or whatever you need. The noises may be varied by adding another set of sound frequencies and loudnesses.

Time

Since the 99/4A does not have an accessible real-time clock, time may be simulated by placing a counter in the CALL KEY routine or another loop that is executed regularly. The following example shows a counter as you move the asterisk up and down with the up and down arrows (E and X) keys. After a time of 100, the number of moves you have made is printed. You will notice that if you press a key, the counter moves more slowly than if no key is pressed, so the counter is not as even as a metronome but good enough for games.

```
100 REM  TIMING
110 CALL CLEAR
120 X=12
130 CALL HCHAR(X,15,42)
140 TIME=0
150 CALL KEY(0,K,S)
160 TIME=TIME+1
170 FOR I=1 TO LEN(STR$(TIME))
180 CALL HCHAR(22,I+3,ASC(SEG$(STR$(TI
    ME),I,1)))
190 NEXT I
200 IF TIME=100 THEN 350
210 IF K<>69 THEN 240
220 DX=-1
230 GOTO 260
240 IF K<>88 THEN 150
250 DX=1
260 CALL HCHAR(X,15,32)
270 X=X+DX
280 IF X>0 THEN 300
290 X=24
300 IF X<25 THEN 320
310 X=1
320 CALL HCHAR(X,15,42)
330 MOVES=MOVES+1
340 GOTO 150
350 PRINT "MOVES= ";MOVES
360 STOP
```

Following is another example of a way to time a process—in this case, typing your name.

```
100 REM  SPEED TEST
110 CALL CLEAR
120 TIME=0
130 PRINT "TYPE NAME THEN PRESS ENTER"
    ::::::::
140 FOR Y=3 TO 28
150 CALL KEY(0,K,S)
160 TIME=TIME+1
170 IF S<1 THEN 150
180 IF K=13 THEN 210
190 CALL HCHAR(21,Y,K)
200 NEXT Y
210 PRINT "TIME= ";TIME
220 STOP
```

An accurate way to delay for a specific length of time in your program is to use CALL SOUND for the number of milliseconds you need. Use 30 for the volume level and a very high frequency if you don't want to hear anything. While the CALL SOUND statement is being executed you may also be doing graphics of calculations. To end your timing device you will need another sound statement with a duration of 1. The following example illustrates how the CALL SOUND statements may be used for a rocket countdown.

```
FOR I=10 TO 1 STEP -1
CALL SOUND (1000,44000,30)
120 PRINT I
130 NEXT I
140 CALL SOUND (1,44000,30)
150 Program continues for rocket blastoff.
```

Arrow Keys

In games where you move a character up, down, left, or right, you may wish to have the player press the arrow keys. The arrows are on the keys E, D, X, and S. A CALL KEY statement is used to receive the player's input; then the program branches, depending on which arrow is pressed. Any other key pressed should be ignored so your program doesn't crash with bad values.

The following routine will draw a trail of asterisks as you press the arrow keys. Remember, you must consider the edges of the screen or you will get a "BAD VALUE" message. Lines 270-340 test for the edge values and will keep the asterisk at the edge position.

```
100 REM  MAKE-A-TRAIL
110 CALL CLEAR
120 X=12
130 Y=15
140 CALL HCHAR(12,15,42)
150 CALL KEY(0,K,S)
160 IF K<>69 THEN 190
170 X=X-1
180 GOTO 270
190 IF K<>68 THEN 220
200 Y=Y+1
210 GOTO 270
220 IF K<>88 THEN 250
230 X=X+1
240 GOTO 270
250 IF K<>83 THEN 150
260 Y=Y-1
270 IF X>=1 THEN 290
280 X=1
290 IF X<=24 THEN 310
300 X=24
310 IF Y>=1 THEN 330
320 Y=1
330 IF Y<=32 THEN 350
340 Y=32
350 CALL HCHAR(X,Y,42)
360 GOTO 150
370 STOP
```

Remember that there are many ways of coding to get the same result, and the examples presented here are just that—*examples*. The following routine illustrates another way to use the arrow keys to move a character. This time the previous character is deleted. Also, lines 330-410 will make the asterisk scroll to the other side of the screen instead of staying at the edge.

```
100 REM  MOVE A STAR
110 CALL CLEAR
120 X=12
130 Y=15
140 CALL HCHAR(X,Y,42)
150 CALL KEY(0,K,S)
160 IF K<>69 THEN 200
170 DX=-1
180 DY=0
190 GOTO 310
200 IF K<>68 THEN 240
210 DX=0
220 DY=1
230 GOTO 310
240 IF K<>88 THEN 280
250 DX=1
260 DY=0
270 GOTO 310
280 IF K<>83 THEN 150
290 DX=0
```

```
300 DY=-1
310 CALL HCHAR(X,Y,32)
320 X=X+DX
330 IF X>0 THEN 350
340 X=24
350 IF X<25 THEN 370
360 X=1
370 Y=Y+DY
380 IF Y>0 THEN 400
390 Y=32
400 IF Y<33 THEN 420
410 Y=1
420 CALL HCHAR(X,Y,42)
430 GOTO 150
440 STOP
```

A more compact approach to automatic scrolling is to replace lines 330-360 and 380-410 with these two lines:

```
330 X=INT(24*((X-1)/24-INT((X-1)/24)))+1
380 Y=INT(32*((Y-1)/32-INT((Y-1)/32)))+1
```

Split Keyboard

A split keyboard is used when two competing players or teams are interacting with moving objects on the screen. Instead of CALL KEY(0, KEY, STATUS), you will need to recieve input with CALL KEY (1, KEY1, STATUS1) and CALL KEY(2, KEY2, STATUS2). You may wish to use a *Video Games 1* Command Cartridge overlay for the arrow keys. You'll notice the arrow keys for the right side of the keyboard are keys I, J, K, and M. The key codes returned in CALL KEY are 5 for up, 2 for left, 3 for right, and 0 for down for both sides of the split keyboard. *Note:* There is a slight problem in testing for zero on the 99/4A console, so use logic such as IF KEY2 + 1<>1 instead of IF KEY2 <>0. It also seems wise to avoid using SHIFT, ENTER, G, B, slash, semi-colon, comma, periods, and the space bar for key input (such as firing a missile) because the key codes for these keys are different on the 99/4 and 99/4A. You will want your game to work on both consoles so you can share with others.

An example of the logic for two players and a split keyboard is shown in lines 910-1510 from the game *Maze Race* in the section "Computer Gaming."

Joysticks

Enter the sample programs that come with your TI Wired Remote Controllers to get an idea how to program movement with one or two joysticks. Keep in mind that CALL JOYST (KU, X, Y) returns X and Y values of 0 and plus or minus 4, depending on the position of the lever. By the way, don't get these X and Y values confused with X- and Y-coordinate values for HCHAR and VCHAR.

Following is a sample program that allows the player to move the asterisk with either the arrow keys or a joystick. Line 150 is a CALL KEY statement. If no key on the keyboard is pressed, all the arrow key logic is skipped and CALL JOYST (line 330) is executed. If a key has been pressed, then the joystick logic statements (lines 330-350) are skipped. (Remember: ALPHA LOCK up for joysticks, down for arrow keys.)

```
100 REM   JOYSTICKS
110 CALL CLEAR
120 X=12
130 Y=15
140 CALL HCHAR(X,Y,42)
150 CALL KEY(0,K,S)
160 IF S=0 THEN 330
170 IF K<>69 THEN 210
180 DX=-1
190 DY=0
200 GOTO 360
210 IF K<>68 THEN 250
220 DX=0
230 DY=1
240 GOTO 360
250 IF K<>88 THEN 290
260 DX=1
270 DY=0
280 GOTO 360
290 IF K<>83 THEN 150
300 DX=0
310 DY=-1
320 GOTO 360
330 CALL JOYST(1,JA,JB)
340 DX=-JB/4
350 DY=JA/4
360 CALL HCHAR(X,Y,32)
370 X=X+DX
380 Y=Y+DY
390 X=INT(24*((X-1)/24-INT((X-1)/24)))
    +1
400 Y=INT(32*((Y-1)/32-INT((Y-1)/32)))
    +1
410 CALL HCHAR(X,Y,42)
420 GOTO 150
430 STOP
```

Detecting a Crash

Probably the most common way of determining if your moving object hit some obstacle in position X, Y is by using CALL GCHAR(X ,Y ,C). The C value returned is the character number occupying positon X, Y on the screen. For example, you may then test if C = 32 (space); if so, the program could continue. But if C = 96 (one type of object), the program would branch one way, and if C = 99 (another object) the program would branch another way—with the appropriate sounds and graphics.

Another method of determining the character in a certain position is to have the screen positions in an array and have each array element contain information about the character in that position. For example, you may have an array A(24,32) for the 24 rows and 32 columns of the screen. Each element of A could be zero for a space and 1 for a block in a maze. Your testing statement would look like

```
200 IF A(X,Y)=1 THEN 240
```

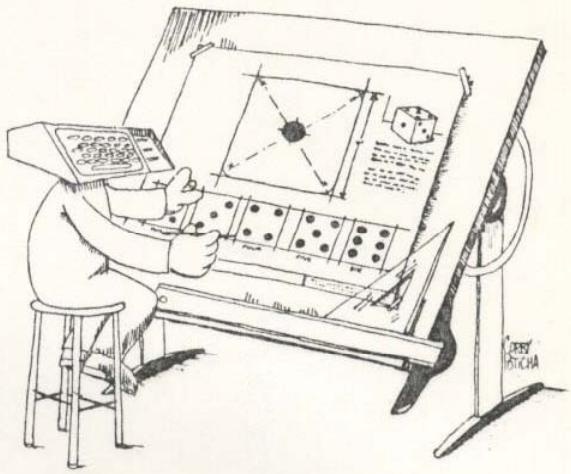
This means if the position X ,Y is a block, then branch to line 240 where a crashing noise is made and appropriate action takes place. Note that by using OPTION BASE 1, you will eliminate Row 0 and Column 0 and save memory space.

Do It!

I've presented some fundamental hints and ideas for programming; now it's your turn to put on your thinking cap, turn on the computer, and have fun writing your *own* games!

CHUCK -A- LUCK

HOW PROS PROGRAM



PART 1: "A bad beginning makes a bad ending"

Have you ever LISTed a program that you bought, and after looking at the listing thought, "That's not so hard. . ."?

Actually, you're quite correct in *believing* that writing a good program is not really so difficult. But when beginning programmers sit down to translate this belief into a finished program, many wind up confused and frustrated. This can most often be attributed to their accompanying belief that writing a program is just a matter of sitting down at a keyboard and banging away at it—a procedure that is destined to fail.

To explain an alternate approach—one that all experienced programmers use—I'll list the sequence of events that I go through whenever I want to write a program.

First, I sit down and decide what the program is going to do. If it's a game, I write down all the rules (even if I'm making the game up). If it's a business-oriented program, I decide what features it has to have—i.e., sorting, saving data, or printer output. Without this initial planning, I wouldn't have a goal in mind when I reached subsequent stages.

Second, I design the program. A design is a plan showing the functions (the "whats") that a program contains. For example, a program that plays the game of *Chuck-A-Luck* would contain the following functions: (1) explaining the rules, (2) rolling dice, (3) accepting bets, (4) paying off (or collecting) money, and (5) checking for the final win/loss condition. (See Figure 1 for the rules of the *Chuck-A-Luck* game that I'll be using as an example.) I don't figure out *how* I'll do these things at this time; I just figure out *what* the program has to do.

Third, I group together any "whats" that I feel are different parts of the same top-level module or function. For example, giving the rules, generating the dice characters, and getting player names are all part of initialization; so at first I put them together under the top-level function name of START-UP. Now, I'll write these functions down in a list. For this simple game of *Chuck-A-Luck*, my list of top-level modules looks like this: START-UP, DICE-ROLLS, and END-GAME.

Next, I look closely at each function (or module) and list everything I need to do in each of these modules. For example, the START-UP function will also have to include things like DIMensioning data, asking if rules are needed, asking the number of players, and initializing data fields. The DICE-ROLLS module will have to take bets for each roll, roll the dice (and display them), decide the winners and losers, and recompute new cash balances for each player. The END-GAME routine will have to print an appropriate message after all players go broke or a winner is determined, ask if any player wants to try again, and restart the game.

Notice that all I have done so far is write down the "whats" of the program. I haven't looked at the "hows" yet. The technique I have been using is called *top-down design* and consists of breaking a problem or program into its component modules. These new modules are themselves broken down into even smaller ones until you finally arrive at reasonably sized, easily codable low-level modules.

Sometimes, as you break modules into smaller and smaller routines, you may find at the lowest levels that some modules are duplicated. That means that the same module can belong to (or be used by) more than one higher level module. This kind of routine is called a *subroutine*. A good example of a subroutine that you would code in TI BASIC would be a routine to display messages on the screen using CALL HCHAR. It would

Figure 1. CHUCK-A-LUCK Rules

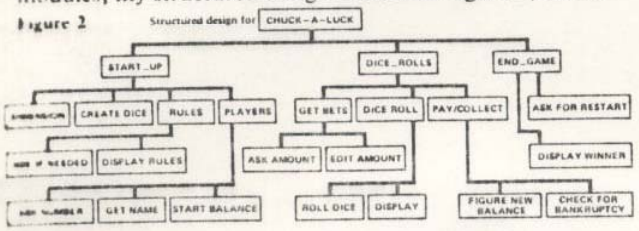
1. Each player starts with $500.
2. Each player bets an amount of money from $10 to $50 on a dice value from one to six.
3. Three dice are rolled.
4. If no die has a value equal to the value selected by a player, he loses his bet.
5. If one die has a value equal to the value selected by a player, that player receives an amount equal to the amount that he bet.
6. If two dice have that value, the player receives twice the amount he bet.
7. If three dice have that value, the player receives three times the amount he bet.
8. A player who goes bankrupt is out of the game.
9. The game ends when only one player remains. The remaining player is the winner.
10. If all the remianing players go bankrupt at the same time, there is no winner.

be called by other routines in your program (using the GOSUB statement) whenever they wanted a message displayed on the screen.

But when do we stop *designing* and actually start *writing* the program? There's a different answer for each program and programmer. The idea here is to stop at a point where you feel that you can picture in your mind what the code should look like. For advanced programmers, this may mean that there are fewer modules in a design, and each module will have a lot of lines of code in it. For beginners, I would recommend stopping when each module is self-explanatory to you—usually this requires about 10-20 lines of BASIC code.

As I am doing my design, I keep track of the modules by drawing a structured design chart which shows which lower-level modules belong to (are to be part of) each top-level module. After going over all of my top-level modules, my structured design looks like Figure 2, below.

Figure 2



Take a look at the START-UP module. It includes a low-level function called GET NAME. Now look at DICE-ROLLS. It includes a low-level module called EDIT AMOUNT. These routines demonstrate two rules I always follow when I design programs: First, whenever possible, I try to make the program "user-friendly." This includes things like displaying understandable error messages (instead of a cryptic "NOT POSSIBLE"), using player names (instead of numbers), and giving prompts that explain what action is required. Too many people write programs that call you PLAYER #1 and tell you to do something by saying things like "CODE?". It takes only a little longer to write a program that says "OK, MIKE, HOW MUCH WILL YOU BET THIS TIME?" And the results are well worth the effort.

Of course, in a business-oriented program, you don't usually ask for people's names. But such a program can become user-friendly just by judicious use of self-explanatory prompts and error messages. Of course, being user-friendly makes for longer and larger programs. I personally don't worry about how much extra memory it requires at first. After all, I can always remove those wonderful messages and replace them with a "NOT POSSIBLE" message if I have to!

By the way, if you stop to think about it, the TI BASIC and Extended BASIC that you work with is very user-friendly. It does things like prompt you for cassette tape I/O and give you meaningful error messages when you are in EDIT or COMMAND mode.

The second rule that I always follow is that any data that is input into a program must be fully edited—i.e., it must be checked to make sure it is the proper type and in the proper form. Always! Always! Always! I said it three times because this is one of the major differences between a professional program (which can be used by anyone without "blowing up"—especially when encountering some strange input from an unfamiliar user) and a program which is usable only by the person who wrote it.

Some of the rules that I always like to follow include:
1. Make sure that numeric data really is numeric (of course this is something the TI BASIC does for you automatically).
2. Make sure that integers really are integers (and not decimals or scientific notation).
3. Make sure that the data itself is realistic (always test for maximums and minimums—e.g., making sure nobody bets more money than he has!).

I'm now finished with my design as far as *what* modules are needed. The fifth step in creating a program is to decide what information I need to communicate between these modules. The information that is passed from one module to another is called a *variable*. And deciding on what variables are needed before you sit down to write program code is just as important as deciding what modules you need. If you make a mistake in your design variables, the last phase of programming (called debugging) will take twice as long as it needs to be. This is because whenever you realize that you need a new variable, you have to make coding changes in modules (that have already been coded) in order to handle them. And changing code is what destroys well-written programs!

Programs will also need variables that are used only within a module (i.e., things like loop counters), but you don't have to worry about them during your design. As long as a variable that is only used within a module has a unique name (not used again in another module), then no problems should arise when debugging. Of course, if the variable name will be used again in another module (which is a bad idea unless memory is, tight), then it is just as important as a regular variable's.

The variables that I need to communicate between my *Chuck-A-Luck* modules are
The number of players
The player names
The cash each player has on hand
The amount bet by each player
The dice value on which each player bets
The value of each die

Choosing the names for these variables is equally important. A poorly chosen name is asking for trouble when you get down to writing and debugging your code. A good variable name has the following three attributes:
It is long enough to say what it is and what it's for.
It is short enough so as not to slow down the program.
It does not look too similar to any other variable.

For the *Chuck-A-Luck* game, I'll use NO_PLAYERS, PLAYER_NAME, PLAYER_CASH, PLAYER_BET, PLAYER_DICE, and DICE_VALUE as my variable names. And, I won't re-use variables that are used within a module.

Now my design is finally complete, and I'm ready to start coding. I have done everything that I could to insure that the program will do its job and am ready for the sixth step in creating a good program—planning the code. As we have just learned, the first rule of good programming is PLAN, PLAN, PLAN!

CHUCK-A-LUCK PART 2: "Make no little plans. . . ."

Building a good program is a lot like building a house. First, you need a good design. Then, you need good tools, good materials, and good work habits to use them all properly. We have discussed a way to develop a good design by using a techinque commmonly called *structured design* or *top-down design*. Now we'll talk a little about how to get the necessary tools and materials and cultivate the habits that we need.

After completing our design effort, you might expect the next step to be coding the program. But this, in fact, is not the case. Just having a good design doesn't mean that the code in your program will be correct or that you will write the best code for the job. In every task, there are two things to remember: The first is that you want to do the right *job*. The second is that you want to do the job *right*. To do the right job means that your code has to follow the design that you came up with. To do the job right, you have to create the best code for the job. And like anything else, these both require planning. That's right! We still have some planning to do. Only this time, we must plan our code.

The first thing to do is refresh your memory on the design we came up with to play *Chuck-a-Luck*. Notice how we developed the modules that tell us *what* to do, but not *how* to do it. The purpose of planning our code is to figure out *how* we want to do it in the best possible way. At the same time, we want the "hows" that we develop to be easily coded and debugged, to execute quickly, and to be easily modifiable so that we can make future improvements.

Starting UP with START-UP

Let's start with the module called START-UP. One of its top-level components was DIMENSION. That module is needed to set up the dimensioning of any arrays needed in the program. Although it is not absolutely necessary to code the DIMension statement at the very beginning of your program, I have found that it is always best to put it right up front. So, when I plan my code, the DIMENSION module will be my very first line of code. Another good coding habit to get into is to start your programs at line 100, which leaves you room in the front of your program in case you have to add an extra statement to start off your program. I will reserve lines 100-140 for any dimensioning of data that I will need. But before I go on with the remaining design of the code, I think that we had better take time out to talk about the DIM statement and what it is used for.

When I was doing the design, I knew from my original plan that the program was going to have to handle 4 players. That meant that every time I did something concerning a player, I would have to know which player I was dealing with. For example, if each player was going to make a bet and win or lose, the program would keep track of these things (called variables) for each player. There are two ways of doing this. The first way is to give a different name to each one of these variables for each player. That is, I could keep track of each player's bet by having one variable called BET_1 and another variable called BET_2, and so on. This way, I would know at a glance what was contained inside the variable. The only problem with this way of doing things is that the program needs separate code for each player. This means that you would have to key in more lines of code. It means more chances for data entry errors. It also means the possibility that you could accidentally write the code for each player a little bit differently, which in turn means that you would need to debug your code for each player.

Suppose, however, that you did not need to give each player a different variable name. Suppose that you could just call the variable by the single name of BET. Then the code for each player would be the same. As a matter of fact, you would have to code the logic only once, because it could be re-used for each player. As you can see, this would be a great improvement. You still have more than one player so you would have to be able to say which player's BET you wanted to deal with. Well, the way that the BASIC language handles this is to allow you to set up an array called BET. This array has only one name but contains multiple slots. Imagine an apartment building called BET containing only one floor with a lot of rooms in it. The room numbers start with 0 and increment by one. The computer can put the betting information for player 1 into room number 1, the information for player number 2 into room number 2, and so on. Now, in order to look at the bet of player number 3, all we have to do is tell the computer to look at room number 3 of BET. We do this by saying BET(3). The value 3 is called a *subscript* of the array called BET.

This is an improvement over saying BET_3 but not much. But if the computer can be told which subscript (room number) to use via another variable, then you can realize a great improvement. Suppose all you had to do was tell the computer to look at something called X, and that X had the value of the subscript in it. Now you just put the room number inside X and tell the computer to look at BET(X). How do you put the room number into X? The same way you put any number inside any variable. You can say things like X=3 or X=A+B or set X to a range of values in a FOR-NEXT loop. The important thing is that you do not have to know in advance what is in X before you execute the code. By the way, I used the name X just as an example of my subscript name. We could have called it PLAYER_NUMBER, or I, or any other legal variable name. Also, just because a variable is being used as a subscript in one part of your program, it doesn't mean that the variable can only be used as a subscript. Any variable can be used as a subscript. It is also possible for two (or more) variables to be used as subscripts for the very same array, depending on what you are trying to do.

"Roomy" Arrays in TI BASIC

Now, two questions should be running through your head. The first question should be, "How many rooms can TI BASIC build for a given array?" The answer is that it depends on what (if anything) you tell it to do. If you don't tell it anything, it will automatically set aside 8 rooms (slots) for any array it may meet in your code. It will do this the first time it sees the array. If you need more than 8, you may not want to waste space on unused slots. In either case—less than 8 or more than 8 with no waste space—you tell it how many slots you need by using the DIM statement.

The second question should be, "What about room 0?" In my game, it is always empty. In some programs, however, room 0 may very well be used. If room 0 is not going to be used at all, you can tell this to TI BASIC so that it won't waste computer memory with a room 0. This is done by

ing a statement with OPTION BASE 1 in front of the
M statement.

Of course, just as an apartment building can have rooms more than one floor, an array can have more than one el of slots. But since we don't need multiple levels in our gram, we'll leave a discussion of this to a later time. For v, let's look at our variable list and see what variables going to be arrays so that we will put them in our DIM ement. We will need to keep track of information for 4 players. In addition, there will be 3 dice and each will have a value. Look at line 100, of the TI BASIC ng that starts on page 59, to see how I coded the DIM ement for these arrays. Notice that you cannot use a "-" part of a name in TI BASIC. You can make variable nes with several words in them by using the underline racter ("_") to connect the words. For instance, I coded array as DICE_VALUE in this program.

aving Out the Difficult is Easier

We are not ready to begin planning the code for the rest the START-UP module. Because (by definition) this code l only be used once for each game, I like to keep it up away from the main logic of my program. For this son, I usually begin coding these one-shot modules at number 20000 to give myself a lot of leeway in case ave out a line of code or have to add another line during bugging. I always increment my statement numbers by or more. In addition, I also make sure that there are plen- of unused statement numbers between the end of one dule and the start of another.

The first module to be coded in START-UP is responsi- for creating the graphics for the dice. Naturally, you v expect me to give you the code. But I won't! You see, not really important that I do this right away; I can ays create the dice later after I am sure that the rest of program is working correctly. This is one of the impor- t advantanges of designing and planning your program. When you plan your code, don't rush right into figuring how the code in all your modules will look. First, decide at modules or parts of modules can be left out without ecting the program logic. For example, the code to display tructions can be added as the very last part of your pro- amming effort. A program usually will contain whole dules requiring complex code that can be replaced by easy le the first time through. After you are sure that the pro- m as a whole is working correctly, you can gradually lace the easy code with the complex code. Why? Because easier to find your mistakes in an easy program! So an portant "rule of thumb" is to always start out with an y version of your program. Then, as you add the dif- ult pieces, you at least know where to look if you hit a g in your debugging.

So, if I leave out all the graphics for now, what can I bstitute in their place? I can simply display the number each die instead of graphically showing the dice emselves. After the program is running, I will go back d add the graphics as well as any sound routines. Look what I am trying to accomplish this way:

1. By leaving out unnecessary code, I can get the pro- m up and running faster. This means that I can begin bugging my program earlier. This in turn means *easier* bugging because there is less code to go through.

2. By using easy code in place of complex code in some odules, I make it easier to debug the "guts" of my pro- gram. After knowing that the program runs correctly, I can begin replacing the easy code with the hard parts a little at a time. Then I will test only one or two new parts at a time. This means easier debugging because any problems will probably be due to the new code.

3. After ensuring that the main portions are running correctly, I can "fool around" with the hard portions without worrying that I will hurt the program's logic. For example, after I know that the program is running correctly by displaying the dice numbers, I can now experiment with how I want the dice themselves to be displayed. I can even come up with two versions—one for TI BASIC and a different one for Extended BASIC using sprites! I won't have to worry that adding different versions of this code will destroy my program.

4. By getting a version up early, I can see if my program is worth continuing. After seeing it in action, I may decide that it just isn't worth the effort to continue with the coding.

So for right now, I won't code the CREATE DICE routine but I will set aside lines 20000-20500 for the code later. The next module is called RULES and will be responsible for giving the rules when asked. One part asks if the rules are wanted; another displays them. Like the CREATE DICE module, the entire code for this module isn't needed now. But if I do code in the part asking if the rules are wanted, I can test this part of the logic. If the program you are writing is large enough, you may decide to leave both of these parts out on your very first try.

Since I have decided to code part of this module, I will lay it out in lines 21000-22000. The first thing I want to do is clear the screen. This will attract the players' attention and remove any "clutter" that may be on the screen from any previous program. It's always a good idea to start out your program with a CALL CLEAR statement. Notice that in my code in lines 21010-21050, I am asking the players for information *and telling them in what form I expect the answer to be!* Too often, a programmer will code his program so that he is expecting a particular answer, but never tell the person using his program what form the answer should be in. There is nothing more frustrating to a user than trying to figure out what the person who coded the program means when the program displays a message like CODE?, and what the valid values of the input are. You should try to develop the habit of explaining what data you are looking for and what legal values the program will allow as part of your code for an INPUT statement.

The next thing the module does is make sure that only the first character is going to be looked at. This is done by using the SEG$ function to strip off the rest of A$. One of my programming "rules of thumb" is to minimize the chance of a program user entering bad data. If I am only expecting a Y or N, I want to look at only the *first* character of the input. If the wrong answer is given, an appropriate message is displayed and the original question is asked again. The code to display the message will be eventually located in lines 21000-21990, but I'll just put in a REM statement to show where the code will be added later.

The next module (called PLAYERS in our design) is very important and easy to code, so I will code it in full the first time out. This is done in lines 22000-22330. Notice that it prompts the player for the required data in each case, and edits the input to insure that only valid data gets in. One of the main differences between a well written program and a poorly written one is the amount of editing done on in-

put. The "hows" of an edit for an alphanumeric field should always include a test for an empty field (called a *null string*). TI BASIC allows a null string to be entered in response to an input statement. This kind of string data can cause a number of problems in your program, expecially if you want to display the data on the screen. I always test for an empty field whenever I INPUT a string variable. That is what I am doing in line 22140.

It may also be necessary to limit the size of an alphanumeric field depending on how you want to display it on the screen. For example, you may want to limit the size of a player's name so that it fits on the same line as his cash balance. The best way to handle this is to check its length (using the LEN function) as part of your edit. If the player enters a name that is too long, you can tell him so, and ask that he enter a shortened version of his name. There is also, however, another way: You may shorten the field yourself by using the SEG$ function—as I do in line 22250.

When numeric fields are entered into your program, there are always four things you must edit for. First, you have to make sure that numeric data was entered. Luckily, TI BASIC will do this for you automatically so you don't have to write any code to test for this. You should get in the habit of immediately testing your input as follows: (1) check to make sure it isn't too large, (2) check to make sure it isn't too small, and (3) especially check to make sure it is a whole number (if that's what you are expecting). Look at my code in statements 22020-22030 to see what I mean. Also note that if the answer is illegal, I ask for the item to be re-entered. If you don't make it obvious that you want the data entered again, it is possible that the person using your program may not even know that he or she made a mistake and get confused on what to enter next.

The main portion of our design is called DICE-ROLLS; it is responsible for actually playing the game. First, it gets the bets from each player. Then it rolls the dice. Finally, it makes the payments to or collects the losings from each player who is still in the game. Since this code is executed often, I will place it in front of my program. The three main components are called GET BETS, DICE ROLL and PAY/COLLECT. The first two components will be coded as subroutines called from DICE-ROLLS. Line 210 calls GET BETS and line 230 calls DICE ROLL. The third module, PAY/COLLECT, will be coded as part of the DICE-ROLLS module.

Save the Unimportant for Mañana.

Why did I set the modules up this way? The answer to this question requires a little background in the style of coding that I have adopted. As you know by now, I have a number of set methods that I follow. One of these rules of thumb is that if I get a module that I will be expanding or replacing later, I set it up as a subroutine to be coded later. I just code in a GOSUB statement and keep going. If it is a module that has to be coded fully the first time around, I usually code it right then—unless it looks like something that is hard to code. In that case, I code in the GOSUB statement and hold off coding it in until I have to. I write my programs this way because I never want to tackle any code that will destroy my train of thought. After all, one of the reasons we did a design in the first place was to make sure that nothing important would be left out. So if I keep coding, I won't get sidetracked into worrying about the hard parts until I absolutely have to.

Lines 530-560 are used to figure out how many "hits" a player has after the dice are rolled. Notice that this is done using two FOR-NEXT loops, one inside the other. The inside loop in lines 530-560 checks to see if a player bet any of the dice numbers that came up. The outer loop from statements 250-760 controls which player we are looking at. For now, I won't code the full CHECK FOR BANKRUPTCY module. I will instead code a short module (statements 740-750) to check for bankruptcy and STOP the program if there's a loser. Notice how the use of arrays has made this code simple to write. Try to imagine what it would look like if I had to name each variable separately!

The module called END-GAME is also not very important to the main logic of the program, so I'll ignore it for now. This means that the only modules I haven't looked at are GET BETS and DICE ROLL. I coded them in lines 1200-1900 and 2000-2990 respectively. I am leaving a lot of room in the DICE ROLL routine because I still don't know exactly how I am going to do all of it. Oh, I know how to roll the dice, but I haven't gotten around to figuring out what the graphic display of the dice and the design of the screen will look like. . .and until I do, I can't really figure out all of the "hows" of this module. For now, I will code the DISPLAY routine to just show what the dice are.

In order to simulate rolling the dice, I will have to create three random numbers between 1 and 6. This is done using the RND function in statement 2110. Remember that RND is really random only when you start your program with a RANDOMIZE statement. We will eventually put this in statement 140. But until I have fully debugged my program, I will leave the RANDOMIZE statement out. Without it, the dice rolls will not be truly random. They will always follow the same pattern from the start of the program. This allows me to replay a game exactly the same way each time, so that if I find a bug and have to correct my code, I can test the corrected code under the same conditions that caused the bug in the first place. With the RANDOMIZE statement in my program, I may never hit the same conditions that caused the bug and won't be sure that I made the right correction.

After coding in these statements, you can find the result in Listing 1. Let's briefly review just what this program can and cannot do. First, it *does* play the game according to the rules of *Chuck-a-Luck*. It will handle the bets of up to 4 players. It will keep track of cash held by each player and declare a loser. Once I have this program debugged, I then have to plan what pieces I want to add next. The program is missing three important features: First, it stops as soon as one player goes bankrupt and it cannot be restarted without rerunning the program. Second, it cannot display the rules. Third, it is boring because it doesn't have any of the graphics and sound features that the TI-99/4 can add to a program to make it interesting.

Once I have written enough code to run at least a "stripped-down" version of the program, I should turn my efforts to debugging it. Only after I was reasonably sure that this version of the program was working properly, would I begin to add more code. I then would add one module at a time and retest. And that will be the subject of *Chuck-a-Luck*, Part 3.

CHUCK-A-LUCK PART 3: "I never make misteaks"

Don't laugh. All too often you find programs with errors as glaring as that in my first sentence. So let's correct it: I never make mistakes! Now, doesn't that sound egotistical? Nobody would have the nerve to say it out loud. But some people who write programs act like they never make a mistake while programming! The best programmers that I have met not only admit that they make coding errors, but they have also developed quick and efficient ways to find these inevitable mistakes—called *bugs* by programmers. As with everything else, we need a good plan—a "debugging" plan—to catch them.

In the last section, we wrote a large percentage of the code required to play the full game. As a matter of fact, the only important module not coded was the graphics routine. So obviously, it's time to bring on the bugs! WHOA! First we have to figure out how to test for the various bugs I KNOW are in there. Before we do this, let's stop and talk about the different types of bugs found infesting even the best programs.

The first bug that must be eradicated is the "Baddus Plannus." This bug hits programs that do everything (according to the design) correctly but don't achieve the desired result or implement all the rules that you originally laid out. For example, as soon as I began testing my original code for *Chuck-A-Luck*, I hit a situation that I had not planned for and which was outside the scope of the rules of the game. In my original list of rules, I said that a player's bet could be from $10 to $50. As soon as I began debugging my program, however, I immediately saw a flaw in the whole idea! If a player bets in anything other than $10 units, he may eventually wind up with less than $10 in his bankroll. In that case, he can't make a minimum $10 bet and yet he isn't bankrupt. When that happens, the player is in limbo and the whole idea of the game falls apart. A major disaster? No, not necessarily. You see, when you have a good design, these kinds of problems can usually be overcome. I could have changed the logic to allow a player to bet only multiples of $10; instead, I just changed the rules so that bets of less than $10 are allowed. You may have noticed that this change is already in the code found in the last section.

Note that I am not ashamed to admit this error. Indeed I expect something of the sort to happen whenever I write a program. So when I set up my debugging plan, the first few items on my list are tests of the rules. These items don't have to be the first things actually tested, but they must be tested by the time we finish debugging.

The second bug that creeps inside programs is the very evil "Baddus Designus." This guy shows up when the code *almost* does what you want. A sure sign that your program has this problem is that it doesn't do everything that you wanted it to. It may mean that you left out some modules needed to get the program running correctly. It could also mean that a piece of code needs more information (or variables) to do its job. In other words, you forgot (or missed) some facts when you were designing your code. This kind of bug is uncovered by making sure that each routine is thoroughly tested and also by ensuring that each routine is tested using different values in the variables.

The third bug is "Baddus Codus." This means that a piece of code doesn't work even though it has all the information it needs. There are a number of reasons for this kind of bug, but they all boil down to three major ones:

1. You didn't write code that TI BASIC or Extended BASIC understands (for example, you typed in misspelled keywords).

2. You don't really know how a particular feature of BASIC works. You expect it to do something that it just won't do. This can hit your code unless you are prepared to check the reference manual for the usage of BASIC statements that you are not thoroughly familiar with.

3. You wrote code that doesn't do the job. The code may be in the wrong sequence (i.e., you are zeroing out a number just before printing it out on the screen), or a piece of code line is missing, or you typed in the wrong variable name, or even keyed in the wrong variable letter. It all boils down to normal human error.

Bug Catching

If you are lucky, TI BASIC or Extended BASIC will catch some of your errors for you. But don't rely on it. The only good way to check for a case of "Baddus Codus" is to look over your code before running it and then carefully watch how your program behaves when you run it.

Since a test plan for each program depends on the particular code and therefore is unique, the best that I can do for you is list some rules to follow when making up your test plan and debugging your programs.

A. List the program and visually check the code. Review your code for incorrect spelling of variables, miscoded statements (i.e., missing double colons between statements in Extended BASIC), and incorrect CALL names. Fix any errors you find immediately. After you have done this, do it again. Then save this copy of your program to disk or tape before you run the program. This will protect your hard-earned code if your computer decides to "eat" your program on the very first test. Label this Version 1.

B. Write down the function of each major module. Under each module, list the range of valid variables. This should be done so that when you begin debugging, you can set up your tests using both the largest and smallest values possible for each module.

C. Set up a test for each major module. Write down what values you will input and what you expect the output values to be. If you don't write it down before you begin your test, you won't really know if a module is working correctly while you are debugging.

D. Decide whether or not you can use the BREAK command to test the module. In many cases, a routine or module can be tested locally. By that, I mean that the module uses only a few variables and that you can set some values for these variables at the start of the module and BREAK at the end. Then you can check to make sure that the results are correct by PRINTing them on the screen when the computer stops at the BREAK point. For example, suppose a routine starts at line 1000 and uses the variables X and Y as input. The routine is supposed to use these values to calculate the variable Z using some formula. You can test

this routine locally by adding the following code in the front and back:

1000 BREAK (replaces the REM statement at the start of the routine)
routine
code is
here
.
.
.
1100 BREAK

Now RUN your program and make X and Y whatever values you want them to be when the program initially stops at line 1000. When you type in CON, the machine will execute your routine and and stop at the second BREAK statement at line 1100. When your program stops, type in PRINT Z so you can look at Z's value. In fact, you may want to add the following program statement after the second BREAK: 1110 GOTO 1000. In this way, the routine will continually repeat so that you can test your code using a number of input variables without the trouble of having to execute the rest of the program each time. That's why I call it a *local* test. Just make sure you remove that extra GOTO statement as soon as you finish testing that module!

Of course this technique isn't possible with all routines, and in some cases, it's not worth the effort. Just keep in mind that it's one debugging tool that you can use. It also shows a good reason to get into the habit of writing very straightforward code. In a routine, you should try to minimize GOTO statements which take the program outside the routine. If the routine above had GOTO statements that jumped outside the routine, it would be almost impossible to test the routine locally, because you could never be sure that your program would reach statement 1100. Although program size limitations may force you to reuse code, write all your routines with only one entry point and one exit point if possible.

E. Begin your tests. Carefully note any time that a routine does not give a correct result. Don't stop the program (using the Shift C or FCTN4 keys) each time you notice a problem. Just note the nature of the problem and what the program was doing at the time. For example, if you notice a problem in a routine only when the second player is betting, or if the dice roll is a 6, this is very important information and you should make sure that you write it down. Wait until you have uncovered a number of problems or until the computer stops with a BASIC error message.

F. Check each routine where an error occurred. Mentally "walk" through the code by doing each instruction or calculation on a piece of paper. Usually, you will find your errors this way quite easily. When you locate the error, write down the line number and the solution *but don't key it in!* This is because as soon as you change any of the code in a program statement, BASIC will reset all of the variables to 0 (for numbers) or empty (for strings). This may make it impossible to debug some other routine during the same test run. If you cannot find the bug by walking through the code, look at any intermediate results that may be available by PRINTing any intermediate variables. You may be able to find your mistake this way. This works especially well in complex code with a lot of intermediate totals.

G. If you get to a very difficult spot where the code looks OK, but you are sure it contains an error, don't panic! Use the BREAK *xxx* command, where *xxx* is an actual line number. This allows you to stop the program every two or three lines. At every BREAK, PRINT the important variables, and write them down along the line number of the BREAK. Then type in another BREAK *xxx* command, using a line number two or three lines further along. Type in CON and wait for the program to stop again. You can usually narrow the problem down to a single line this way. If you can't find a misspelling or other typographical error, re-enter the program line very carefully when you have finished this round of debugging. This will likely fix the error (as long as the code you are entering is good code).

H. When you have gone as far as you can in this test, fix all the bugs that you have discovered. Check off any of the tests that have successfully been concluded.

I. Save this new version of your program to disk or tape. I usually have a version number in a REM statement in the front of my programs. I increase this version number every time I change my code. This allows me to know what version of the program I have read into the machine when I begin my tests the next day. If you are saving to cassette tape, make sure you label the tape with the new version number. If you are using a disk, you may want to add the version number as part of the program name (i.e., SAVE DSK1.CHUCKV3). Making the version number part of your SAVE routine can save you some agonizing problems. There is nothing worse than realizing that you are debugging the same code that you fixed the day before.

J. As your program runs, review its actions against the rules and requirements that you originally set up when you began your plan. See if the results are what you expected. If they aren't, immediately stop testing and try to figure out why. You may have to change the rules. You may even have to redesign part of your program. It isn't worth testing any more until you fix this kind of problem.

K. If you get an idea to improve your program, write it down. Don't stop testing to make minor improvements. You may overlook a major flaw while adding a small feature. Add all of these improvements at one time, and revise your test plan to retest the old code as well as test the changes.

After my initial debugging, I began to add some of the modules that I left out the first time. The first routine I added checked to see if the game was over. This feature was added in lines 750, 770-890, and 5000-5400. I do this by checking each player's cash balance. If a player has a balance greater than zero, I increase a counter which tells me how many players are still in the game. I also save that player's number. That way, if only one player is left at the end of a round, I know who it is. If the game is over, I check to see if a replay is wanted. I also added the code at 21100-21500 which displayed the rules. I then retested the program to check both that the new modules worked and that they did not cause any damage to the old code in the rest of the program.

In the next section I'll explain how I added the graphics for both the TI BASIC and Extended BASIC versions. For now, you can study and type in the complete TI BASIC game listing that follows.

```
 60 REM **   CHUCK-A-LUCK    **
 70 REM *       TI BASIC      *
 80 REM
 90 REM
100 DIM DICE_VALUE(3),PLAYER_NAME$(4),
    PLAYER_CASH(4),PLAYER_BET(4),PLAYE
    R_DICE(4)
110 DIM DICE_PIP(9,9),LOC_X(27),LOC_Y(
    27)
120 RANDOMIZE
160 GOSUB 20000
170 REM BETTING LOOP
200 REM GET BET
210 GOSUB 1200
220 REM THROW DICE
230 GOSUB 2000
240 REM UPDATE CASH BALANCE
250 FOR I=1 TO PLAYERS
260 IF PLAYER_CASH(I)=0 THEN 760
280 PRINT "-":PLAYER_NAME$(I);", YOU BE
    T ON";PLAYER_DICE(I);"FOR";PLAYER_
    BET(I);"DOLLAR";
290 IF PLYER_BET(I)<2 THEN 310
300 PRINT "S";
310 PRINT "."
320 WIN=0
330 FOR J=1 TO 3
340 IF PLAYER_DICE(I)<>DICE_VALUE(J)TH
    EN 360
350 WIN=WIN+1
360 NEXT J
370 IF WIN=0 THEN 690
380 WIN=WIN*PLAYER_BET(I)
390 PRINT "YOU ";"WIN";WIN;"DOLLAR";
400 IF WIN<2 THEN 420
410 PRINT "S";
420 PRINT "."
430 PLAYER_CASH(I)=PLAYER_CASH(I)+WIN
440 PRINT "YOU NOW HAVE";PLAYER_CASH(I
    );"DOLLAR";
450 IF PLAYER_CASH(I)<2 THEN 670
460 PRINT "S";
470 PRINT "."
480 GOTO 760
490 PRINT "YOU LOST";PLAYER_BET(I);"DO
    LLAR";
500 IF PLAYER_BET(I)<2 THEN 520
510 PRINT "S";
520 PRINT "."
530 PLAYER_CASH(I)=PLAYER_CASH(I)-PLAY
    ER_BET(I)
540 IF PLAYER_CASH(I)>0 THEN 640
550 PRINT "YOU ARE BANKRUPT!"
560 NEXT I
570 REM CHECK FOR END OF GAME
580 GOSUB 5000
590 IF NO_LEFT>1 THEN 970
600 INPUT "WANT TO PLAY AGAIN (Y/N)?":
    A$
610 A$=SEG$(A$,1,1)
620 IF A$<>"Y" THEN 850
630 GOSUB 22000
640 GOTO 200
650 IF A$<>"N" THEN 880
660 PRINT "THANK YOU FOR PLAYING.":"-":
    "-"
670 STOP
680 PRINT PL$
690 GOTO 800
970 FOR I=1 TO 600
980 NEXT I
990 GOTO 200
1200 CALL CLEAR
1210 FOR I=1 TO PLAYERS
1220 IF PLAYER_CASH(I)=0 THEN 1500
1230 ON INT(RND*4+1)GOTO 1240,1260,1280
    ,1300
1240 PRINT "NOW, ";
1250 GOTO 1350
1260 PRINT "OK, ";
1270 GOTO 1350
1280 PRINT "ALRIGHT, ";
1290 GOTO 1350
1300 PRINT "YOUR TURN, ";
1350 PRINT PLAYER_NAME$(I);",";
1360 PRINT "YOU HAVE ";PLAYER_CASH(I);"
    DOLLAR";
1370 IF PLAYER_CASH(I)<2 THEN 1390
1380 PRINT "S";
1390 PRINT ".":"WHAT'S YOUR BET? "
1400 INPUT PLAYER_BET(I)
1410 IF PLAYER_BET(I)<1 THEN 1450
1420 IF PLAYER_BET(I)>PLAYER_CASH(I)THE
    N 1450
1430 IF PLAYER_BET(I)>50 THEN 1450
1440 IF INT(PLAYER_BET(I))=PLAYER_BET(I
    )THEN 1470
1450 PRINT "THAT'S NOT POSSIBLE."
1460 GOTO 1230
1470 PRINT "WHAT NUMBER WILL YOU BET ON
    ?"
1480 INPUT PLAYER_DICE(I)
1490 IF INT(PLAYER_DICE(I))<>PLAYER_DIC
    E(I)THEN 1520
1500 IF PLAYER_DICE(I)<1 THEN 1520
1510 IF PLAYER_DICE(I)<7 THEN 1540
1520 PRINT "TRY AGAIN."
1530 GOTO 1470
1540 NEXT I
1550 RETURN
2000 REM
2010 CALL CLEAR
2020 CALL SCREEN(10)
2030 FOR I=1 TO PLAYERS
2032 IF PLAYER_CASH(I)=0 THEN 2370
2035 GOSUB 28000
2040 ROW=(I-1)*5+1
2050 COL=15
2060 MSG$=PLAYER_NAME$(I)
2070 GOSUB 4900
2100 ROW=ROW+1
2110 COL=15
2120 MSG$="BET"
2130 GOSUB 4900
2150 COL=20
2160 MSG$="$"&STR$(PLAYER_BET(I))
2170 GOSUB 4900
2200 ROW=ROW+1
2210 COL=15
2220 MSG$="CASH"
2230 GOSUB 4900
2250 COL=20
2260 MSG$="$"&STR$(PLAYER_CASH(I))
2270 GOSUB 4900
2300 ROW=ROW+1
2310 COL=15
2320 MSG$="DIE-"
2330 GOSUB 4900
2340 COL=21
2350 MSG$=STR$(PLAYER_DICE(I))
2360 GOSUB 4900
2370 NEXT I
2500 FOR I=1 TO 3
2510 DICE_VALUE(I)=INT(RND*6)+1
2520 NEXT I
2600 REM DISPLAY DICE
2610 FOR I=1 TO 3
2620 CHAR_NO=DICE_VALUE(I)
2630 IF CHAR_NO=1 THEN 2740
2640 IF CHAR_NO=4 THEN 2740
2650 IF CHAR_NO=5 THEN 2740
2660 IF RND<.5 THEN 2740
2670 IF CHAR_NO<>2 THEN 2700
2680 CHAR_NO=7
2690 GOTO 2740
2700 IF CHAR_NO=6 THEN 2730
2710 CHAR_NO=8
2720 GOTO 2740
```

```
 2730 CHAR_NO=9
 2740 REM DISPLAY A DIE
 2750 FOR J=1 TO 9
 2760 K=(I-1)*9+J
 2780 CALL HCHAR(LOC_X(K),LOC_Y(K),96+DI
      CE_PIP(CHAR_NO,J))
 2790 NEXT J
 2800 CALL SOUND(20,1111,0,1166,0,1221,1
      )
 2990 NEXT I
 3800 FOR I=1 TO 400
 3810 NEXT I
 3900 CALL SCREEN(4)
 3910 CALL CLEAR
 3920 RETURN
 4900 FOR Z=1 TO LEN(MSG$)
 4910 CALL HCHAR(ROW,COL+Z+1,ASC(SEG$(MS
      G$,Z,1)))
 4920 NEXT Z
 4930 RETURN
 4990 REM CHECK FOR A WINNER
 5000 NO_LEFT=0
 5010 FOR I=1 TO PLAYERS
 5020 IF PLAYER_CASH(I)=0 THEN 5050
 5030 NO_LEFT=NO_LEFT+1
 5040 LAST_PLAYER=I
 5050 NEXT I
 5060 IF NO_LEFT>0 THEN 5200
 5110 PRINT "NO ONE IS LEFT.":"THE GAME
      ENDS IN A TIE."
 5120 GOTO 5400
 5200 IF NO_LEFT>1 THEN 5400
 5300 PRINT PLAYER_NAME$(LAST_PLAYER);"
      WINS!"
 5400 RETURN
20000 PL$="PLEASE ANSWER THE QUESTION"
20010 CALL CHAR(96,"0000000000000000")
20020 CALL CHAR(97,"0000001818000000")
20030 CALL COLOR(9,2,16)
20050 CALL CLEAR
20090 ROW=12
20100 FOR I=1 TO 9
20110 FOR J=1 TO 9
20120 READ DICE_PIP(I,J)
20130 NEXT J
20140 IF INT(I/2)=I/2 THEN 20170
20150 MSG$="CHUCK-A-LUCK"
20160 GOTO 20180
20170 MSG$="
20180 COL=10
20190 GOSUB 4900
20200 NEXT I
20300 CNT=0
20310 FOR I=1 TO 3
20320 FOR J=1 TO 3
20330 FOR K=1 TO 3
20340 CNT=CNT+1
20350 LOC_Y(CNT)=J+I*4
20370 LOC_X(CNT)=K+2
20400 NEXT K
20410 NEXT J
20420 NEXT I
21000 CALL CLEAR
21010 INPUT "NEED INSTRUCTIONS (Y/N)? ":
      A$
21020 A$=SEG$(A$,1,1)
21030 IF A$="Y" THEN 21100
21040 IF A$="N" THEN 22000
21050 PRINT PL$
21060 GOTO 21010
21100 PRINT :::::"WELCOME TO THE GAME
      OF":" CHUCK-A-LUCK!:"
21110 PRINT "THIS GAME CAN BE PLAYED BY"
      :"2 TO 4 PLAYERS. EACH PLAYER STAR
      TS OUT WITH $500. FOR"
21120 PRINT "EVERY TURN, EACH PLAYER BET
      S FROM $1 TO $50 ON A DICE VALUE
      FROM 1 TO 6. THREE"
21130 PRINT "DICE ARE THEN ROLLED. EACH
      PLAYER WILL THEN RECEIVE AN AMOUN
      T EQUAL TO THIS BET"
21140 PRINT "MULTIPLIED BY THE NUMBER OF
      TIMES THE VALUE HE SELECTED CAME
      UP. IF NO DIE HAS THE"
21150 PRINT "VALUE SELECTED, THE PLAYER
      LOSES HIS BET. A PLAYER WHO GOES
      BANKRUPT IS OUT OF THE"
21160 PRINT "GAME. THE GAME IS OVER WHEN
      ONLY 1 PLAYER REMAINS. IF NOONE R
      EMAINS, THERE IS NO"
21170 PRINT "WINNER.":::
21500 FOR I=1 TO 1000
21510 NEXT I
22000 INPUT "HOW MANY PLAYERS (2-4)? ":P
      LAYERS
22010 IF PLAYERS<2 THEN 22060
22020 IF PLAYERS>4 THEN 22060
22030 IF INT(PLAYERS)=PLAYERS THEN 22100
22060 PRINT PL$
22070 GOTO 22000
22100 FOR I=1 TO PLAYERS
22110 PRINT "PLAYER NUMBER";I;"ENTER YOU
      R"
22120 INPUT "NAME-":PLAYER_NAME$(I)
22140 IF PLAYER_NAME$(I)<>"" THEN 22250
22170 PRINT PL$
22180 GOTO 22110
22250 PLAYER_NAME$(I)=SEG$(PLAYER_NAME$(
      I),1,10)
22310 PLAYER_CASH(I)=500
22320 NEXT I
22330 RETURN
25000 DATA 0,0,0,0,1,0,0,0,0
25010 DATA 1,0,0,0,0,0,0,0,1
25020 DATA 1,0,0,0,1,0,0,0,1
25030 DATA 1,0,1,0,0,0,1,0,1
25040 DATA 1,0,1,0,1,0,1,0,1
25050 DATA 1,1,1,0,0,0,1,1,1
25060 DATA 0,0,1,0,0,0,1,0,0
25070 DATA 0,0,1,0,1,0,1,0,0
25080 DATA 1,0,1,1,0,1,1,0,1
25090 DATA X
28000 T2=700
28005 T=120
28010 CALL SOUND(T,392,1)
28020 CALL SOUND(T,523,1)
28030 CALL SOUND(T,659,1)
28040 CALL SOUND(T,784,1)
28050 CALL SOUND(T,784,1)
28060 CALL SOUND(T,784,1)
28070 CALL SOUND(T,659,1)
28080 CALL SOUND(T,659,1)
28090 CALL SOUND(T,659,1)
28100 CALL SOUND(T,523,1)
28110 CALL SOUND(T,659,1)
28120 CALL SOUND(T,523,1)
28130 CALL SOUND(T2,392,1)
28140 CALL SOUND(1,39999,30)
28150 CALL SOUND(T2,392,1)
28160 CALL SOUND(T,523,1)
28170 CALL SOUND(T,659,1)
28180 CALL SOUND(T,784,1)
28190 CALL SOUND(T,784,1)
28200 CALL SOUND(T,784,1)
28210 CALL SOUND(T,659,1)
28220 CALL SOUND(T,659,1)
28230 CALL SOUND(T,659,1)
28240 CALL SOUND(T,392,1)
28250 CALL SOUND(T,392,1)
28260 CALL SOUND(T,392,1)
28270 CALL SOUND(T2,523,1)
28280 RETURN
```

CHUCK-A-LUCK PART 4: "The Die is Cast"

Before me was the task I had been putting off from the beginning: to plan the graphics for the DICE-ROLL routine.

Because the program had been coded in TI BASIC all along, I coded this routine using HCHAR and VCHAR graphics. It occurred to me however, that the Extended BASIC graphics ability (i.e., sprites) would add a lot to this program. Then I saw that it could be done both ways.

The only problem was that I am not very graphics-oriented. Oh, I do all right, but I am no world-beater at eye-boggling displays. That left me one option: I called for HELP!! and turned to my "Guru of Graphics," Ron Binkowski. You may have seen his name on some fine programs he has written for *99'er Magazine*.

I asked Ron to develop a graphics routine to display dice rolling inside a Chuck-A-Luck wheel. About two weeks later, he called me back with the bad news, "No dice." (Pardon the pun—I just couldn't resist it). Rolling the dice was just too complicated for this program, but Ron did come up with an idea to move them graphically.

Starting to Roll

I reworked Ron's routine so that it could support both SPRITE graphics for inclusion in the Extended BASIC version and HCHAR graphics for TI BASIC. The design indicated that DICE-ROLL needed an initializing routine (to set up some variables) as well as the actual graphics roll itself. I added another module to display each player's name, cash balance, amount bet, and dice value bet.

The DICE-ROLL routine needed three new arrays. Each die can be thought of as a formation 3 pieces high and 3 pieces wide. Each character can have either a dot (*pip*) or be blank. Since there are three dice and each needs 9 characters, we will have to keep track of the locations of 27 characters. The 9 characters for the first die will be numbered 1-9; for the 2nd die, 10-18; and 19-27 for the third die. The array called LOC_X keeps track of the x-coordinates (the horizontal rows) of these characters while LOC_Y keeps track of the y-coordinates (the columns). This means that both of these arrays must be DIMensioned with 27 entries.

The array called DICE_PIP tells whether characters are blank or have a pip for each possible value of the dice. Since there are six possible dice values, each to contain information on nine characters, we will need a two-dimensional array composed of 9 entries for each of 6 possible dice-values.

Arrays are Like Buildings

Remember our discussion about arrays? I said that you can envision an array as a building with a number of rooms on each floor. Well, in a two-dimensional array, the first variable can be thought of as the floor number. The second number is the room on the floor. For example, you can think of DICE_PIP(2,4) as the value located in the 4th room of the second floor of a building called DICE_PIP. For our program it will contain the information about the 4th character (middle row on the left) needed to display a dice roll of 2.

To make the display more interesting, Ron added 3 more dice values. He realized that, depending on how a die fell, the values of 2, 3, and 6 could be portrayed two different ways. The three extra "floors" in DICE_PIP are alternate displays for the values of 2, 3, and 6. This meant the DICE_PIP had to be DIMensioned as (9,9). I added this at line 110. In addition, for the SPRITE version of the routine in Extended BASIC, Ron needed an array to keep track of particular pieces of the die, to determine if they were in position. He called this array LOC, and since there are 27 different pieces, I DIMensioned it at 27 in line 120.

I then added the code in lines 20010-20420 (see Extended BASIC listing starting on page 63) to fill in the data needed for the new arrays. Lines 20100-20200 are used to read in the data for DICE_PIP. Each DATA line (in 25000-25080) describes whether a character of a dice value is supposed to be blank (=0) or have a pip (=1). Each line gives the information needed for the 9 characters making up the dice value. Line 25090 is an extra DATA line. TI BASIC usually slows down when it reads the last DATA line in a program, but with an extra DATA statement, it never reads the last line, and never slows down.

In order to simulate the DISPLAY AT function, available only in Extended BASIC, I added a routine to the TI BASIC version in statements 4900-4930 to print whatever was in MSG$ beginning at COL on the row contained in ROW. It runs much faster than the code given in TI's *Programming Aids I* software package because it is restricted to a single row and does no preliminary editing of the message area. In lines 20300-20420, I added the codes to show where each of the 9 characters for each die are to be displayed. In the TI BASIC version, these are actual row and column numbers. In the Extended BASIC version, these contain the dot row and dot column values needed for sprites.

I then coded lines 2000-2370 to display the information about each player on the screen. The new code in 2600-3920 displays the three dice values graphically. Lines 2630-2740 give a 50-50 chance that a dice value of 2, 3, or 6 will be displayed in its alternate format. The 9 characters making up the die are then displayed in the loop in lines 2750-2990 in the TI BASIC version, and lines 2750-3020 in the Extended BASIC version. For TI BASIC, this consists of a simple loop which displays at LOC_X and LOC_Y the appropriate DICE_PIP for each of the nine pieces. After the character on the last die is displayed, I wait a little while and then leave the routine.

Notice that in order to highlight the dice roll routine, I changed the color of the screen and added a little music. My "music jar" of melody listings borrowed from other programs gave up only one piece that remotely matched up with gambling, the "Call to the Post" tune played at the track just before a race. Perhaps you have a more fitting musical phrase.

The sprite version of the display routine is more complex than the HCHAR version. I will go through it very carefully because Ron has some great ideas about controlling sprites. Note that this routine was written with multiple statements on each line. This has to be done to make your BASIC code run as fast as possible when handling sprites. Slow code at this point could make it very difficult to handle them smoothly.

Graphic Routines

If you have an interest in designing Extended BASIC programs with sprites, tracing through the following program will put you well on your way toward your own creative

endeavors. All line numbers from this point on will refer to the Extended BASIC version only.

2750-2820 This code figures out the sprite number for each character of the die being displayed and starts it out as a sprite with a random motion. Note that this motion can be either positive or negative so that we get them flying in all directions. We also set the LOC value for that character to zero, to show that we haven't yet moved the character back to its final location.

2840-2920 This routine uses a variable called CNT to keep track of the number of characters moved back to their starting locations. If this number is low enough, we will randomly choose the character we work on. If CNT is 21 or greater, however, we won't choose the character randomly. We'll just look through the LOC array sequentially to find the first character that we haven't yet moved back to its location (i.e., its LOC has a zero in it).

Why is Ron going through the trouble of doing it this way? The answer requires a little thought. Suppose we just randomly kept choosing a figure. By the time 20 or so characters have been reset to the final location, the odds on randomly selecting a good character will then be 7/27 or 26%. The odds on the next selection being a good character will then be 6/27 and they keep getting smaller and smaller. With one character left, the odds on hitting it randomly are 1/27 or less than 4%. As you can see, it is very unlikely that you will hit a good character when only a few are left. To prevent a long wait until the computer randomly locates a good character, Ron set up his code so that the last 7 or so sprites will not be randomly chosen. Of course, he is also checking CNT to see if he has finished with all 27 characters.

2930-2980 This part of the routine takes the selected character, changes its color to black (to highlight it on the screen while we play with it), and freezes it momentarily. That is what the CALL MOTION(#1,0,0) is for. The major problem in sprite handling is that they keep moving at a pretty high speed, while BASIC keeps plodding along with old data. Ron prevents this problem by freezing the sprite before finding its location. This means that he gets *accurate* data via the CALL POSITION code.

After locating the sprite, he computes the velocities needed to move it back to its original (and final) location. The Extended BASIC reference manual talks about row velocities and column velocities, but it doesn't explicitly tell you that you can control the direction of the sprite. For example, if you want to move a sprite at a 45 degree angle, both the row and column velocities must be equal. To move at a 30 degree angle, just make the column velocity equal to twice the row velocity. Ron is using this fact in statement 2960 to figure out how far the sprite is, vertically and horizontally, from where it is supposed to go. He calculates this in MY and MX respectively. He then adds the two to get a value called TOT. The distances can be positive or negative depending on the sprite's location relative to its final position—left or right, above or below.

In order to get a good value of TOT, we have to ignore the signs of the distances. In other words, we don't care if the number is positive or negative, as long as we know its *absolute* value. We find it with the ABS function. By making the row and column velocities a function of both the distance the sprite it has to go (MY or MX) and the TOT value, we can direct the sprite to travel in the right direction. Take a look at the last statement in line 2960. It uses the MAX function available in Extended BASIC. TOT must be a reasonably-sized number because we will divide MY and MX by TOT to get our velocities. Since it is possible for the sprite to be right where it should be, TOT can be zero. If you divide by 0, however, your program will stop with an error. To make sure that TOT has a value of at least one, you would normally code in something like this:

```
xxxxx TOT = ABS(MX) + ABS(MY)
yyyyy IF TOT < 1 THEN TOT = 1
zzzzz
```

This can be done just as easily with the MAX function, which gives the larger of the two alternatives. In this case, if 1 is greater than the result of the addition, it will return 1. On the other hand, if the result of the addition is greater than 1, it will return that number. Using the MAX function eliminates the need for an IF statement right in the middle of my code. MAX (along with its cousin, the MIN function) is a handy feature of Extended BASIC that can save you a lot of coding trouble. We now use the values that we just computed to set the sprite moving again using a CALL MOTION.

2990-3010 I have also set a new variable (my, we are collecting a whole slew of them now!) called CHK to be equal to zero. This counter will be used to make sure that we don't try the next lines of code more than 10 times before we give up and refigure a new MOTION command. If we haven't tried it more than 10 times, we do a CALL COINC to see if the sprite has reached its goal. If not (HIT = 0) we go back and do it again. If the sprite has reached its final location, Ron stops it with a CALL MOTION, and does a CALL LOCATE to make sure it is being stopped exactly where he wants it. This is necessary because a sprite that keeps moving between the CALL COINC and the final CALL MOTION may no longer be in the right spot. He changes the color back to white.

3020-3920 This code checks to see if we finished all the characters and restarts the process if we haven't. It then changes the screen back to green. It also issues a CALL DELSPRITE which clears the sprite characters from the screen.

Protection and Improvement

We have now finished the Extended BASIC version of the code. Our game gets a final debugging and is ready to go! The next step is just some administrative work to make sure that your effort will not be in vain. First, change the REM statement at the beginning of the program so that it says FINAL VERSION as well as the version number. Next, save it on cassette tape or disk. Label the tape or disk with the name of the program, the date, and the version number along with the words FINAL VERSION. Make two copies. If you are saving on tape, make one copy on each side and verify both. Then make another copy on a backup tape. You should always have a backup tape kept separately from your original master copies. Remove the tabs in the back of the tape to prevent accidental erasures. For disks, add the write-protect tab. Make a backup disk. Keep it separate from your regular disks. Then enjoy the fruits of your labor!

```
 60 REM     •••   CHUCK-A-LUCK   •••
 70 REM     •       EXTENDED BASIC      •
 80 REM
 90 REM
100 DIM DICE_VALUE(3),PLAYER_NAMES(4),
    PLAYER_CASH(4),PLAYER_BET(4),PLAYE
    R_DICE(4)
110 DIM DICE_PIP(9,9),LOC_X(27),LOC_Y(
    27)
120 DIM LOC(27)
130 RANDOMIZE
160 GOSUB 20000
170 REM BETTING LOOP
200 REM GET BET
210 GOSUB 1200
220 REM THROW DICE
230 GOSUB 2000
240 REM UPDATE CASH BALANCE
250 FOR I=1 TO PLAYERS
260 IF PLAYER_CASH(I)=0 THEN 760
280 PRINT "":PLAYER_NAMES(I);", YOU BE
    T ON";PLAYER_DICE(I);"FOR";PLAYER_
    BET(I);"DOLLAR";
290 IF PLAYER_BET(I)<2 THEN 310
300 PRINT "S";
310 PRINT "."
520 WIN=0
530 FOR J=1 TO 3
540 IF PLAYER_DICE(I)<>DICE_VALUE(J)TH
    EN 560
550 WIN=WIN+1
560 NEXT J
570 IF WIN=0 THEN 690
580 WIN=WIN*PLAYER_BET(I)
590 PRINT "YOU ";"WIN";WIN;"DOLLAR";
600 IF WIN<2 THEN 620
610 PRINT "S";
620 PRINT "."
630 PLAYER_CASH(I)=PLAYER_CASH(I)+WIN
640 PRINT "YOU NOW HAVE";PLAYER_CASH(I
    );"DOLLAR";
650 IF PLAYER_CASH(I)<2 THEN 670
660 PRINT "S";
670 PRINT "."
680 GOTO 760
690 PRINT "YOU LOST";PLAYER_BET(I);"DO
    LLAR";
700 IF PLAYER_BET(I)<2 THEN 720
710 PRINT "S";
720 PRINT "."
730 PLAYER_CASH(I)=PLAYER_CASH(I)-PLAY
    ER_BET(I)
740 IF PLAYER_CASH(I)>0 THEN 640
750 PRINT "YOU ARE BANKRUPT!"
760 NEXT I
770 REM CHECK FOR END OF GAME
780 GOSUB 5000
790 IF NO_LEFT>1 THEN 970
800 INPUT "WANT TO PLAY AGAIN (Y/N)?":
    A$
810 A$=SEG$(A$,1,1)
820 IF A$<>"Y" THEN 850
830 GOSUB 22000
840 GOTO 200
850 IF A$<>"N" THEN 880
860 PRINT "THANK YOU FOR PLAYING.":"":
    ""
870 STOP
880 PRINT PL$
890 GOTO 800
970 FOR I=1 TO 600
980 NEXT I
990 GOTO 200
1200 CALL CLEAR
1210 FOR I=1 TO PLAYERS
1220 IF PLAYER_CASH(I)=0 THEN 1500
1230 ON INT(RND*4+1)GOTO 1240,1260,1280
    ,1300
1240 PRINT "NOW, ";
1250 GOTO 1350
1260 PRINT "OK, ";
1270 GOTO 1350
1280 PRINT "ALRIGHT, ";
1290 GOTO 1350
1300 PRINT "YOUR TURN, ";
1350 PRINT PLAYER_NAMES$(I);","
1360 PRINT "YOU HAVE";PLAYER_CASH(I);"D
    OLLAR";
1370 IF PLAYER_CASH(I)<2 THEN 1390
1380 PRINT "S";
1390 PRINT ".":"WHAT'S YOUR BET? "
1400 INPUT PLAYER_BET(I)
1410 IF PLAYER_BET(I)<1 THEN 1450
1420 IF PLAYER_BET(I)>PLAYER_CASH(I)THE
    N 1450
1430 IF PLAYER_BET(I)>50 THEN 1450
1440 IF INT(PLAYER_BET(I))=PLAYER_BET(I
    )THEN 1470
1450 PRINT "THAT'S NOT POSSIBLE."
1460 GOTO 1230
1470 PRINT "WHAT NUMBER WILL YOU BET ON
    ?"
1480 INPUT PLAYER_DICE(I)
1490 IF INT(PLAYER_DICE(I))<>PLAYER_DIC
    E(I)THEN 1520
1500 IF PLAYER_DICE(I)<1 THEN 1520
1510 IF PLAYER_DICE(I)<7 THEN 1540
1520 PRINT "TRY AGAIN."
1530 GOTO 1470
1540 NEXT I
1550 RETURN
2000 REM
2010 CALL CLEAR
2020 CALL SCREEN(10)
2030 FOR I=1 TO PLAYERS
2035 GOSUB 28000
2040 ROW=(I-1)*5+1
2060 DISPLAY AT(ROW,15):PLAYER_NAMES$(I)
2160 DISPLAY AT(ROW+1,15):"BET   $";STR$
    (PLAYER_BET(I))
2260 DISPLAY AT(ROW+2,15):"CASH $";STR$
    (PLAYER_CASH(I))
2350 DISPLAY AT(ROW+3,15):"DIE-   ";STR$
    (PLAYER_DICE(I))
2370 NEXT I
2500 FOR I=1 TO 3
2510 DICE_VALUE(I)=INT(RND*6)+1
2520 NEXT I
2600 REM DISPLAY DICE
2610 FOR I=1 TO 3
2620 CHAR_NO=DICE_VALUE(I)
2630 IF CHAR_NO=1 THEN 2740
2640 IF CHAR_NO=4 THEN 2740
2650 IF CHAR_NO=5 THEN 2740
2660 IF RND<.5 THEN 2740
2670 IF CHAR_NO<>2 THEN 2700
2680 CHAR_NO=7
2690 GOTO 2740
2700 IF CHAR_NO=6 THEN 2730
2710 CHAR_NO=8
2720 GOTO 2740
2730 CHAR_NO=9
2740 REM DISPLAY A DIE
2750 FOR J=1 TO 9
2760 K=(I-1)*9+J
2780 CALL SPRITE(#K,96+DICE_PIP(CHAR_NO
    ,J),16,LOC_X(K),LOC_Y(K),RND*120-6
    0,RND*120-60)
2800 LOC(K)=0
2820 NEXT J
2830 NEXT I
2840 CNT=0
2850 IF CNT<21 THEN 2900
2860 FOR I=1 TO 27 :: IF LOC(I)=0 THEN
    2920
2870 NEXT I :: GOTO 3800
2900 I=INT(RND*27)+1
2910 IF LOC(I)=1 THEN 2900
```

```
2920  LOC(I)=1 :: CNT=CNT+1
2930  CALL COLOR(#1,2):: CALL SOUND(-1,1
      10,0,165,1,220,2)
2940  CALL MOTION(#1,0,0):: CHK=0
2950  CALL POSITION(#1,Y,X)
2960  MY=LOC_Y(I)-Y :: MX=LOC_X(I)-X ::
      TOT=MAX(1,ABS(MY)+ABS(MX))
2980  CALL MOTION(#1,MY*50/TOT,MX*50/TOT
      )
2990  CHK=CHK+1 :: IF CHK>10 THEN 2940
3000  CALL COINC(#1,LOC_Y(I),LOC_X(I),20
      ,HIT):: IF HIT=0 THEN 2990
3010  CALL MOTION(#1,0,0):: CALL LOCATE(
      #1,LOC_Y(I),LOC_X(I)):: CALL COLOR
      (#1,16):: CALL SOUND(-1,1111,0)
3020  IF CNT<27 THEN 2850
3800  FOR I=1 TO 400
3810  NEXT I
3900  CALL SCREEN(4)
3910  CALL DELSPRITE(ALL):: CALL CLEAR
3920  RETURN
4990  REM CHECK FOR A WINNER
5000  NO_LEFT=0
5010  FOR I=1 TO PLAYERS
5020  IF PLAYER_CASH(I)=0 THEN 5050
5030  NO_LEFT=NO_LEFT+1
5040  LAST_PLAYER=I
5050  NEXT I
5060  IF NO_LEFT>0 THEN 5200
5100  PRINT "NO ONE IS LEFT.":"THE GAME
      ENDS IN A TIE."
5110  GOTO 5400
5200  IF NO_LEFT>1 THEN 5400
5300  PRINT PLAYER_NAMES(LAST_PLAYER);"
      WINS!"
5400  RETURN
20000 PLS="PLEASE ANSWER THE QUESTION"
20010 CALL CHAR(96,"FFFFFFFFFFFFFFFF")
20020 CALL CHAR(97,"FFFFFFE7E7FFFFFF")
20030 CALL COLOR(9,2,16)
20050 CALL CLEAR
20090 ROW=12
20100 FOR I=1 TO 9
20110 FOR J=1 TO 9
20120 READ DICE_PIP(I,J)
20130 NEXT J
20140 IF INT(I/2)=I/2 THEN 20170
20150 MSG$="CHUCK-A-LUCK"
20160 GOTO 20180
20170 MSG$="                "
20180 DISPLAY AT(ROW,14):MSG$
20200 NEXT I
20300 CNT=0
20310 FOR I=1 TO 3
20320 FOR J=1 TO 3
20330 FOR K=1 TO 3
20340 CNT=CNT+1
20350 LOC_Y(CNT)=(J+I*4)*8
20370 LOC_X(CNT)=(K+2)*8
20400 NEXT K
20410 NEXT J
20420 NEXT I
21000 CALL CLEAR
21010 INPUT "NEED INSTRUCTIONS (Y/N)? ":
      A$
21020 A$=SEG$(A$,1,1)
21030 IF A$="Y" THEN 21100
21040 IF A$="N" THEN 22000
21050 PRINT PLS
21060 GOTO 21010
21100 PRINT :::::"WELCOME TO THE GAME
      OF":"    CHUCK-A-LUCK!":::
21110 PRINT "THIS GAME CAN BE PLAYED BY"
      :"2 TO 4 PLAYERS. EACH PLAYER STAR
      TS OUT WITH $500. FOR"
21120 PRINT "EVERY TURN, EACH PLAYER BET
      SFROM $1 TO $50 ON A DICE VALUE
      FROM 1 TO 6. THREE"
21130 PRINT "DICE ARE THEN ROLLED. EACH
      PLAYER WILL THEN RECEIVE AN AMOUN
      T EQUAL TO HIS BET"
21140 PRINT "MULTIPLIED BY THE NUMBER OF
      TIMES THE VALUE HE SELECTED CAME
      UP. IF NO DIE HAS THE"
21150 PRINT "VALUE SELECTED, THE PLAYER
      LOSES HIS BET. A PLAYER WHO GOES
      BANKRUPT IS OUT OF THE"
21160 PRINT "GAME. THE GAME IS OVER WHEN
      ONLY 1 PLAYER REMAINS. IF NOONE R
      EMAINS, THERE IS NO"
21170 PRINT "WINNER.":::
21500 FOR I=1 TO 1000
21510 NEXT I
22000 INPUT "HOW MANY PLAYERS (2-4)? ":P
      LAYERS
22010 IF PLAYERS<2 THEN 22060
22020 IF PLAYERS>4 THEN 22060
22030 IF INT(PLAYERS)=PLAYERS THEN 22100
22060 PRINT PLS
22070 GOTO 22000
22100 FOR I=1 TO PLAYERS
22110 PRINT "PLAYER NUMBER";I;"ENTER YOU
      R"
22120 INPUT "NAME-":PLAYER_NAMES(I)
22140 IF PLAYER_NAMES(I)<>"" THEN 22250
22170 PRINT PLS
22180 GOTO 22110
22250 PLAYER_NAMES(I)=SEG$(PLAYER_NAMES(
      I),1,10)
22310 PLAYER_CASH(I)=500
22320 NEXT I
22330 RETURN
25000 DATA 0,0,0,0,1,0,0,0,0
25010 DATA 1,0,0,0,0,0,0,0,1
25020 DATA 1,0,0,0,1,0,0,0,1
25030 DATA 1,0,1,0,0,0,1,0,1
25040 DATA 1,0,1,0,1,0,1,0,1
25050 DATA 1,1,1,0,0,0,1,1,1
25060 DATA 0,0,1,0,0,0,1,0,0
25070 DATA 0,0,1,0,1,0,1,0,0
25080 DATA 1,0,1,1,0,1,1,0,1
25090 DATA X
28000 T2=700 :: T=120
28010 CALL SOUND(T,392,1)
28020 CALL SOUND(T,523,1)
28030 CALL SOUND(T,659,1)
28040 CALL SOUND(T,784,1)
28050 CALL SOUND(T,784,1)
28060 CALL SOUND(T,784,1)
28070 CALL SOUND(T,659,1)
28080 CALL SOUND(T,659,1)
28090 CALL SOUND(T,659,1)
28100 CALL SOUND(T,523,1)
28110 CALL SOUND(T,659,1)
28120 CALL SOUND(T,523,1)
28130 CALL SOUND(T2,392,1)
28140 CALL SOUND(1,39999,30)
28150 CALL SOUND(T2,392,1)
28160 CALL SOUND(T,523,1)
28170 CALL SOUND(T,659,1)
28180 CALL SOUND(T,784,1)
28190 CALL SOUND(T,784,1)
28200 CALL SOUND(T,784,1)
28210 CALL SOUND(T,659,1)
28220 CALL SOUND(T,659,1)
28230 CALL SOUND(T,659,1)
28240 CALL SOUND(T,392,1)
28250 CALL SOUND(T,392,1)
28260 CALL SOUND(T,392,1)
28270 CALL SOUND(T2,523,1)
28280 RETURN
```

SPELLING FLASH

TI BASIC

Spelling Flash will help students review their spelling periodically. This program does not use the Texas Instruments Speech Synthesizer.

Its design incorporates one of the simplest, yet most elemental programming structures: the *loop*. One of the most valuable features of computers is their ability to repeat any task many times over. *Spelling Flash* uses a GOTO statement to form the loop. The program begins, reads a word from its data, presents it to the student, accepts the response, prints a message to the student and then repeats the process. Line 330, GOTO 200, simply sends the program back to line 200, where the process begins again. In this case, the loop (and in *Spelling Flash*, the program) ends when it reads the non-word "ZZZ." Line 210 checks for this *flag*; if the spelling "word" is "ZZZ," it ends the program.

In order to use this program, the spelling words have to be typed into the program as DATA statements. The accompanying listing has a selection of spelling words, starting in line 380, but you can put in words of your choice, of course. If you use more words than are in the listing shown, and in the process generate more DATA statements with more line numbers, you will have to alter the value after THEN in line 210 to reflect the new line number of the END statement.

The words will be read as string variables. They may be entered with separate statements for each word, or several words may be listed in each statement, as long as they are separated by commas. "Words" in this context may, of course, also consist of phrases or names with embedded spaces or other special characters. Such phrases *must* be enclosed in quotes. ZZZ must be the last word in the list of words; if it isn't, the computer will return a data error when it tries to read data that's not there.

When the program runs, the screen is first cleared and a spelling word is flashed on the screen. After a short delay, the word is cleared and the student is asked to type in the spelling word. The subroutine in lines 340-370 cause the delay; if it seems too long or too short, the value in line 340 can be changed. The student signals that he's finished spelling the word by pressing the ENTER key. The program gives some positive reinforcement with some sounds and the message, "YOU SPELLED IT RIGHT!!" If the word is incorrectly spelled, the student must try again until it is correct.

This section of the program is also a loop. An incorrect spelling sends the program from line 280 back to line 220 until the student gives the correct spelling. After the student has spelled the word correctly, the screen is cleared again and the next word flashes on the screen.

```
100 REM      *********************
110 REM      *                   *
120 REM      *SPELLING FLASH*
130 REM      *                   *
140 REM      *********************
150 REM
160 REM
170 REM
180 CALL CLEAR
190 PRINT "SPELLING FLASH":::::
200 READ WORD$
210 IF WORD$="ZZZ" THEN 410
220 PRINT "SPELL ";WORD$
230 GOSUB 340
240 PRINT "SPELLING WORD:"
250 INPUT W$
260 IF WORD$=W$ THEN 290
270 PRINT :::"SORRY-PLEASE TRY AGAIN":::
280 GOTO 220
290 CALL SOUND(500,-1,2)
300 CALL SOUND(500,-2,2)
310 PRINT :::"YOU SPELLED IT RIGHT!!"
320 GOSUB 340
330 GOTO 200
340 FOR DELAY=1 TO 800
350 NEXT DELAY
360 CALL CLEAR
370 RETURN
380 DATA TIGER,BEHAVE,BEGIN,"JOHN ADAM
S"
390 DATA TOMORROW,OZONE,SPRIG,GOOSE
400 DATA CREAM,COVER,AROMA,ABATE,ZZZ
410 END
```

This program may be saved on cassette tape for the students' daily use. Each week, you can alter the list of spelling words by changing the DATA statements.

Pocket Typing Trainer

Pocket Typing Trainer

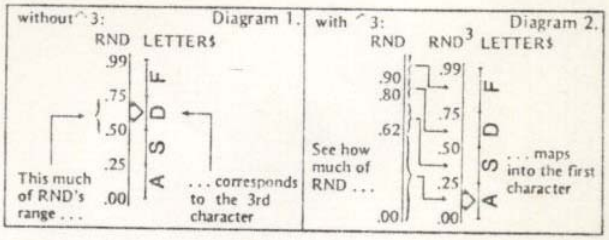
Here is a pocket-sized program for the TI-99/4A—small enough to fit on a 3 × 5 card—that is not only quick to key in, but is also educational, illustrates a powerful technique with random numbers and is fun for all ages. The *Pocket Typing Trainer* asks which characters the user would like to practice, and then plays back an endless series of random five-character groups for him to copy. Two tones rising at the end of the typist's response say "Correct;" two tones descending here mean "Oops." Try it! If you are a beginning typist, start with characters ASDF, the home keys of the left hand. Stop the program with a FCTN 4 (or SHIFT C on the 99/4) keystroke when you can type those four consistently without looking at them, and RUN the program again with ASDFJKL, and so forth . . . . If you are already a typist and you want to practice some of the unusual features of the TI-99/4A keyboard, as well as some of the characters important in BASIC (but not usually part of the typist's repertoire), try the characters "$( )*+-.

You are unlikely to notice it, but the *Pocket Typing Trainer* tends to focus on the characters which the typist is getting wrong—a remarkably sophisticated feature to find in a pocket-sized program—and one which brings me to my next point.

```
100 REM  POCKET TYPING TRAINER
110 REM
120 REM
130 DISPLAY "TYPE IN THE LETTERS YOU W
    ANT TO PRACTICE TODAY"
140 INPUT LETTERS
150 LENGTH=LEN(LETTERS)
160 OUT$=NUL$
170 FOR I=1 TO 5
180 OUT$=OUT$&SEG$(LETTERS,INT(LENGTH*
    RND^3+1),1)
190 NEXT I
200 DISPLAY "":OUT$
210 INPUT "":INS
220 IF OUT$=INS THEN 320
230 FOR I=1 TO 5
240 LS=SEG$(OUT$,I,1)
250 IF LS=SEG$(INS,I,1)THEN 280
260 N=POS(LETTERS,LS,1)
270 LETTERS=LS&SEG$(LETTERS,1,N-1)&SEG
    $(LETTERS,N+1,LENGTH-N)
280 NEXT I
290 CALL SOUND(100,131,3)
300 CALL SOUND(100,110,3)
310 GOTO 160
320 CALL SOUND(100,110,3)
330 CALL SOUND(100,262,2)
340 GOTO 160
```

Skewing the Distribution

Line 180 is where OUT$, the random character string, is manufactured a character at a time. It might have been written without the^3, in which case equal segments of the interval from zero to one would be assigned to the characters given by the typist. (Since the 99/4's built-in random generator, RND, generates "uniform random" numbers, every character would have the same chance of being chosen.) With^3, the random numbers are cubed before a character is chosen. Since the numbers are less than 1, they get smaller as they are cubed; this results in many more RND's corresponding to characters at the left end of the LETTER$ string. For example, suppose that LETTER$, the string of characters which the typist wants to practice, has four characters. If RND turns out to be .50001, then the character a bit more than half way down LETTER$ (i.e., the third character) would be the one chosen. But if we cube RND, the result is .12500, which is well within the first quarter of the range from 0 to 1; and the first character is chosen. Perhaps Diagrams 1 and 2 would help to illustrate this more clearly. The *Pocket Typing Trainer* takes advantage of this by moving missed letters to the beginning of LETTERS$ (Line 270).



The lesson here is that uniform random numbers like those provided by RND are a perfectly satisfactory foundation for any sort of randomness one could desire. This includes the statisticians' favorites: Gaussian, binomial, gamma, and so on. One simply needs to apply the proper transformation$.

Homework

Tailoring and embellishing programs to suit users' personalities is at least half the fun of computing. The *Pocket Typing Trainer* can be extended in many directions. Here are some of the options:

Problem #1 (simple): Modify the program to allow the typist to choose how many random characters he'd like on the line.

Problem #2 (moderate): Change the program to heighten the emphasis on characters which the typist is getting wrong whenever the error rate is high.

Problem #3 (sound and graphics practice): Keep score, and periodically (say every 25 lines) treat the typist to a colorful and melodic display, one whose elaborateness is greatest for a perfect score.

WHAT IS UCSD PASCAL And Why Is Everybody Talking About It?

You can hardly pick up a computer publication, or attend a computer conference or fair these days without being inundated with discussion of UCSD Pascal. To understand what all the fuss is about, you must first know something of what is meant by *portability* and understand the concept of *pseudocode* and its relation to the *pseudomachine*.

Portability, Pseudocode, and P-machine

Let's start by assuming that you already know that Pascal is a structured, high-level, compiled language (just as TI BASIC is a high-level interpreted language). In this article we won't go into the theory of compilers, interpreters, or the structure of Pascal as a computer language; we'll save that for a future time. For right now, let's imagine that your friend has written a really great Pascal compiler and operating system in his native 6502 Assembly Language for his Apple computer. You'd like to move it to your fully-configured TI-99/4 which has a TMS9900 microprocessor. What are your options? Sure, you could always recode the Pascal system for your TMS9900 (assuming you had a TMS9900 assembler), but it would probably be almost as much work as starting from scratch. How about first writing a 6502 simulation program for your TMS9900 and letting it re-write all the 6502 code? But even if you do this, the extra layer in between will result in a loss of speed and a greater memory overhead. This is what the microcomputer community has been up against—virtually *no portability* in moving languages or applications software from one system to another without a major re-working of the code.

Now let's design a hypothetical processor to provide a convenient "home" for Pascal. We'll give it built-in instructions for doing the type of things that the Pascal language likes to do. Let's call this pseudomachine a *p-machine* for short, and configure it to be a simple, idealized stack computer that uses pseudocode, or *p-code*—the native language or machine code for the p-machine.

Great, but where do we go from here? What's the use of a p-machine, and how does it contribute to software portability? Must we throw out all existing hardware and software and start over by giving everyone p-machines? Obviously not. Rather, consider what would happen if we could eliminate the differences between the instruction repertoires of specific microprocessors, so that they all execute an identical p-code. If a p-machine emulator for each CPU were written (in its native assembly language), one of the largest obstacles to portability would be overcome: Software could be written on different computers in a high-level language such as Pascal, then compiled to p-code, and finally "interpreted" for each specific CPU. Since the p-code would be universal, in theory a program written on, say, an Apple could be run without modification on a TI-99/4, if the program consisted entirely of p-code. Score one for portability!

This is, in effect, what has been done in the UCSD Software System. All high-level languages in the system—only one of which is Pascal—are compiled into p-code. One way of looking at it is that the *system software* is not portable at all, because it is always executed on a p-machine. The portability is provided by a p-machine emulator for each host. So when you think of a TMS9900-based system running Pascal, it is really running a simulation of a computer which is running Pascal object programs.

Speed vs. Space: A Tradeoff

What price do we pay for the benefit of portability? The detour through a p-machine often produces slower execution than would native code. But raw execution speed is often overshadowed because p-code is considerably smaller than the corresponding native code—allowing the available memory to store a more capable program. If a program can be represented with p-code that fits entirely into available memory, and using native code requires extensive overlaying, then the p-code version will actually run faster!

For best performance, it is desirable to optimize some portions of a program for space and others for speed. Since the UCSD Pascal System provides communication between an assembly language routine and a Pascal host program, it is possible (with some reduction in portability) to code time-critical routines (usually less then 10% of a program) directly in assembly language. The low-level assembly routine can request access to host program global variables and constants, and can also allocate its own global storage space.

A project is underway at SofTech Microsystems (the firm responsible for the licensing and maintenance of the UCSD Pascal System) to alleviate many of the performance drawbacks of p-code (e.g., speed) without sacrificing port-

ability. Code generators will translate time-critical procedures into native code through an optional step in the compilation process. A code generator will take a complete p-code program as input, and produce, as output, a mixture of unmodified p-code and translated native code procedures. Programs can then be written and maintained entirely in Pascal, with the p-code object version still completely portable. A prototype code generator for the TMS9900 demonstrated that improvement in execution performance compared to interpretive execution has been around a factor of 15! And if we take into account that translated native code for the TMS9900 is about 50% larger then the corresponding p-code, the improvement is indeed significant.

The Operating System

UCSD Pascal is not only a language compiler, but a complete operating system with utilities and libraries. In addition to the Compiler, you have a screen-oriented Editor and a File Manager (or *Filer*). The design philosophy behind UCSD Pascal was to keep users continually informed about the state of the system and the options available in that state. This is done with a prompt line that allows users to select options by typing single-character commands.

The screen orientation of the Editor means that you'll be doing lots of paging instead of scrolling. The editor positions a cursor into the text file being edited and surrounds it with a "window" into that area of the file. When you look at the display screen, you are peering into this window. To modify text, you simply move the cursor to the place where the change is desired and indicate the change. Commands are provided for moving cursor, finding and replacing patterns of text, making insertions and deletions, and copying text from elsewhere and moving it to any position indicated by the cursor. In addition to the powerful text editing commands, special facilities are provided for processing documents—e.g., user-specified left and right margins and auto-indenting to encourage the writing of structured programs. In microcomputer systems without an 80-column display, horizontal scrolling allows users to move the text window left and right to view the entire Pascal page.

When you enter the Filer, you have access to another complete set of commands: (1) *housekeeping commands* such as listing directories, compressing files on a disk, and testing disks for bad sectors; plus (2) *program execution and file manipulation commands* for executing named object programs, invoking (with shortcuts) important system programs, designating files for removal, and renaming or transferring among on-line devices.

The Pascal Compiler translates Pascal programs from a humanly readable text form (source code saved on disk by the Editor) into p-code form (object code) which is saved on disks for future execution. The Compiler is designed to translate the entire contents of a text file in one pass. But unlike the Editor and Filer, it has hardly any interactive commands. You can, however, change certain controls (*directives*) which govern the way in which the Compiler does its work.

Error Handling

A big difference between an interpreted language (such as BASIC) and a compiled language (such as Pascal) is demonstrated in the way syntax and run-time errors ar handled: If the Compiler finds a syntax error, it halts an displays an error message (if you've set it to return to th Editor automatically), or prints on the screen a progres display containing copies of the line (and previous line where the program error was found, as well as the code number of the syntax error. You can fix the error by return ing to the Editor or attempt to compile the rest of the pro gram. In some less drastic conditions, the program will, i fact, compile all the way to the end without the Compile losing its way.

Run-time (execution) errors also cause all the action t stop. A three-line error message tells you the type of error the segment and block where it occurred, and how far i is from the beginning of the block (which you convert t the actual line of code). In simple cases, this will be all th help that's needed to pinpoint the error; in more comple cases, you'll have to insert WRITELN statements (th equivalent of PRINT) to determine the values of variable before the program blew up. (There's no convenien BREAK statement as in TI BASIC.)

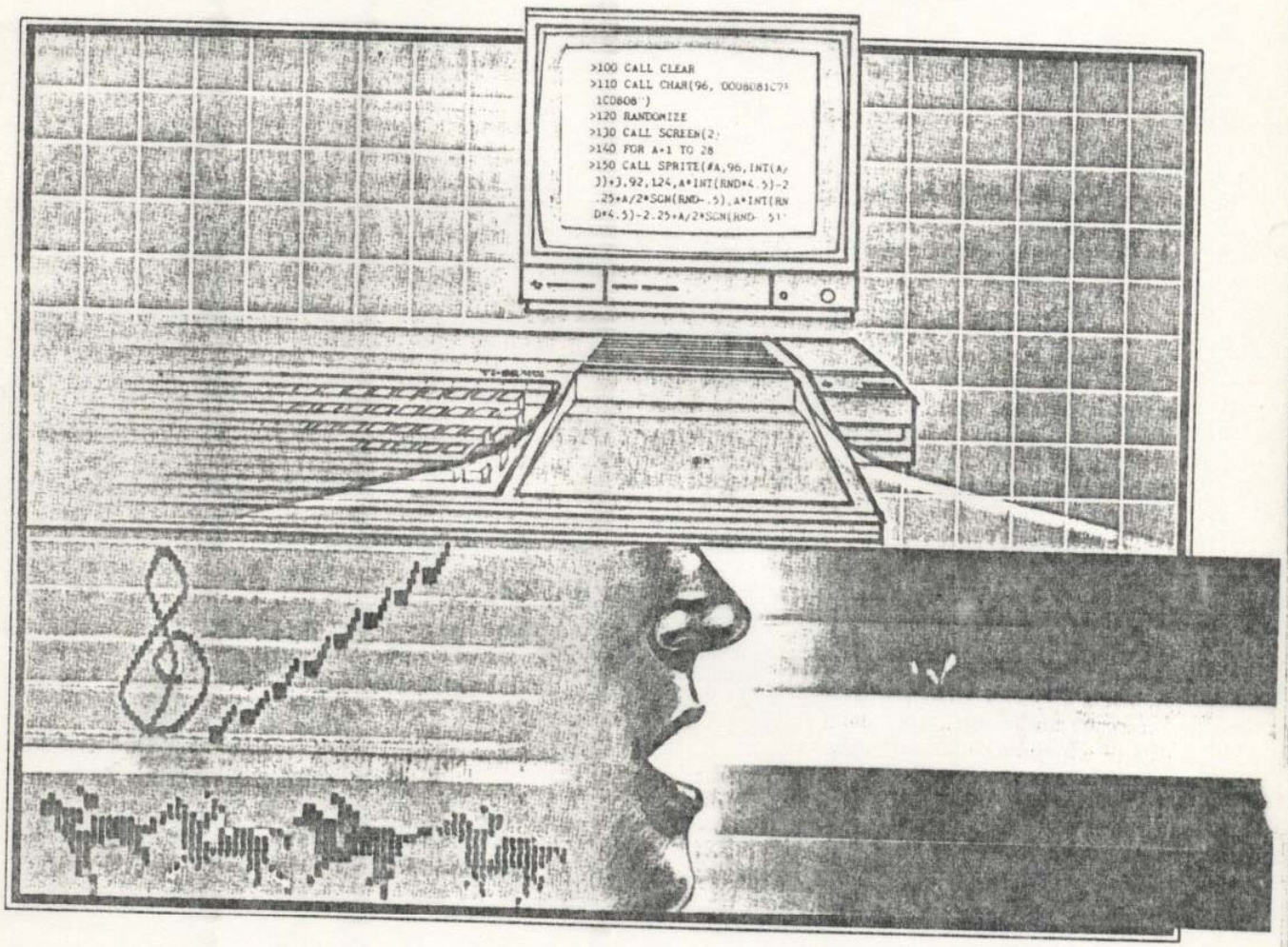
Additional Language Support

The UCSD Pascal System does, in fact, support additional compiled languages. At present, the FORTRAN-77 and BASIC Compiler are supported directly by SofTech Microsystems (MicroFocus CIS COBOL is also presently running under the UCSD p-System). SofTech also has a Cross-Assemblers Package (a complete set of cross-assemblers generating native code for the Z80, 8080, Z8, PDP-11/LSI-11, 6502, 6800, 6809, and 9900 microprocessors) that allows programming on the host machine of your choice, for the object machine of your choice. Think of the possibilities. . . .

UCSD Pascal and the TI-99/4 Community

Texas Instruments has implemenfed UCSD Pascal in a P-Code Card for the TI Peripheral Expansion System. The P-Code Card contains an operating system called the UCSD p-System and allows access to a variety of languages in addition to Pascal. Besides being a powerful tool for software *developers*, UCSD Pascal in TI's version is also of great importance to software *users:* Users won't have to buy all the software and hardware that software developers need in order to write and debug programs. The simplest configuration for software users requires the TI Home Computer, a monitor or TV set, the TI Peripheral Expansion System, the Memory Expansion Card, the P-Code Card and a cassette drive; software developers will need the Disk Memory System (the Disk Drive Controller and up to three disk drives) as well. Under this two-tier system, a TI-99/4A user will be able to run some very sophisticated and powerful *applications* software with only a minimal investment in the *system* hardware and software.

3
Inside BASIC and Extended BASIC



3

Inside BASIC and Extended BASIC

Ready to try it on your own? All it takes is BASIC logic—and a few tricks.

Language Conversion:

TRS-80 BASIC to TI BASIC

Tucked away in my basement, I have both a Radio Shack TRS-80 and a Texas Instruments TI-99/4A. The half-dozen personal computer magazines I read each month provide coding and ideas for many new programs for my TRS-80. I now have a large collection of these programs and have grown to appreciate greatly the help and enjoyment this software library provides. Unfortunately, it just hasn't been that easy to acquire software for the TI machine. [But now, with the birth of *99'er Magazine*, this situation will be rapidly remedied.—Ed.] The solution for me was obvious. I'd convert my TRS-80 programs to TI BASIC.

At the suggestion of *99'er Magazine's* editor, I read an article by Harley M. Templeton appearing in the November 1980 issue of *Personal Computing* magazine. Although the article highlighted the major differences between the versions of BASIC used on the two systems, it didn't point out which differences matter and which are merely interesting but of little practical importance. As you might expect, the only way to find out is actually to convert a program and learn from the problems that you encounter.

To set up a fair test, I selected TRS-80 programs from opposite ends of the spectrum: The first was a "number cruncher" which I had written to convert the number correct on a test to a scaled value on a continuum of learning. (My nine-to-five job involves the management of the standardized testing programs for the Portland, Oregon, School District.) The other program was an adaptation of the ideas behind a slot machine in David Ahl's *Basic Computer Games*—a program with extensive use of graphics.

The first trouble I encountered was in converting the PRINT AT command available on the TRS-80. The procedure suggested by Templeton was to set a loop as follows:

```
400 A$="PRINT THIS STARTING AT 10,2"
500 CALL CLEAR
600 FOR I=1 TO LEN(A$)
700 N1=ASC(SEG$(A$,I,1))
800 CALL HCHAR(10,(I+1),N1)
900 NEXT I
```

In theory this works fine, but it is slow if the string length is long; single characters don't walk across the screen—they crawl! Since the program requires a prompt printed in the middle of the screen to cue the operator to enter the next five values for the scaling procedure, my final solution was to use the following coding:

```
100 PRINT "MESSAGE AT THE MIDDLE OF THE
    SCREEN"
200 PRINT : : : : : : : : :
```

This procedure causes the text prompt to scroll up from the bottom to the middle of the screen. It is not especially speedy, but it is fast enough for the data entry in cases where you don't need lines that disappear at the top of the screen as the result of this scrolling action.

The ease with which the "number crunching" code converted was a pleasant surprise. It was important to keep track of the differences in the line numbers for GOTO's and other branches, but that, in fact, presented little problem. What was more difficult was converting the logic of IF-THEN-ELSE clauses. TRS-80 (Microsoft) BASIC allows multiple statements following the THEN- and ELSE-coding that are difficult to keep straight and re-code. The multiple line conditionals can be converted, but the conversion requires a clear head and a basic understanding of how the program works.

Because I had written the TRS-80 program myself (it had more lines of documentation than coding) and naturally understood its operation, the conversion was fairly straightforward. After I changed nearly all the PRINT and PRINT AT statements, the program worked the first time (surprise). To check it out, I made a comparison run on the TI-99/4 and the TRS-80. Surprisingly, they ran the same job in almost the same time: three minutes for a forty item test. Finally I spruced up the program a little with CLEAR and CALL SCREEN commands to take advantage of the color options available on the TI machine.

The second program was a challenge. It had essentially four main parts: (1) an introductory message, (2) the set-up graphics of the "slot machine," (3) the rotation of the wheels in the slot machine, and (4) the determinaton of the winnings and losses. The first and easiest part of the program to set up was the section which printed the introductory messages. I couldn't resist adding the CALL SCREEN command and sprucing up the comments to make it more attractive (at least to me). In this instance, the lack of speed for the HCHAR command was a benefit since it painted the screen at a leisurely-yet-pleasing pace. Before I was through, I had changed all the code in this section for aesthetic reasons.

My real conversion problems began in the second section. There, I came face-to-face with the significant dif-

ferences in the way graphics are handled by the two systems. In moving from a screen of 16 × 64 to one of 24 × 28, I had to stop and develop a new outline shape for the slot machine—one that would fit the TI screen. Deciding the colors to be used in defining the outline of the machine and the shapes to be matched (cherry, bar, bell, orange, lemon) took extra time. After some experimentation using dark blue against a white background, the lemon became a lime (dark green). To develop a new set of four characters for the orange, I experimented with CALL CHAR until the figure finally looked like a circle instead of one of Dali's exploded watches. Since there isn't an orange color available, the orange became a plum (magenta). I was still a character short, so I used the heart from the back of the user's manual.

En route to coding this part of the program, I had to devise the shapes, assigning them to one of the sixteen character sets. Twice, however, I made the mistake of trying to conserve memory by using one of the character sets with pre-defined codes. This caused errors in the print statements using these codes. The moral of that experience: Whenever possible, stay away from the first eight character sets when defining new characters. It took a while to work the kinks out of this section, but the addition of color made a tremendous diffence, and I became hooked on TI graphics. (I'll probably never turn the TRS-80 version of this program again).

At this point, I realized that virtually every line of the original program had been rewritten in the move to the TI machine. Since this was to be an article on program conversion, not programming, I called the editor at *99'er Magazine* to make sure I hadn't missed the point of the article. Gary, however, wasn't surprised at all, and encouraged me to include suggestions on rewriting as well as conversion.

The third section of this program was probably the toughest to convert. I have been responsible for programming and systems analysis for over ten years on a variety of large computer systems. This has required establishing structured programming standards for every program with which I work. Even though I had personally keyed in the slot machine program, I had forgotten how poorly it was documented. This is not a criticism of Ahl's book, but rather a realistic comment on what you are likely to encounter

An Example of Code Translation From TRS-80 BASIC to TI BASIC

```
120 FOR I1=1 TO NI
130 IF IZ(I1)<>0 THEN PRINT "THIS ITEM
DROPPED";ID$: GOTO 160
140 IF K$=K1$ THEN IF C1(I1)=0 GOTO 160
ELSE C2=10*C1(I1)+200
150 DX=C3-C2
160 NEXT I1
Translates to:
120 FOR I1=1 TO NI
130 IF IZ(I1)=0 THEN 140
132 PRINT "THIS ITEM DROPPED";ID$
134 GOTO 160
140 IF K$<>K1$ THEN 150.
142 IF C1(I1)=0 THEN 160
144 C2=10*C1(I1)+200
150 DX=C3-C2
160 NEXT I1
```

SUMMARY OF COMMANDS

TRS-80 Commands Not Requiring Conversion

ABS	GOSUB	REM
ASC	INPUT	RESTORE
ATN	INT	RETURN
CHR$	LEN	SGN
COS	LET	SIN
DATA	LOG	SQR
DIM	ON/GOSUB	STR$
END	ON/GOTO	TAN
EXP	PRINT	VAL
GOTO	READ	

TRS-80 Commands Easily Converted

TRS-80	TI BASIC
CLS	CALL CLEAR
FIX	INT
INKEY$	CALL KEY
INPUT#-1	INPUT#1
LEFT$(A$,N)	SEG$(A$,1,N)
MID$(A$,N1,N2)	SEG$(A$,N1,N2)
RANDOM	RANDOMIZE
RIGHT$(A$,N)	M=LEN(A$)-N+1 SEG$(A$,M,N)
RND(N)	INT(N*RND+1)
STOP	BREAK
TAB	TAB (with comma)
?	PRINT
'	REM

TRS-80 Commands That Can Be Ignored

CLEAR	CSNG	DEFSNG
CDBL	DEFDBL	DEFSTR
CINT	DEFINT	FRE

Commands Difficult to Convert to TI BASIC

TRS-80	TI BASIC
IF . . THEN . . ELSE	IF . . THEN . . ELSE* refer to line numbers
POINT	CALL CHAR . . CALL GCHAR
POKE (graphics)	CALL CHAR . . CALL HCHAR
PRINT AT	FOR . . ASC . . CALL HCHAR . . NEXT PRINT . . FOR . . PRINT . . NEXT
RESET	CALL CHAR . . CALL HCHAR
SET	CALL CHAR . . CALL HCHAR

* Improved capability with multi-line statements in Extended BASIC

Commands Not Available In TI BASIC**

ERL	PEEK	STRING$
ERR	POKE	USR
ERROR	POS	VARPTR
ON ERROR	RESUME	PRINT USING

** Most of the equivalent commands are available in TI Extended BASIC

when converting a program. After an hour of tracing through a maze of GOSUBs without the benefit of a single comment, I decided on a total rewrite.

The TRS-80 version had the program determine the coordinates of one of the nine open spots on the slot machine and then perform a PRINT AT at the location. Using FOR-NEXT loops, I was able to overprint the nine spots to give the illusion of a rotating machine wheel. By converting the PRINT AT commands to HCHAR calls and storing the four codes for each shape in an array, I simulated this action on the TI-99/4. The graphics were fantastic (an unbiased estimate), but the speed was disappointing. In the TRS-80 version it was necessary to insert dummy FOR-NEXT loops to slow down the rotation of the wheels; the TI version, on the other hand, was too slow right from the start.

The single enhancement I had made to the TRS-80 version was to have the wheels stop one at a time, to prevent giving away the final result of the pull during rotation. To keep the wheels moving at a constant speed on the TI-99/4, I included dummy counting loops as each wheel was stopped. In spite of its lack of speed, the richness of the TI-99/4 graphics made the TI BASIC program a more appealing simulation of real slot machine action than the TRS-90 version.

To summarize, if the program you want to convert is a number cruncher with a few graphics, the conversion should go smoothly and result in a TI BASIC program which runs with speed roughly comparable to its TRS-80 cousin. But if the program involves the heavy use of graphics, expect to rewrite it. And if the program is poorly documented to boot, keep a bottle of aspirin handy. Futhermore, because of the limitations of the TI BASIC IF-THEN-ELSE, and the lack of a PRINT AT command you can expect nearly every converted program to increase in length. On the plus side, however, the extended variable names available in TI BASIC make it possible to enhance the quality of the documentation and structure of the rewritten program.

One final note: TI's Extended BASIC Command Cartridge adds the PRINT AT and PRINT USING statements, has the capability of controlling up to 28 moving objects simultaneously, has improved IF-THEN-ELSE capability, and supports true subroutine definition (a significant aid in structuring programs). Although Extended BASIC probably won't alter the need for rewriting graphic programs, it should make the job a lot easier.

Language Conversion:

APPLESOFT to TI BASIC

The Apple II has also generated its fair share of applications and games programs—most of them taking advantage of the Apple's color graphics capability. In this regard the Apple is more like the TI-99/4A than the non-color TRS-80.

The APPLESOFT language card has about 29 non-graphic commands which are identical to TI BASIC. These commands, shown in Table 1 below, can be copied without much concern over compatibility.

ABS	DEF	GOTO	ON...GOSUB	SOP
ASC	DIM	INT	READ	STEP
ANT	END	LEN	REM	STOP
CHR$	EXP	LET	RETURN	STR$
COS	FOR...TO	LOG	SGN	TAN
DATA	GOSUB	ON...GOTO	SIN	

Table 1

In the remaining 26 or so commands, the differences range from very slight to major. Most importantly, the differences, though slight in format or content, can cause major problems in converting code. I'll go into each command, showing what to look for and how to resolve difficulties.

String Commands

APPLESOFT uses three different commands (LEFT$, MID$, and RIGHT$) in place of the TI's SEG$. The statement LEFT$(A$,N) references the first N characters of string A$. This directly translates into SEG$(A$,1,N). MID$(A$,M,N) is the same as SEG$(A$,M,N). Right$(A$,N) references the last N characters in string A$. The best way to duplicate this is to combine the LEN and SEG commands as follows: SEG$(A$,LEN(A$)-N+1,N).

The VAL function acts the same way in both APPLESOFT and TI BASIC if the field being VALed is a valid numeric string. That is, both will return 45.2 as the value of "45.2". If the string does not contain valid numeric characters, however, the results are very different. TI BASIC will stop the program if the field contains non-numeric characters. APPLESOFT, however, will return with the numeric equivalent of the numbers found in the string before the first non-numeric character. For example: VAL ("123AB") will return with 123. If the first character of the string isn't numeric, APPLESOFT returns a 0.

This is important because it means that APPLESOFT does not have to edit a string prior to the VAL statement. A typical program will have code such as:

```
10 INPUT A$
20 X=VAL(A$)
30 IF X=0 THEN 10
```

I've found that in most cases, I can ignore the whole issue by using TI's built-in numeric editor and coding INPUT X in place of statements 10 to 30 above. If you can't do this, use the following routine to replace the APPLESOFT VAL command:

```
10 FOR Y=1 TO LEN (A$)
20 IF (ASC(SEG$(A$,Y,1))<48)
   +(ASC(SEG$(A$,Y,1))>57)THEN 40
30 NEXT Y
40 IF Y=1 THEN 80
50 Y=Y-1
60 Y=VAL(SEG$(A$,1,Y))
70 GOTO 90
80 Y=0
90 END
```

Note: This is *not* a rigorous equivalent of APPLESOFT's VAL, but it is sufficient for whole numbers greater than −1.

FOR-TO-STEP-NEXT

In the usual run of programs, the FOR-TO-STEP statement is identical in the two interpreters. There is, however, a very significant difference to look out for. The BASIC statement FOR Z=5 TO 4 will execute once in APPLESOFT but will not execute at all in TI BASIC! This difference is important but can easily be spotted while transcribing a program. It isn't so obvious if the statement is FOR Z=A TO B where A and B are computed variables. The safest thing is to test for A greater than B. If it is, make B equal to A before entering the loop.

Both interpreters treat the STEP statement the same way and are very similar in the format of the NEXT statement—though in APPLESOFT, NEXT may be used by itself to end a single FOR loop. If the FOR loops are nested, however, APPLESOFT needs the control-variable name following NEXT, as does TI BASIC.

INPUT/OUTPUT (I/O)

Both machines use very similar INPUT and PRINT statements. They differ only in the use of print separators. Both use the comma as a tab command and the semicolon as a non-space separator. APPLESOFT reserves the colon

for a special use and doesn't treat it as a new line separator. When converting, always keep this in mind because it provides a powerful formating tool when converting PRINT statements. The TAB command is similar in both interpreters, but TI machine skips to a new line if a TAB value is less than the current column location. The APPLE will ignore the TAB statement in this case.

As part of the print function, APPLESOFT has a command of the format SPC(N), which is used to print N spaces. This must be replaced with a string of N spaces in the TI PRINT statement. APPLESOFT has to be very careful with spaces because it does not format a number with leading and trailing spaces the way TI BASIC does. This means that it is very rare to see something like PRINT J;K in APPLESOFT—a perfectly acceptable command in TI code since all numbers are printed with a trailing space.

The APPLE II screen starts off with the cursor at the top and works its way down to the bottom before scrolling begins. The APPLE uses HTAB and VTAB statements to shift the print position horizontally and vertically in order to print information at different locations on the screen. TI BASIC uses the colon, instead, to force line feeds. When converting, either change the print format to use line-feeds (colons), or use HCHAR to print at an equivalent location. Note: TI provides a full PRINT AT (using HCHAR) routine as part of its *Programming Aids I* package, but it is very slow. In many cases (where scrolling is acceptable), you are better off setting up a sequence of PRINT commands using the colon (PRINT : : : : : :). If you must use the HCHAR method of print out, here's a routine to print string A$ at row RO, column CO:

```
10 FOR X=1 TO LEN(A$)
20 CALL HCHAR
(RO,CO+X-1,ASC(SEG$(A$,X,1)))
30 NEXT X
```

This routine is much faster but requires you to remember to begin at column 3 (where TI BASIC begins its PRINT line) and not to allow A$ to extend past column 30 (where TI ends its PRINT line).

The prompt for APPLESOFT input is the same as for TI BASIC except that it uses a semicolon in place of the colon to separate the prompt from the input variable. For example:

```
10 "ENTER A NUMBER";Q
          VS
10 "ENTER A NUMBER":Q
```

The last I/O difference concerns getting a single character without using the INPUT statement: APPLE uses the GET statement, while TI uses the CALL KEY statement.

SCREEN COMMANDS

The APPLE has three modes of processing: Text mode and two different graphics modes. While in Text mode, the programmer has a number of commands which provide a wide range of control over the screen. The APPLE screen, in this mode, acts like the TI—except it starts at the top and works its way down to the bottom before scrolling. It also allows the programmer to set the width of the print screen ("text window") and the length (number of lines) of the text window, among other things. Some of the most commonly encountered commands are:

CALL −936	Clears the screen inside the test window
CALL −912	Scrolls the text window up 1 line
CALL −868	Clears the current line from the cursor to the right
HOME	Same as TI's CALL CLEAR
POKE 33,L	Sets left margin of window to L
POKE 33,W	Sets width of window
POKE 34,T	Sets top of window
POKE 35,B	Sets bottom of screen
FLASH	Starts 'flashing' output from white letters on black to black letters on white and back again
INVERSE	Reverses output to black letters on white
NORMAL	Resets FLASH and INVERSE
POS(N)	Gets current horizontal column of the cursor (i.e., N will have column number 0-39)

To simulate FLASH or INVERSE, use TI BASIC's CALL COLOR statement. For Example, CALL COLOR (3,16,2) gives white numbers from 0 to 7 on a black background. Changing this to CALL COLOR (3,2,16) will cause the inverse of it to appear (black numbers on a white background).

RANDOM NUMBERS

Because APPLESOFT has the ability to retain a random number for re-use, you cannot always convert the APPLE RND statement directly to TI. In APPLESOFT, if the statement is RND(0), APPLESOFT re-uses its last random number. If the statement is RND(N) where N is positive, it gives a new random number. If the statement is RND(N) where N is a negative number, N acts as a 'seed' number, and all other RND statements will follow a standard sequence. Note that the value N can be any positive number in order to give a new random number.

If you see a statement using RND(0), backtrack to the last statement with RND(N) and save that random number in place of RND(0). For example:

```
10 IF RND(2)<.5 THEN 500
:
60 IF RND(0)<.75 THEN 600
```

in APPLESOFT would convert in TI BASIC to:

```
10 Q=RND
15 IF Q<.5 THEN 500
:
60 IF Q<.75 THEN 600
```

MULTISTATEMENT LINES

A key point about APPLESOFT that I haven't yet mentioned is that it allows multiple statements on one program line. Each statement is separated by a colon. This allows code like:

```
10 X=X+Y:Y=Y+1:Z=Z+1
```

Translating multistatement lines can be a big problem because there may not be available line numbers to assign to the converted statement lines. For example:

```
400 A= A+1:FOR I=1 TO X:B=I*A:NEXT I
401 GOSUB 403
402 RETURN
403 REM
404 GOSUB 600
405 A=A+10
406 RETURN
```

The problem here is that there is no room to separate the multiple statements on line 400.

You can get around this by using a line number translation: Multiplying all line numbers by 10 allows you space to insert the extra line of code. The translated code is as follows:

```
4000 A = A + 1
4002 FOR I = 1 TO X
4004 B = I*A
4008 NEXT I
4010 GOSUB 4030
4020 RETURN
4030 REM
4040 GOSUB 6000
4050 A = A + 10
4060 RETURN
```

IF-THEN-ELSE

APPLESOFT does not require the ELSE feature of an IF statement because it allows other statements after the THEN part of the IF statement, as in the following:

```
10 IF A = X THEN X = X + 1:Y = Y + 1
20 A = X + Y
```

If X is equal to A, all statements following THEN are executed. If X isn't equal to A, the program simply advances to statement 20. The TI BASIC equivalent is:

```
10 IF X = A THEN 15 ELSE 20
15 X = X + 1
16 Y = Y + 1
20 A = X + Y
```

Because TI BASIC lacks multiple statements per line, it requires much more coding and a concurrent increase in memory needed for code. Keep this in mind if you are tempted to enter a program requiring 16K RAM in APPLESOFT; it probably won't fit in your TI machine. [Of course, if you have TI Extended BASIC, all this is moot, since this Command Cartridge allows multiple statement lines. See "HOW E-X-T-E-N-D-E-D IS EXTENDED BASIC?"—Ed.]

LOGICAL EXPRESSIONS

Both interpreters allow logical expressions to be used as if they were numeric values. APPLESOFT treats true expressions as if they are equal to 1, while false expressions are equal to 0. For TI BASIC true expressions are −1, false are 0. Whenever converting code from APPLESOFT, just insert a "−" in front of the logical expression:

```
10 X = (0$ = "A")*5
```

becomes

```
10 X = -(0$ = "A")*5
```

AND/OR

APPLESOFT allows multiple IF tests to be combined using the Boolean operators AND and OR. TI BASIC also allows this using the "*" and "+" arithmetic operators, respectively. For example:

```
10 IF (A = B) AND (C = D) THEN X = X + 1
```

is replaced with

```
10 IF (A = B)*(C = D) THEN 15 ELSE. . .
15 X = X + 1
```

In some cases, a straight conversion of the APPLESOFT IF-THEN will result in wasteful code. It is always a good idea to understand the purpose of the tests being made, and if possible, re-code them more efficiently. For example:

```
10 IF (A = B) AND (C = D) THEN X = X + 1
20 Y = Y + 1
```

would convert to:

```
10 IF (A = B)*(C = D) THEN 15 ELSE 20
15 X = X + 1
20 Y = Y + 1
```

but it would take less code (and therefore less core!) to invert the test:

```
10 IF (A<>B)+(C<>D) THEN 20
15 X = X + 1
20 Y = Y + 1
```

SPECIAL FUNCTIONS

Each interpreter has special functions oriented toward the manufacturer's hardware. Some of these are similar to other functions available in a different computer. I will list only the ones most commonly seen in APPLESOFT programs.

CLEAR	Initializes all variables. Automatically done by TI BASIC as part of RUN.
HIMEM LOMEM	Sets highest and lowest memory available to BASIC. No equivalent in TI BASIC.
FRE(0)	Gets amount of available memory left.
PDL(N)	GETS joystick input. In TI BASIC, use CALL JOYST instead. The PDL function returns with values from 0 to 255. If the value of N is 0 to 3, you are referencing the joysticks, but values from 4 to 255 can do weird things. Luckily, the APPLE joysticks don't seem to be used much. Also, the only way to test for the 'FIRE' buttons is to PEEK(−16287) through PEEK(−16284) for paddles 0 thru 3.
POP	Cancels the last GOSUB. This is mostly used in edit subroutines where an error causes the progam to go to an error routine instead of RETURNing. The only way to code an equivalent in TI BASIC is to have the edit routine coded in an error switch which is interrogated as soon as the subroutine RETURNs.
ON ERR RESUME	This tells APPLESOFT to GOTO a part of the program if it encounters certain errors while processing. In TI BASIC, any errors are either handled by the BASIC interpreter (e.g., dividing by zero), or cause the program to end (e.g., reading past the last DATA statement). The ON ERR is most often used to trap an error expected by, or consciously caused by the programmer.
USR(X)	Jump to a machine language subroutine.

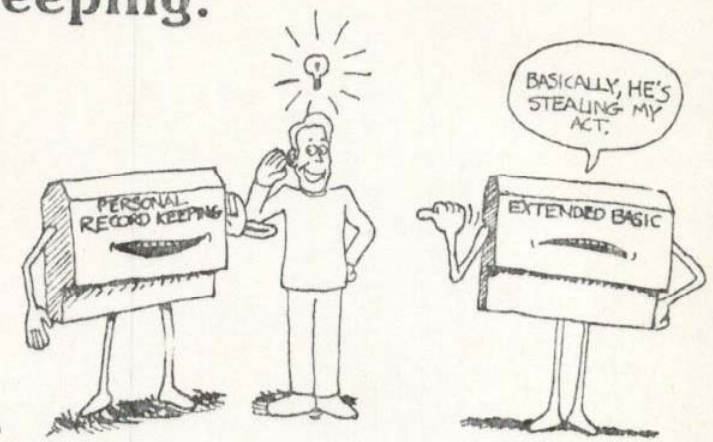
As you can see from the foregoing, converting most code from APPLESOFT to TI BASIC is straightforward, with most of the effort devoted to converting PRINT statements. Most importantly, don't get frustrated if your first attempts don't succeed the way you intended. After a while, it will all become second nature.

The Secret of Personal Record Keeping:

Implementing

DISPLAY AT and ACCEPT AT

Without Extended BASIC



Some of you may have accidentally stumbled upon features of the TI-99/4 that are not described anywhere but which are nonetheless quite helpful. I did. . .and what happily resulted was a way to quickly print text to and accept it from anywhere on the screen without having to pass through loops or causing the screen to scroll.

Those of you with Extended BASIC already have this capability with the DISPLAY AT and ACCEPT AT statements. Now you can have these powerful features in TI BASIC (the language built into the TI-99/4 and 99/4A computers), provided the *Personal Record Keeping* Command Cartridge is inserted. This cartridge, which is quite powerful and versatile in itself, will interface with the console's BASIC routines and allow you to use two new statements: CALL D and CALL A. [See "Personal Record Keeping: Managing a Mobile Home Park" for more information on the *PRK* cartridge. Those of you without the *PRK* cartridge but who happen to have the *Statistics* cartridge should be able to use that instead.—Ed.]

Before getting into the documentation, I should, of course, mention that you can also print anywhere on the screen without CALL D by handling the printing character by character using the subroutine given in the examples in your manuals, i.e., "Character Definition." The drawbacks of that method include lack of speed (the letters appear one by one), more cumbersome programming and more memory space taken up.

1. DISPLAY AT – numerical data

CALL D (R, C, L, V)

R = row number of first character of print line
C = column number of first character of print line
L = maximum length of print line; must be >=1
V = variable for the value that is to be printed

R/C— The R(ow) and C(olumn) variables are meaningful with values between 1 and 24, and 1 and 28, respectively (the print field 24×28 is used). Values below the minimum of 1 (0 and negative numbers) are treated as the value 1. Values above the maximum (24 or 28) are automatically subtracted as many times as is required to bring the result between 1 and 24 or 28; this result is then used as the R and C value. This is a nice feature that eliminates many program halts of "BAD VALUE" that often result from careless programming. Data at the end of the screen line is not printed at the beginning of the next screen row as is the case with the CALL HCHAR statement.

L— The L position can be used with a fixed number (the maximum meaningful number is 28) or as a variable to which the function can be assigned in numerical form, like SEG$ in strings.

V— Instead of a numerical variable, you can also put a number in this position; it will then be printed on the screen in a position according to the rules above.

Example 1

```
100 CALL CLEAR
110 V=326525
120 CALL D(12, 10, 5, V)
130 GOTO 130
```

Of course you can explain why this program displays only 3265 in the middle of the screen. (Remember that a sign—equivalent to a digit—precedes each number, and that plus signs are suppressed on printing.) How would you have to change line 120 to give the full 326525?

2. DISPLAY AT - string data

Version 1: CALL D(R, C, L, S$)
Version 2: CALL D(R, C, L, ("PAUL W. KARIS")
Version 3: CALL D(R, C, L, CHR$(N))

The variables R, C, and L work as described previously under section 1, above.
Here expecially, L can be put to good use as a built-in SEG$.

ersion 1: the string variables S$ is printed
ersion 2: the string between quotes is printed
ersion 3: a complicated way of saying CALL HCHAR(R,
, N) that is merely mentioned here as illustration of the
ossiblities

xample 2

```
100 CALL CLEAR
110 A$="THIS IS MID-SCREEN"
120 CALL D(12, 4, 19, A$)
130 GOTO 130
```

ACCEPT AT – numerical data

The ACCEPT AT statement works like INPUT but can
: formated anywhere on the screen. The input prompt can
: printed in the appropriate place with the technique of
ction 2, above. The built-in value checks are an additional
ature.

ALL A(R, C, L, F, A, MN, MX)

, C, and L have been explained in section 1.

= function variable
= accept variable
IN = minimum value
IX = maximum value

F— The numerical variable in this position assumes a value 1-7 depending on certain function keys being depressed. The values connected to these functions in this way should not be confused with the ASCII values of these functions that can be useful in CALL KEY statements. For completeness, I'll also tabulate the ASCII values here.

Function Key		CALL A value (F position)	ASCII value
TI-99/4A	TI-99/4		
FCTN 5	SHIFT W – BEGIN	6	14
FCTN 8	SHIFT R – REDO	4	6
FCTN 7	SHIFT A – AID	3	1
FCTN 9	SHIFT Z – BACK	7	15
FCTN 4	SHIFT C – CLEAR	2	2
FCTN 6	SHIFT V – PROC'D	5	12
	ENTER	1	13

CLEAR will not only give F a value of 2, but it also clears the input printing field on the screen and is to be used when typed input is not yet entered and should be changed. Warning: This means that if you write a program that continually loops to a CALL A statement, CLEAR cannot be used to break the program. Only QUIT or cutting the power will work then, but it will also erase your program in the process! The solution to this problem is to program your escape routine, e.g., IF F=3 THEN 10000 enabling you to use AID to bring the program to line 10000 which reads: 10000 END.

A— The variable in the position of A assumes (accepts) the value you typed in much in the same way as the input variable does after you depress ENTER. The F variable, of course, then gets the value 1 since you have used the function key ENTER. If you press ENTER when the print/input field contains no information (only "space"), F will take on the value in the above table if one of the function keys has previously been pushed.

MN— The numbers or the values of the numerical
MX— variables in the positions MN and MX respectively determine the minimum and maximum values that A will accept. A gentle beep when you press the ENTER warns you if you try to step beyond these imposed limits. The screen, of course, will accept any numerical data, provided that the length does not exceed L(e.g., if L=2 and MX=10000 you still cannot get A to become more than 99 since the screen will not accept more than 2 digits). Since the plus and minus signs (+ and –) as well as the letter E (scientific notation) are all considered to be numerical input, they will also be accepted. String data, however, are not accepted by the screen at all when you use CALL A in this way.

If MN=MX, A will accept only the MN and the MX value. If MN>MX, A shouldn't accept any value at all, but illogically, it does accept the MN value.

Example 3

```
100 CALL CLEAR
110 CALL D(3, 3, 28, "ENTER 1, 2, OR 3")
120 CALL A(10, 25, 1, F, B, 2, 3,)
130 CALL CLEAR
140 FOR T=1 TO 500
150 NEXT T
160 CALL D(15, 3, 28, "YOUR CHOICE WAS")
170 CALL D(15, 20, 2, B)
180 FOR T=1 TO 500
190 NEXT T
200 GOTO 100
```

4. ACCEPT AT – string data

CALL A(R, C, L, F, A$)

R, C, and L are explained in section 1.
F is explained in section 3.
A$ = accept string variable.

A$ The variable in the A$ position is filled with the typed string information when you press ENTER.

Example 4

```
100 CALL CLEAR
110 M$ ="PLEASE ENTER YOUR NAME"
120 CALL D(5, 3, 26, M$)
130 CALL A(10, 3, 20, F, N$)
140 CALL CLEAR
150 FOR T=1 TO 500
160 NEXT T
170 CALL D(5, 2, 28, "THANKS " & N$)
180 FOR T=1 TO 500
190 NEXT T
200 GOTO 100
```

Now you're on your own: It's your turn to apply these two new commands and, perhaps, discover some additional ones.

[Note: In the event that Texas Instruments gets away from producing "hybrid" Command Cartridges (containing both BASIC and GPL coding), future releases of *Personal Record Keeping* will not offer the capabilities described in this article.—Ed.]

Dynamic Screen Graphics

MANIPULATION OF CHARACTER

Would you appreciate being able to write shorter programs that effectively do the same thing as longer ones? Or, would you enjoy watching the computer do a large amount of the tedious and boring designing, defining and selecting of dozens of graphics characters—work that you would otherwise have to do yourself? If your answer to both of these questions is YES, read on, fellow 99'er.

The scheme used in the TI-99/4A to represent screen character patterns with hexadecimal numbers is compact and convenient—ingenious really. It's compact because only 16 digits uniquely specify the on-off states of the 64 pixels in each 8 × 8 pixel character block. Such a system is certainly more satisfactory than display systems that provide only a small selection of predefined characters. It's convenient because the programming requires only simple statements of the form:

```
CALL CHAR(IJK,"0123456789ABCDEF")
```

to define any 8 × 8 character imaginable. Likewise the statement:

```
CALL HCHAR(ROW,COLUMN,IJK,REPEAT)
```

will put character IJK anywhere on the screen. After a brief period, one is able to work intuitively, giving little conscious thought to the format.

Yet even with this system, there remains a considerable amount of tedious work to be done because every character we want on the screen (beyond the resident alphabet, etc.) must be defined and must be located. Doing this for many characters can mean lots of work, as in Figure 1, where a graphic occupying less than half the screen contains 33 different characters. All 64 user-definable characters would use up 64 lines of code just to define; if resident characters were redefined, we could end up having in memory a hundred or so program lines devoted to this one purpose.

In addition, there is the wear and tear on the programmer. He gets his ears burned if he leaves out one of those quote marks. Additional possibilities for errors include leaving out a comma or parenthesis or, worse, having a pattern identifier string with more or less than 16 numbers, or inadvertently typing in a nonhexadecimal symbol. Just type in four or five dozen CALL CHAR(IJK,"0123456789 ABCDEF") statements and you will surely develop an acute case of boredom. Such static definition—with a program line for every new character and the resulting long list of CALL CHAR statements—is a lot of trouble and a source of errors.

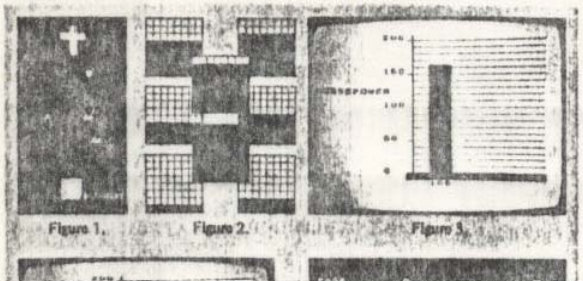
It is also unnecessary. A little experimenting will show that we can define screen characters with data statements and a loop. Only a single CALL CHAR statement need be

yped in and carried in memory. Such a method was used
i the program which draws Figure 1. The program is given
i Listing 1, *Xmas-Tree*. The hexadecimal strings which
efine the screen characters to be used are in data statements
tarting at line 270. The loop starting at line 440 reads a
ata statement and puts the hexadecimal string it has pick-
d up into a CALL CHAR statement. Thus the definition
sent off to graphic memory where it can be used later
i the program as many times as needed. In this program,
ach data entry contains a comment to help one figure out
hat is happening on the screen, and each data entry con-
ins three items: identification string, character number,
nd pattern-identifier string. On the next pass through the
op, another hexadecimal string is picked up and put in
he CALL CHAR statement. Thus another defined screen
haracter is sent off to memory.

After the program has cycled the last time through the
oop, all the screen characters described in the data
tatements are in memory. They are now available using
ALL HCHAR or CALL VCHAR statements just as if
he program had run through dozens of CALL CHAR lines.
ewer program lines have been used, the possibility of er-
ors reduced, and life has been made much easier for the
rogrammer.

In a similar manner, characters are located on the screen
eginning at line 740. For this application the data entries
ave the form: identification string, row number, column
umber, character number. The identification string serves
nly as documentation. The loop at line 940 puts this in-
ormation in a CALL HCHAR statement which then sends
off to the video display processor. All characters will now
ppear on the screen at their assigned locations. Of course,
he information we have in data statements could also be
tored on a floppy disk.

Dynamically defining characters and putting them on the
reen with data statements and loops (1) saves program lines
nd effort, (2) reduces errors, and (3) can make a program
asier to follow if documentation is added. Although for
is program no special attempt has been made to reduce
he memory required, the information in data statments
ould be packed tighter by omitting identification. Also, we
ould incorporate the number of repetitions in the data
atements.



ure 1. Many different characters can mean lots of work for the programmer.
re 2. Screen characters used for one-pixel resolution in bar height.
3. Bar graph with one-pixel resolution.
4. Three variables plotted with one-pixel resolution.
5. An example of 99/4 graphics.

Another opportunity for making character definition and placement a part of program dynamics occurs in plotting bar graphs. Bar graphs are a frequent application for computer graphics, and they look terrific on the color monitor.

On the TI-99/4A it is easy to plot a bar (Y characters high) by just using CALL VCHAR(ROW,COLUMN,IJK,Y). But the resolution will be very poor because we can adjust the bar height in increments of only one full character, which is about 3/8 of an inch on the 13-inch monitor. Ideally we'd have a continuously adjustable bar height, but this infinite resolution cannot be realized with raster-scan systems. We can, however, get resolution equal to the pixel height. Toward this end we will define eight screen characters as shown in Figure 2. The first character has the bottom row of pixels turned on, the next one has the bottom two rows turned on, etc. The eighth character has all pixels turned on.

These characters are then used as bar tops. Stick the right one on top of your bar graph and you have resolution of one pixel (which is 1/8 of a character)—quite satisfactory with existing CRT's. On the 13-inch monitor this height increment is about 3/64 of an inch.

The program in Listing 2, *Bar-Topper*, which uses this method, plots the bar graph in Figure 3. The characters available for use as bar tops are defined beginning at line 360. Scale of 1 character = 10 units is applied to the value entered at the keyboard starting at line 700. The integral value of Y is found and the remainder used to select the bar top character needed. The actual selection is done by the ON GOTO statement at line 780.

This program does work, but represents a brute force approach. If there is only one bar on the graph, then only one character will be used at the bar top. Yet eight bar-top characters have been defined and are sitting in memory. To take an extreme case, suppose we have four variables to be represented by four bars of different colors. Here, 32 characters must be defined and available for use as bar tops, yet only four bar-top characters will actually be used. Besides taking up memory, we have used half of the user-defined characters. This approach is wasteful. Why define characters that sit in memory but are never used?

Let's try a better idea by devising a program that defines bar-top characters after reading the data. Then it can define only characters that are needed. In other words, the data determine what bar-top characters are defined. To do this, we will have in the program a master string containing fourteen zeros and sixteen F's. Segments exactly sixteen spaces long can be taken from this master string with a SEG$ statement. Next, the segment can be used as the pattern-identifier string and put in a CALL CHAR statement to define a bar top. Where will these 16-space segments start? Well, the data can cause a character with the first row of pixels turned on to be defined, or a character with the second row turned on, etc.

A possible coding to do this might be as follows:

```
110 MASTER$ = "00000000000000FFFFFFFFFFFFFFFF"
115 REMAINDER = BARHEIGHT - INT(BARHEIGHT)
120 TOPPATTERN = INT(REMAINDER*8 + .5) + 1
130 STARTPOSITION = 2*TOPPATTERN - 1
140 TOPPATTERN$ = SEG$(MASTER$,STARTPOSITION,16)
150 CALL CHAR(97,TOPPATTERN$)
160 CALL HCHAR(21 - Y,16,97,3)
```

Here the 21 in 21 – Y allows the bar to be up to 20 rows high.

Suppose, for example, that data calls for a bar top with the bottom two rows turned on. Then TOPPATTERN will

be 2. Then STARTPOSITION = 3. Then the pattern-identifier string created in line 140 will be

```
TOPPATTERN$="000000000000FFFF"
```

(as you can see, if you will take the trouble to count this off, starting at the third position in the master string). The resulting screen character that is defined in line 150 will be one with the bottom two rows of pixels turned on. As the program runs, we want each datum to determine where the 16-space segment will begin. Thus we have used the remainder to calculate STARTPOSITION. By notching back and forth with STARTPOSITION, the routine will define any character needed to top off a bar.

With this particular routine there will be a little problem associated with rounding up to the next higher grid line on the next higher row. For instance, if the scale used is 1 character = 10 units, we would want 99.9 to appear on the graph as 100. Another problem (I didn't say this was too simple) involves the character to be used for the body of the bar. This character must have all pixels turned on, but the routine above will not create such a character for all values of the data set.

Auto-Top, a program in which these problems are solved, is given in Listing 3. A routine similar to the one above starts on line 750. Character 96, which is used for the body of the bar, is defined earlier in the program. Note that this master string contains 18 F's. (If you try this program, you had better count them carefully.) TOPPATTERN = 9 will pick up the extra F's at the 17th and 18th positions.

The problem of rounding up to the next higher grid line (so 99.9 will show up as 100 as in the earlier example) is taken care of in lines 820 and 830 where a one-row-on character is defined and put on the very top of the bar if, and only if, TOPPATTERN = 9.

A graph with only one bar is not very useful. We can generate additional bars with a loop. The routine in Listing 4, *Three-Bars*, plots three bars of different colors. See line 680. (My 13-inch monitor displays a lot of spillover with most colors—especially with red. There is less spillover with light or medium green or blue, and with white and yellow.) As the loops runs, it will shift to succeeding color sets with the expression 89+BAR*8 as can be deduced by considering the statement

```
CALL CHAR(89+BAR*8,TOPPATTERN$).
```

When BAR=1, this statement defines character 97; when BAR=2, character 105; and when BAR=3, character 113. The first character is in color set 9, the second in color set 10, and the third in color set 11, allowing for three bars of different colors.

The position of the bars is shifted by the expression. 11+5=16 is the position of the left edge of the first bar, and the left edges of all bars are 5 columns apart. These bars are three columns wide. Figure 4 shows this graph as photographed on the 13-inch monitor.

This program and the earlier ones here might be a little longer than if they were written in the standard way. However, they will not get much longer if the graphics are made more elaborate. For example, the bar graph program does not get much longer if more bars are added.

The bar graph in Figure 5 was made using these techniques. I present it here just to show off the kind of goodlooking graphics that can be made with the TI-99/4A and TI BASIC. This program—with its outlining and the fact that it reads and writes data for eight variables from files and calculates items such as percentages—is more involved than the listing given here.

This brings up a new problem that has been created: In many of my programs I run out of characters. I did not notice this limitation when I was typing in so many CALL CHAR, CALL HCHAR, and CALL VCHAR statements. Actually when you think about it, there are not very many characters available. If you start at the left of the screen and put a different character in each space, you will run out of characters in the fifth line if you include punctuation, number, the alphabet, and the eight user-definable sets.

In other words, it takes only about 17% of the screen to display all available characters. Mathematically, we are not about to run out of characters since there are 256 different ways to put together just one row of a character. And the number of characters that can be on the screen in this graphic mode is 24 rows of 32 columns = 768 spaces.

Since my interest is primarily in graphics, available user-definable characters are more important to me than memory. Memory problems can often be avoided. To put a unique character on every space on the screen would require 48 character sets—several times more than any home computer presently has. I do not know if this is unreasonable. Two years ago the idea of a 48K memory sounded unreasonable. Perhaps some computer architect will devise a method of going to a higher resolution with nested character sets. [For a discussion of the high-resolution bit-mapped graphics supported by the TI-99/4A, see "3-D Animation with the TMS9918A Video Chip."—Ed.]

Finally, note that for some applications it can be useful to define random graphics characters. This process, however, really eats up character sets. In Listing 5, *Twinkle*, random characters are defined that also have a certain amount of shape. Line 240 of this code generates random numbers from 1 to 16, and lines 480 to 620 convert them to hexadecimal notation 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F. These numbers are assembled into a 16-space string. This hexadecimal string then goes into a CALL CHAR statement to define a random graphic characher.

Shape is forced on the character in lines 280 to 470 by rejecting certain numbers generated by the random number generator. In this particular application, the edges of the characters are "rounded off" so they will not appear square.

I use such random-patterned screen characters to soften up the edges of my "block graphics" designs. ("Blockhead graphics?") Another application is to create dramatic effects as is done in *Twinkle* given in Listing 5.

I also use random characters to induce variations on things that, as in nature, change with time-shadows or explosions, for instance. Some video games could undoubtedly profit from this technique. I get a little tired of aliens that always blow up the same way. Hmm—come to think of it, there is that video game with the pigeon in it. . . .

Listing 1

```
100 REM  *********************
110 REM  *    XMAS-TREE      *
120 REM  *********************
130
140
150 REM  ABOUT 7568 BYTES
160 REM  PRESS ANY KEY TO END SCREEN
    DISPLAY.
170 CALL SCREEN(2)
180 CALL COLOR(9,7,1)
190 CALL COLOR(10,14,1)
200 CALL COLOR(11,14,4)
210 CALL COLOR(12,12,4)
220 CALL COLOR(13,2,4)
230 CALL COLOR(14,7,4)
240 CALL COLOR(15,15,1)
250 CALL COLOR(16,5,16)
260 REM  PATTERN-IDENTIFIERS
270 REM  FORMAT: IDENTIFICATION, CHARA
    CTER NUMBER, HEXADECIMAL STRING...
280 REM  EXAMPLE: TREE TRUNK,142,4E53
    B635C659487A, TREE BODY,143,000000
    0000000000...
290 DATA LEFTSIDE OUT,96,0107070F0F0F1
    F7F,RIGHTSIDE OUT,97,80E0E0F9F0F0F
    8FE,BORDERTOP,98,7F7F3F0707010100
300 DATA TREE BOTTOM,99,FFF8F8F0F0E0C0
    00,8,100,FF3F0F0707030100
310 DATA BORDER TOP,102,FFFFFFFFFF0000
    00,BORDER BOTTOM,103,000000FFFFFFFF
    FFF
320 DATA PLUM,104,10FEFEFEFE7C3810,BEL
    L,107,10103838387C7C7CFE
330 DATA PLUM,112,10FEFEFEFE7C3810,DIA
    MOND,113,10103838387CFE7C3810,BELL,115
    ,10103838387C7C7CFE
340 DATA PLUM,120,10FEFEFEFE7C3810,DIA
    ,121,10103838387CFE7C3810,BELL,123,101
    038383877C7CFEFE
350 DATA PLUM,128,10FEFEFEFE7C3810,DIA
    MOND,129,10103838387CFE7C3810,BELL,131
    ,10103838387C7C7CFE
360 DATA LEFT INSIDE,136,FEF8F0F0E0E0C0
    00,RIGHT INSIDE,137,7F1F0F0F0707070
    300,CT,138,FFE7E7830101010101
370 DATA BOTTOM IN,139,0107070F0F0F3FF
    F,8,140,80C0E0E0E0F0FCFF
380 DATA TRUNK,142,4E53B635C659487A,TR
    EE BODY,143,00000000000000000,POT L
    ,144,3F3F3F3F3F3F3F3F
390 DATA POT LEFT BOTTOM,146,3F3F3F3F3F
    F3F0F0F,POT R,147,FCFCFCFCFCFCFCFC
400 DATA POT R B,149,FCFCFCFCFCFCF0F0,
    P B,150,FFFFFFFFFFFFFFFF
410 DATA TOP,152,C3C33C18183CC3C3,STAR
     RADIAL,153,FFFFFF0000FFFFFF,STAR
    RADIAL,154,E7E7E7E7E7E7E7E7
420 REM  DEFINE-LOOP
430 RESTORE 290
440 FOR CODE=96 TO 154
450 READ IDENTIFICATIONS,CHARACTERNUMB
    ER,HEX$
460 IF CHARACTERNUMBER>CODE THEN 480
470 GOTO 490
480 CODE=CHARACTERNUMBER
490 CALL CHAR(CODE,HEX$)
500 NEXT CODE
510 REM  START SCREEN DISPLAY
520 CALL CLEAR
530 REM  ----TREE BODY----
540 CALL HCHAR(24,1,143,32)
550 CALL HCHAR(19,6,143)
560 CALL HCHAR(18,2,143,11)
570 CALL HCHAR(17,3,143,9)
580 CALL HCHAR(16,3,143,8)
590 CALL HCHAR(15,4,143,7)
600 CALL HCHAR(14,4,143,7)
610 CALL HCHAR(13,4,143,6)
620 CALL HCHAR(12,5,143,5)
630 CALL HCHAR(11,6,143,4)
640 CALL HCHAR(10,6,143,3)
650 CALL HCHAR(9,6,143,3)
660 CALL HCHAR(8,7,143)
670 REM  ----TREE TRUNK----
680 CALL VCHAR(20,7,142,2)
690 REM  ----PLANT POT----
700 CALL VCHAR(22,6,144,2)
710 CALL VCHAR(22,7,150,3)
720 CALL VCHAR(22,8,147,2)
730 REM  SCREEN LOCATION DATA
740 REM  FORMAT:IDENTIFICATIONS,ROW,CO
    LUMN,CHARACTERNUMBER...
750 DATA POT BASE LEFT SIDE,24,6,146,P
    OT BASE RIGHT SIDE,24,8,149
760 REM  ----FOLLAGE----
770 DATA LO,18,1,96,LI,18,2,136,LO,17,
    1,96,LI,17,2,136,LO,16,1,96,LI,16,
    2,136,LO,15,2,96,LI,15,3,136
780 DATA LO,14,2,96,LI,14,3,136,LO,13,
    2,96,LI,13,3,136,LO,12,3,96,LI,12,
    4,136,LO,11,4,96,LI,11,5,136
790 DATA LO,10,4,96,LI,10,5,136,LO,9,4
    ,96,LI,9,5,136
800 DATA LO,5,6,96,CT,5,7,138,LO,6,6,9
    6,LI,6,7,136,LO,7,5,96,LI,7,6,136,
    LO,8,5,96,LI,8,6,136
810 DATA RO,5,8,97,RO,6,8,97,RI,6,7,13
    7,RO,7,8,97,RI,7,7,137,RO,8,9,97,R
    I,8,8,137,RO,9,10,97
820 DATA RI,9,9,137,RO,10,10,97,RI,10,
    9,137,RO,11,11,97,RI,11,10,137,RO,
    12,11,97,RI,12,10,137
830 DATA RO,13,11,97,RI,13,10,137,RO,1
    4,12,97,RI,14,11,137,RO,15,12,97,R
    I,15,11,137,LO,16,12,97
840 DATA LI,16,11,137,LO,17,13,97,LI,1
    7,12,137,LO,18,13,97,B,18,12,137,B
    ,19,12,99,B1,19,11,139
850 DATA B1,19,10,140,BO,19,9,100
860 DATA BOTTOM,20,5,96,B,20,6,99,B,19
    ,7,139,B,19,8,99,B,19,4,100,B,19,5
    ,140,B,19,2,96,B,19,3,99
870 DATA B IN,18,7,139,B OUT,18,8,136
880 REM  ----CROSS----
890 DATA TOP,2,7,152,L RADIAL,2,6,153,
    R RADIAL,2,8,153,T RADIAL,1,7,154,
    B RADIAL,3,7,154,B RAD,4,7,154
900 REM  ----ORNAMENTS----
910 DATA OUTSIDE BELL,7,9,104,PLUM,20,
    2,107,PLUM,14,4,112,DIAMOND,13,9,1
    13,BELL,16,10,115
920 DATA PLUM,13,4,112,PLUM,12,9,128,D
    IAMOND,14,4,129,BELL,17,3,131
930 REM  SCREEN LOCATION LOOP
940 HOWMANY=86
950 RESTORE 750
960 FOR CHARACTER=1 TO HOWMANY
970 READ IDENTIFICATIONS,ROW,COLUMN,CH
    ARACTERNUMBER
980 CALL HCHAR(ROW,COLUMN,CHARACTERNUM
    BER)
990 NEXT CHARACTER
1000 CALL KEY(0,K,S)
1010 IF S=0 THEN 1000
1020 END
```

Listing 2

```
100 REM  *****************
110 REM  *  BAR-TOPPER  *
120 REM  *****************
130 REM
140 REM
150 REM
160 REM  PRESS ANY KEY TO STOP DISPLAY
    .
170 VERT!CALMAX=200
```

```
180 SCALE=VERTICALMAX/20
190 CALL CLEAR
200 LABEL$="ENTER HORSEPOWER"
210 ROW=12
220 COLUMN=15
230 GOSUB 1000
240 LABEL$="0 TO 209"
250 ROW=13
260 COLUMN=19
270 GOSUB 1000
280 INPUT "                    ":HORSEPOWER
290 CALL SCREEN(8)
300 CALL COLOR(9,13,8)
310 CALL COLOR(10,2,5)
320 REM  DEFINE CHARACTERS
330 REM  FORMAT:IDENTIFICATIONS,CHARA
    CTERNUMBER,HEXADECIMALS...
340 REM  ---GRID---
350 DATA GRID LINE,91,00000000000000FF
    ,VERTICAL AXIS,92,0101010101010101
    ,TIC MARK,93,0101010101010017F
360 REM  --DEFINE BAR TOPS--
370 DATA BOTTOM ROW OF PIXELSON,96,000
    0000000000FF,SECOND ROW ON,97,000
    000000000FFFF,THIRD ROW ON
380 DATA 98,0000000000FFFFFF,FOURTH,99
    ,00000000FFFFFFFF,FIFTH,100,000000
    FFFFFFFFFF,SIXTH,101,0000FFFFFFFFFF
    FFF
390 DATA SEVENTH,102,00FFFFFFFFFFFFFF,
    EIGHTH,103,FFFFFFFFFFFFFFFF
400 REM  ---BASELINE---
410 DATA BASE,104,FF0000FF00000000FF
420 REM  DEFINE LOOP
430 RESTORE 350
440 FOR CODE=91 TO 104
450 READ IDENTIFICATIONS,CHARACTERNUMB
    ER,HEX$
460 IF CHARACTERNUMBER>CODE THEN 480
470 GOTO 490
480 CODE=CHARACTERNUMBER
490 CALL CHAR(CODE,HEX$)
500 NEXT CODE
510 REM  START SCREEN DISPLAY
520 REM  ---GRAPH GRID---
530 CALL HCHAR(22,13,104,18)
540 FOR ROW=21 TO 1 STEP -1
550 CALL HCHAR(ROW,14,91,17)
560 NEXT ROW
570 LABEL$="HORSEPOWER"
580 ROW=9
590 COLUMN=1
600 GOSUB 1000
610 CALL VCHAR(1,13,92,21)
620 FOR ROW=21 TO 1 STEP -5
630 ROWNUMBER=200-(10*(ROW-1))
640 LABEL$=STR$(ROWNUMBER)
650 COLUMN=10
660 GOSUB 1000
670 CALL HCHAR(ROW,13,93)
680 NEXT ROW
690 REM  CALCULATE BAR HEIGHT
700 BARHEIGHT=HORSEPOWER/SCALE
710 Y=INT(BARHEIGHT)
720 REMAINDER=BARHEIGHT-INT(BARHEIGHT)
730 CALL VCHAR(22-Y,16,103,Y)
740 CALL VCHAR(22-Y,17,103,Y)
750 CALL VCHAR(22-Y,18,103,Y)
760 REM  SELECT BAR TOP
770 TOPPATTERN=INT((REMAINDER*8)+.5)
780 ON TOPPATTERN+1 GOTO 790,810,830,8
    50,870,890,910,930,950
790 CALL HCHAR(21-Y,16,96,3)
800 GOTO 970
810 CALL HCHAR(21-Y,16,97,3)
820 GOTO 970
830 CALL HCHAR(21-Y,16,98,3)
840 GOTO 970
850 CALL HCHAR(21-Y,16,99,3)
860 GOTO 970
870 CALL HCHAR(21-Y,16,100,3)
880 GOTO 970
890 CALL HCHAR(21-Y,16,101,3)
900 GOTO 970
910 CALL HCHAR(21-Y,16,102,3)
920 GOTO 970
930 CALL HCHAR(21-Y,16,103,3)
940 GOTO 970
950 CALL HCHAR(21-Y,16,103,3)
960 CALL HCHAR(20-Y,16,96,3)
970 CALL KEY(0,K,S)
980 IF S=0 THEN 970
990 END
1000 FOR POSITION=1 TO LEN(LABEL$)
1010 LETTER$=SEG$(LABEL$,POSITION,1)
1020 CODE=ASC(LETTER$)
1030 CALL HCHAR(ROW,COLUMN-1+POSITION,C
     ODE)
1040 NEXT POSITION
1050 RETURN
```

Listing 3

```
100 REM  *********
110 REM  * AUTO-TOP *
120 REM  *********
130 REM
140 REM
150 REM  ABOUT 5288 BYTES
160 REM  PRESS ANY KEY TO STOP DISPLAY
170 VERTICALMAX=200
180 SCALE=VERTICALMAX/20
190 CALL CLEAR
200 LABEL$="ENTER HORSEPOWER"
210 ROW=12
220 COLUMN=15
230 GOSUB 870
240 LABEL$="0 TO 209"
250 ROW=13
260 COLUMN=19
270 GOSUB 870
280 INPUT "                    ":HORSEPOWER
290 CALL SCREEN(8)
300 CALL COLOR(9,13,8)
310 CALL COLOR(10,2,5)
320 REM  DEFINE CHARACTERS
330 REM  FORMAT:IDENTIFICATIONS,CHARAC
    TERNUMBER,PATTERNS...
340 DATA GRID LINE,91,00000000000000FF
    ,VERTICAL AXIS,92,0101010101010101
    ,TIC MARK,93,0101010101010017F
350 DATA BAR,96,FFFFFFFFFFFFFFFF,BASEL
    INE,104,FF0000FF00000000FF
360 DATA RESERVED FOR TITLE BOX
370 DATA RESERVED FOR LABELS
380 DATA RESERVED FOR LEGEND
390 DATA RESERVED FOR ADDITIONAL CHARA
    CTERS
400 REM  DEFINE-LOOP
410 RESTORE 340
420 FOR CODE=91 TO 104
430 READ IDENTIFICATIONS,CHARACTERNUMB
    ER,PATTERNS
440 IF CHARACTERNUMBER>CODE THEN 460
450 GOTO 470
460 CODE=CHARACTERNUMBER
470 CALL CHAR(CODE,PATTERNS)
480 NEXT CODE
490 REM  START SCREEN DISPLAY
500 REM  ---GRAPH GRID---
510 CALL HCHAR(22,13,104,18)
520 FOR ROW=21 TO 1 STEP -1
530 CALL HCHAR(ROW,14,91,17)
540 NEXT ROW
550 LABEL$="HORSEPOWER"
560 ROW=9
570 COLUMN=1
580 GOSUB 870
```

```
590 CALL VCHAR(1,13,92,21)
600 FOR ROW=21 TO 1 STEP -5
610 ROWNUMBER=200-(10*(ROW-1))
620 LABEL$=STR$(ROWNUMBER)
630 COLUMN=10
640 GOSUB 870
650 CALL HCHAR(ROW,13,93)
660 NEXT ROW
670 REM  CALCULATE BAR HEIGHT
680 BARHEIGHT=HORSEPOWER/SCALE
690 Y=INT(BARHEIGHT)
700 REMAINDER=BARHEIGHT-INT(BARHEIGHT)
710 CALL VCHAR(22-Y,16,96,Y)
720 CALL VCHAR(22-Y,17,96,Y)
730 CALL VCHAR(22-Y,18,96,Y)
740 REM  SELECT BAR TOP
750 TOPPATTERN=1+INT((REMAINDER*8)+.5)
760 MASTER$="0000000000000000FFFFFFFFFFFF
    FFFFFFFF"
770 STARTPOSITION=2*TOPPATTERN-1
780 TOPPATTERN$=SEG$(MASTER$,STARTPOSI
    TION,16)
790 CALL CHAR(97,TOPPATTERN$)
800 CALL HCHAR(21-Y,16,97,3)
810 IF TOPPATTERN<>9 THEN 840
820 CALL CHAR(98,"0000000000000000FF")
830 CALL HCHAR(20-Y,16,98,3)
840 CALL KEY(0,K,S)
850 IF S=0 THEN 840
860 END
870 FOR POSITION=1 TO LEN(LABEL$)
880 LETTER$=SEG$(LABEL$,POSITION,1)
890 CODE=ASC(LETTER$)
900 CALL HCHAR(ROW,COLUMN-1+POSITION,C
    ODE)
910 NEXT POSITION
920 RETURN
```

Listing 4

```
100 REM  *********************
110 REM  *  THREE-BARS  *
120 REM  *********************
130 REM
140 REM
150 REM  ABOUT 5160 BYTES
160 REM  PRESS ANY KEY TO STOP DISPLAY
170 VERTICALMAX=200
180 SCALE=VERTICALMAX/20
190 OPTION BASE 1
200 DIM Y(3)
210 Y(1)=133
220 Y(2)=159
230 Y(3)=99.9
240 CALL SCREEN(8)
250 CALL COLOR(9,5,8)
260 CALL COLOR(10,3,8)
270 CALL COLOR(11,16,8)
280 CALL COLOR(12,2,5)
290 REM  DEFINE CHARACTERS
300 REM  FORMAT: IDENTIFICATION$,CHARA
    CTERNUMBER,PATTERN$...
310 DATA GRID LINE,91,0000000000000000FF
    ,VERTICAL AXIS,92,0101010101010101
    ,TIC MARK,93,01010101010101017F,BAR1
    ,96
320 DATA FFFFFFFFFFFFFFFF,BAR2,104,FFF
    FFFFFFFFFFFF,BAR3,112,FFFFFFFFFFF
    FFFFF,BASELINE,120,FF0000FF000000F
    F
330 DATA RESERVED FOR TITLE BOX
340 DATA RESERVED FOR LABELS
350 DATA RESERVED FOR LEGENDS
360 DATA RESERVED FOR ADDITIONAL CHARA
    CTERS
370 REM  DEFINE-LOOP
380 RESTORE 310
390 FOR CODE=91 TO 120
400 READ IDENTIFICATION$,CHARACTERNUMB
    ER,PATTERN$
410 IF CHARACTERNUMBER>CODE THEN 430
420 GOTO 440
430 CODE=CHARACTERNUMBER
440 CALL CHAR(CODE,PATTERN$)
450 NEXT CODE
460 REM  START SCREEN DISPLAY
470 CALL CLEAR
480 PRINT TAB(13);Y(1);TAB(18);Y(2);Y(
    3)
490 REM  ---GRAPH GRID---
500 CALL HCHAR(22,13,120,18)
510 FOR ROW=21 TO 1 STEP -1
520 CALL HCHAR(ROW,14,91,17)
530 NEXT ROW
540 LABEL$="HORSEPOWER"
550 ROW=9
560 COLUMN=1
570 GOSUB 870
580 CALL VCHAR(1,13,92,21)
590 FOR ROW=21 TO 1 STEP -5
600 ROWNUMBER=200-(10*(ROW-1))
610 LABEL$=STR$(ROWNUMBER)
620 COLUMN=10
630 GOSUB 870
640 CALL HCHAR(ROW,13,93)
650 NEXT ROW
660 REM  CALCULATE & PLOT BARS
670 MASTER$="0000000000000000FFFFFFFFFFFF
    FFFFFFFF"
680 FOR BAR=1 TO 3
690 BARHEIGHT=Y(BAR)/SCALE
700 YY=INT(BARHEIGHT)
710 REMAINDER=BARHEIGHT-INT(BARHEIGHT)
720 CALL VCHAR(22-YY,11+BAR*5,88+BAR*8
    ,YY)
730 CALL VCHAR(22-YY,12+BAR*5,88+BAR*8
    ,YY)
740 CALL VCHAR(22-YY,13+BAR*5,88+BAR*8
    ,YY)
750 TOPPATTERN=1+INT((REMAINDER*8)+.5)
760 STARTPOSITION=2*TOPPATTERN-1
770 TOPPATTERN$=SEG$(MASTER$,STARTPOSI
    TION,16)
780 CALL CHAR(89+BAR*8,TOPPATTERN$)
790 CALL HCHAR(21-YY,11+BAR*5,89+BAR*8
    ,3)
800 IF TOPPATTERN<>9 THEN 830
810 CALL CHAR(90+BAR*8,"0000000000000000
    FF")
820 CALL HCHAR(20-YY,11+BAR*5,90+BAR*8
    ,3)
830 NEXT BAR
840 CALL KEY(0,K,S)
850 IF S=0 THEN 840
860 END
870 FOR POSITION=1 TO LEN(LABEL$)
880 LETTER$=SEG$(LABEL$,POSITION,1)
890 CODE=ASC(LETTER$)
900 CALL HCHAR(ROW,COLUMN-1+POSITION,C
    ODE)
910 NEXT POSITION
920 RETURN
```

Listing 5

```
100 REM  ************
110 REM  *  TWINKLE  *
120 REM  ************
130 REM
140 REM
150 REM  ABOUT 4504 BYTES
160 REM  HOLD DOWN ANY KEY TO STOP DIS
    PLAY
170 CALL SCREEN(8)
180 CALL COLOR(9,10,8)
190 REM  SHAPED RANDOM CHARACTERS
200 RANDOMIZE
210 FOR J=1 TO 4
220 FOR I=0 TO 15
```

```
230 REM  GENERATE RANDOM NUMBERS BETWE
    EN 0 AND 15
240 N=INT((15-0+1)*RND)+0
250 REM  PUT ON CONSTRAINTS TO ELIMINA
    TE CORNERS
260 ON I+1 GOTO 280,300,340,360,410,43
    0,490,490,490,490,410,430,340,360,
    280,300
270 REM  TOP & BOTTOM ROWS
280 IF N>1 THEN 240
290 GOTO 490
300 IF N=0 THEN 490
310 IF N=8 THEN 490
320 GOTO 240
330 REM  2ND & 7TH ROWS
340 IF N>3 THEN 240
350 GOTO 490
360 IF N>12 THEN 240
370 FOURTEST=N/4-INT(N/4)
380 IF FOURTEST=0 THEN 490
390 GOTO 240
400 REM  3RD & 6TH ROWS
410 IF N>7 THEN 240
420 GOTO 490
430 IF N=15 THEN 240
440 EVENTEST=N/2-INT(N/2)
450 IF EVENTEST=0 THEN 490
460 GOTO 240
470 REM  4TH & 5TH ROWS NO CONSTRAINTS
480 REM  FOR N>9 MUST CONVERT TO HEX N
    OTATION; NOTE IN HEX NOTATION A=10
    ,B=11,C=12 ETC.
490 IF N>9 THEN 500 ELSE 630
500 ON N-9 GOTO 510,530,550,570,590,61
    0
510 G$="A"
520 GOTO 640
530 G$="B"
540 GOTO 640
550 G$="C"
560 GOTO 640
570 G$="D"
580 GOTO 640
590 G$="E"
600 GOTO 640
610 G$="F"
620 GOTO 640
630 G$=STR$(N)
640 HEX$=HEX$&G$
650 NEXT I
660 CALL CHAR(95+J,HEX$)
670 GOTO 680
680 HEX$=""
690 NEXT J
700 REM  DISPLAY TITLE
710 DATA 57,57,39,69,82,32,77,65,71,65
    ,90,73,78,69
720 CALL CLEAR
730 REM  ...BORDER...
740 FOR COL=6 TO 26
750 N=INT((4-1+1)*RND)+1
760 CALL HCHAR(14,32-COL,95+N)
770 CALL HCHAR(10,COL,95+N)
780 NEXT COL
790 FOR ROW=10 TO 14
800 N=INT(4*RND+1)
810 CALL VCHAR(ROW,6,95+N)
820 CALL VCHAR(24-ROW,26,95+N)
830 NEXT ROW
840 REM  ...TITLE...
850 CALL HCHAR(12,10,32,14)
860 RESTORE 710
870 FOR I=1 TO 14
880 READ LETTER
890 COLUMN=9+I
900 CALL HCHAR(12,COLUMN,96)
910 CALL HCHAR(12,COLUMN,LETTER)
920 NEXT I
930 REM  ...TWINKLE...
940 C=0
950 COLUMN=INT((26-6+1)*RND)+6
960 N=INT((4-1+1)*RND)+1
970 CALL HCHAR(10,COLUMN,95+N)
980 CALL HCHAR(14,32-COLUMN,95+N)
990 C=C+1
1000 IF C>45 THEN 850
1010 CALL KEY(0,K,S)
1020 IF K>31 THEN 1040
1030 GOTO 950
1040 END
```

How to Write A BASIC Program That Writes BASIC Programs

PART 1:

A SURPRISING DISCOVERY WITH TI'S *PROGRAMMING AIDS III*

TI's *Programming Aids III* opens the door to some powerful programming techniques. The Cross Reference and Editor capabilities of this software will be appreciated by the serious Extended BASIC programmer. But the excitement really begins when you realize how this software does its thing.

PA III can provide (1) a tabular, line-number cross reference for all variables, arrays, keywords, functions, and line-number references in a program and (2) the ability to delete, move, or resequence specified groups of lines within a program much more quickly than could be done manually at the keyboard.

Required Hardware

Programming Aids III is a set of four Extended BASIC programs (LINPUT, CREF, CREFPRINT, and EDITOR) available on disk at a suggested retail price of $19.95. In addition to a disk controller, disk drive, and the Extended BASIC Command Cartridge, a printer is a practical necessity; either the TI Thermal Printer or an RS232-compatible printer may be used. In fact, there is no provision for screen display of the output from the Cross Reference procedure. (I use the inexpensive "Paper-and-Pencil Printer," however, and so modified the CREFPRINT program to display the cross reference table on the screen, using the crude SHIFT CONTINUE method to stop and start the output. These simple changes are given at the end of this chapter.)

EDITOR

The EDITOR program makes possible virtually any desired modification of line numbers in a BASIC or Extended BASIC Program. Heretofore, the only way to resequence a program was to use the RESEQUENCE (RES) command, which affects all line numbers within a program. By contrast, EDITOR allows one to resequence specified sections of a program without affecting others.

If, for instance, you have numbered subroutine statements in a manner which is easy to remember (1000, 2000, 3000, etc), you can retain this numbering and "open up" a previous part of the program for insertion of additional lines. An even more useful application would be the rearrangement of sections of BASIC code. Suppose, for example, you want to merge several programs, each of which contains subroutines. Without EDITOR, you would be faced with the time-consuming chore of moving all subroutines to the end of the merged program. With EDITOR, this procedure can be completed very simply and quickly by renumbering all subroutine lines.

Finally, the EDITOR program allows deletion of sections of BASIC code. If you want to get a subroutine out of one program to use in another, it's no problem.

How EDITOR Works

If you are wondering how a BASIC program can alter another BASIC program, be assured that it's not done with mirrors. It is a relatively simple procedure which anyone with Extended BASIC can use to write all custom utility programs and even BASIC programs which write other BASIC programs!

The technique is based upon what happens to a program when it is saved with the MERGE option (see pp.122-3 of the TI Extended BASIC manual). If you have ever cataloged a disk containing a file saved with the MERGE option, you may have noticed that, unlike an ordinary program which carries the Type description PROGRAM, a program saved with MERGE is actually a data file consisting of display code with variable length records having a maximum length of 163 bytes. A BASIC program can access this sequential file like any other file.

In addition to creating a data file form, saving a program with MERGE makes two other important changes. First, the order of program lines corresponds to the order of program line numbers. (By contrast, when a program is saved without MERGE, the file is a program memory image, and lines are placed in program memory in the order in which they were entered—not according to line number.) Second, the content of each line is represented in condensed format: All non-essential information is deleted in a coding process. When a program saved with the MERGE option is loaded into memory with the MERGE Command and LISTed (see *TI Extended BASIC* Manual, page 114), the coding process is reversed and each program is reconstructed.

In order to understand how the EDITOR program works, it is necessary to know how line numbers are represented in condensed format. The first two bytes of each record contain the line number represented in ASCII code. Table 1 shows how the line numbers "80" and "9020" are represented in ASCII characters. Starting with the line

number 80, the first step involves representing the base 10 number in binary. Two bytes (8 bits each) are available for this representation. Next, the base 10 representation of each byte is determined and the corresponding ASCII symbol produced. In this case, the character with an ASCII code of 80 is "P". Applying this process to the number 9020 gives the ASCII representation "#<".

Table 1
ASCII Coding of Line Numbers

Line Number	80	
	Byte 1	Byte 2
Binary	00000000	01010000
Base 10	0	80
ASCII		P
Line Number	**9020**	
	Byte 1	Byte 2
Binary	00100011	00111100
Base 10	35	60
ASCII	#	<

Table 2
Sample Cross Reference Output

```
MUSIC 2/1
PROGRAM UNIT (MAIN)
STRING ARRAYS
NS ( )
   100
   120
   140
NUMERIC ARRAYS
NT ( )
   100
   120
   240
NUMERIC VARIABLES
I
   110
   120
   140
   160
   240
J
   150
   170
BASIC KEYWORDS
CALL
   130
   230
   240
   250
DATA
   190
   200
DIM
   100
FOR
   110
   150
GOSUB
   160
NEXT
   170
   180
PRINT
   140
READ
   120
REM
   220
RETURN
   260
STOP
   210
BASIC FUNCTIONS
&
   140
SUBPROGRAMS
CLEAR
   130
SOUND
   230
   240
   250
LINE REFERENCES
   220
   180
```

In condensed code format, when the left-most bit of a byte is "on," the software which reconstructs a program from the code is signaled that some special action will be required in the reconstruction process. In the case of line numbers, this principle applies to the first bit of the first of the two line-number bytes. When all bits except the left-most one are "on" in both bytes, the number represented in base 10 is 32767 (in binary, 01111111 11111111), the highest allowable line number in a program. When the left-most bit is added, the two-byte combination becomes an end-of-file mark. Thus the first two bytes of the last condensed format record must be CHR$(255)&CHR$(255), equivalent to 65535 in base 10.

With this information, you should be able to understand the basic operation of the EDITOR program. The program to be edited is first saved with the MERGE option, and then the EDITOR program is loaded and run. Upon entry of the "OLD" command provided, EDITOR inputs each record in the condensed format file and constructs the line number from the ASCII codes of the first two bytes. Program line numbers thus obtained are stored in an array, with array position corresponding to record number. After the user has altered these numbers using the DELETE (DEL) and RESEQUENCE (RES) commands provided, the SAVE command initiates the process in which altered numbers are reassigned to records in the file. As each record is read a second time, the corresponding line number in the array is translated into two ASCII characters which are substituted for those on the record, and the new record is written to a new file (after making the necessary changes to any line references). At the end of this process, the end-of-file mark is written as the last record on the new file. After initializing program memory with the NEW command, all you need to do is load the new file with the MERGE command. The program will then be reconstructed and can be SAVEd in the usual way.

CROSS REFERENCE

The remaining three programs (LINPUT, CREF, and CREFPRINT) are used to produce a complete tabulation of all lines in which each variable, array keyword, function, and line number reference occurs. An independent tabulation is provided for each subprogram. The cross reference table will give you detailed documentation for use in program development, and would also seem to be a useful tool in analyzing a poorly documented program. (See Table 2)

As in the case of the EDITOR program, the first step involves saving the program to be cross referenced by using the MERGE option. The LINPUT program converts the DISPLAY records of the merged file to INTERNAL code, presumably to speed subsequent execution. The CREF program then reads in each record of the file and analyzes its contents for the presence of all keywords, functions, etc., which occur in TI Extended BASIC, as well as in the user's variable names, arrays, line references, and subprograms. The output, a list of the line numbers in which each element is found, is written to a disk file. The file is then printed by the CREFPRINT program.

The instructions recommend that the CREF program be run in TI BASIC, rather than Extended BASIC, to speed execution. Even with this advantage, however, the cross referencing of a large program should be planned so that you can be doing something else—like taking a trip to Switzerland. Actually, it doesn't take quite that long: Cross-referencing a program of moderate size (270 lines) takes 35 minutes.

HOW CREF Works.

Although a detailed analysis of the cross reference program is beyond the scope of this article, generalization the principles involved presumes an understanding of the structure of condensed code. As mentioned previously, the method used to signal the reconstruction software that it is encountering an "instruction" byte involves an "on" condition in the left-most bit. In contrast to line numbers, most "instructions" in condensed code consist of a single byte. When the left-most bit is "on" (i.e., 10000000) the base 10 representation is 128. Instructions thus begin with the number 10000001 or 129

ASCII byte codes used by the reconstruction software to generate BASIC keywords, punctuation, etc., are translatable with the program *Condensed Format Code Table*. This program generates a file called DSK1.FILENAME which is in condensed format. Each record in the file contains a single byte in the third position beginning with ASCII 129 and ending with ASCII 254. This byte will be interpreted as an "instruction" by the reconstruction software. Preceeding the byte, a two-byte line number is written; following it is an end-of-line mark, ASCII 0. Line numbers have been set equal to the ASCII code for ease in subsequent interpretation of the results.

In order to view the reconstruction of each potential BASIC element, you first initialize program memory with

NEW command, then load the output file with the RGE command, as if it were a program, i.e., MERGE K1.FILENAME. The result is given in Table 3. For le:

HR$(129) is reconstructed as ELSE
HR$(130) as : :
HR$(166) as WARNING

```
100 REM  ********************
110 REM  *                  *
120 REM  *  CONDENSED FORMAT *
130 REM  *      CODE TABLE  *
140 REM  *                  *
150 REM  ********************
160 REM
170 REM
180 REM
190 REM
200 REM  ********************
210 REM
220 REM    OPEN OUTPUT FILE
230 REM     "DSK1.FILENAME"
240 REM    USING PARAMETERS OF
250 REM    MERGED FILE FORMAT
260 REM
270 REM  ********************
280 REM
290 OPEN #1:"DSK1.FILENAME",DISPLAY ,O
    UTPUT,VARIABLE 163
300 REM
310 REM  ********************
320 REM
330 REM   BEGIN COUNTING (I)
340 REM   WITH HIGH BIT ON
350 REM    I.E. 129 130 ... 254
360 REM
370 REM  ********************
380 REM
390 FOR I=129 TO 254
400 REM
410 REM  ********************
420 REM
430 REM   CALCULATE VALUES FOR
440 REM   FIRST TWO BYTES TO
450 REM   REPRESENT LINE NO.S
460 REM   SO THAT LINE NO WILL
470 REM   EQUAL ASCII CODE
480 REM
490 REM  ********************
500 REM
    LNBYTE1=INT(I/256)
    LNBYTE2=I-256*LNBYTE1
530 REM
540 REM  ********************
550 REM
560 REM   WRITE RECORD:
570 REM
580 REM   BYTE#1&2=LINE NUMBER
590 REM   BYTE#3=CODED BASIC
600 REM   BYTE#4=END OF LINE
610 REM
620 REM  ********************
630 REM
640 PRINT #1:CHR$(LNBYTE1)&CHR$(LNBYTE
    2)&CHR$(I)&CHR$(0)
650 REM
660 REM  ********************
670 REM
680 REM   REPEAT LOOP FOR NEXT
690 REM   ASCII CODE.
700 REM
710 REM  ********************
720 REM
```

```
730 NEXT I
740 REM
750 REM  ********************
760 REM
770 REM   WRITE END OF FILE
780 REM   MARK = LINE NUMBER
790 REM   OF 65535
800 REM
810 REM  ********************
820 REM
830 PRINT #1:CHR$(255)&CHR$(255)
840 REM
850 REM  ********************
860 REM
870 REM   CLOSE FILE AND STOP
880 REM
890 REM  ********************
900 REM
910 CLOSE #1
920 STOP
```

Table 3
Condensed Format Code Table

129 ELSE	171 ???	213 LEN
130 : :	172 ???	214 CHR$
131 !	173 ???	215 RND
132 IF	174 ???	216 SEG$
133 GO	175 ???	217 POS
134 GOTO	176 THEN	218 VAL
135 GOSUB	177 TO	219 STR$
136 RETURN	178 STEP	220 ASC
137 DEF	179 .	221 PI
138 DIM	180 ;	222 REC
139 END	181 :	223 MAX
140 FOR	182)	224 MIN
141 LET	183 (	225 RPT$
142 BREAK	184 &	226 ???
143 UNBREAK	185 ???	227 ???
144 TRACE	186 OR	228 ???
145 UNTRACE	187 AND	229 ???
146 INPUT	188 XOR	230 ???
147 DATA	189 NOT	231 ???
148 RESTORE	190 =	232 NUMERIC
149 RANDOMIZE	191 <	233 DIGIT
150 NEXT	192 >	234 UALPHA
151 READ	193 +	235 SIZE
152 STOP	194 -	236 ALL
153 DELETE	195 *	237 USING
154 REM	196 /	238 BEEP
155 ON	197 ^	239 ERASE
156 PRINT	198 ???	240 AT
157 CALL	199 ???	241 BASE
158 OPTION	200 ???	242 ???
159 OPEN	201 ???	243 VARIABLE
160 CLOSE	202 EOF	244 RELATIVE
161 SUB	203 ABS	245 INTERNAL
162 DISPLAY	204 ATN	246 SEQUENTIAL
163 IMAGE	205 COS	247 OUTPUT
164 ACCEPT	206 EXP	248 UPDATE
165 ERROR	207 INT	249 APPEND
166 WARNING	208 LOG	250 FIXED
167 SUBEXIT	209 SGN	251 PERMANENT
168 SUBEND	210 SIN	252 TAB
169 RUN	211 SQR	253 # (files)
170 LINPUT	212 TAN	254 VALIDATE

Table 4
Condensed Record Structure
OPEN #1:"DSK1.BASIC",INPUT,DISPLAY,VARIABLE 163

ASCII CODE FOR LINE 100

3 159*	23 179*
4 253*	24 162*
5 200*	25 179*
6 1	26 243*
7 49 1	27 200*
8 181*	28 3
9 199*	29 49 1
10 10	30 54 6
11 68 D	31 51 3
12 83 S	32 0
13 75 K	
14 49 1	
15 46 .	
16 66 B	
17 65 A	
18 83 S	
19 73 I	
20 67 C	
21 179*	
22 146*	PRESS ANY KEY TO CONTINUE

At the same time, several codes are reconstructed into things which can't be understood directly (e.g., 171-175, 185, and 198-201). It is apparent that some of the ASCII codes are used for purposes other than direct translation to BASIC. Some might be used as descriptors of subsequent bytes (e.g., for purposes of identifying trailing bytes as

numeric data, line numbers references, string data, etc.) while other of these ASCIII codes may not be assigned at all.

Putting this question aside for the moment, let us see how we could write a program that would remove all REM statements from another program. The ASCII code for REMARK (REM) is found in Table 3 to be 154. If we assume that the ASCII character with code 154 will be found in the third position of a REM statement in condensed format (following the line-number bytes), we can write a *REM Remover* program very simply. Such a program would need to read a record from a program file saved with the MERGE option, see if the third byte is CHR$(154), and if not, print the record in a second file. That is what the following program does. To use it with the "Condensed Format Code Table" program, save that program with the MERGE option (SAVE DSK1.CODE,MERGE), run the *REM Remover*, and load the output file, DSK1.REMFREE, with the MERGE command (MERGE DSK1.REMFREE). Presto, Chango! LISTing the program shows it to be "REMless," and this version may now be saved in the usual way under a new file name.

```
100 REM *********************
110 REM *      REM REMOVER    *
120 REM *********************
130 REM
140 REM
150 REM
160 REM
170 PRINT "ENTER FILE NAME"
180 INPUT "DSK1.XYZ":XS
190 OPEN #1:XS,DISPLAY ,INPUT ,VARIABL
    E 163
200 OPEN #2:"DSK1.REMFREE",DISPLAY,OU
    TPUT,VARIABLE 163
210 EOFS=CHR$(255)&CHR$(255)
220 LINPUT #1:XS
230 IF SEG$(XS,1,2)=EOFS THEN 270
240 IF SEG$(XS,3,1)=CHR$(154)THEN 260
250 PRINT #2:XS
260 GOTO 220
270 PRINT #2:CHR$(255)&CHR$(255)
280 CLOSE #1
290 CLOSE #2
300 STOP
```

Of course, more complex applications require a more detailed knowledge of condensed format structure. The *Condensed Record Structure* program listed below will allow you to examine the condensed structure of every line in any BASIC program. With such a representation and the list of codes in Table 3, a great deal of additional information can be deduced.

For purposes of illustration, let us treat the "Record Structure" program itself as the program to be analyzed. First enter the program without the REM statements, and then save it as DSK1.BASIC,MERGE. Now enter RUN to display the code structure of each line. The display for the first line is shown in Table 4.

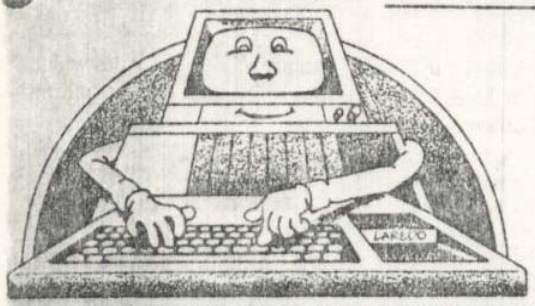
The first column in each pair of columns shows the position of the byte code. The first position displayed is 3 because 1 and 2 are used for the line number. An asterisk has been placed beside all ASCII codes which exceed 128 to easily identify them as "instruction" codes. Codes which are between 32 and 94 are followed by their corresponding ASCII character representations.

```
100 REM *********************
110 REM *                     *
120 REM *  CONDENSED RECORD   *
130 REM *       STRUCTURE     *
140 REM *                     *
150 REM *********************
160 REM
170 REM
180 REM
190 REM
200 OPEN #1:"DSK1.BASIC",INPUT ,DISPLA
    Y ,VARIABLE 163
210 LINPUT #1:XS
220 BYTE1=ASC(SEG$(XS,1,1))
230 BYTE2=ASC(SEG$(XS,2,1))
240 LINENUM=BYTE1*256+BYTE2
250 IF LINENUM=65535 THEN 430
260 DISPLAY AT(1,3)ERASE ALL:"ASCII CO
    DE FOR LINE "&STR$(LINENUM)
270 COL=1 :: J=0
280 FOR I=3 TO LEN(XS)
290 IF I>62 THEN 400
300 ROW=I-2*(COL-1)
310 I=I+1
320 DISPLAY AT(ROW,COL):STR$(I)
330 Y=ASC(SEG$(XS,I,1))
340 DISPLAY AT(ROW,COL+3):STR$(Y)
350 IF Y>128 THEN DISPLAY AT(ROW,COL+6
    ):"*"
360 IF Y>31 AND Y<91 THEN DISPLAY AT(R
    OW,COL+6):CHR$(Y)
370 IF I<20 THEN 390
380 COL=COL+10 :: J=1
390 NEXT I
400 DISPLAY AT(24,2)BEEP:"PRESS ANY KE
    Y TO CONTINUE"
410 CALL KEY(0,K,S):: IF S=0 THEN 410
420 GOTO 210
430 STOP
```

Since it is known that the first line of the program is OPEN #1;"DSK1.BASIC", INPUT, DISPLAY, VARIABLE 163, let us see what sense can be made of the corresponding condensed code. Codes 159 and 253 correspond to OPEN and #. Although the meaning of code 200 is not known, in looking ahead to colums 6 and 7 we might hypothesize that 200 means "A number is about to be encountered, and the next byte will give the number of bytes used to represent that number."

Although 181 is a ":", 199 is another unknown. Looking ahead at positions 10-20, we might again hypothesize that 199 is used for strings, in the way that 200 is used for numbers. The "10" in position 10 is consistent with this hypothesis since DSK1.BASIC is 10 characters long. Next, we encountered the codes for INPUT, DISPLAY, VARIABLE. In position 27 another 200 is encountered, and the hypothesis applied earlier to the 200 in position 5 is consistent with what follows—a "3" in position 28 followed by the 3 numbers "163". Finally, a 0 is encountered that indicates end-of-line. By writing program lines specifically for the purpose, you can use the *Condensed Record Structure* program to deduce additional information about condensed format.

How to Write A BASIC Program That Writes BASIC Programs



PART 2:

RULES OF MERGE FORMAT

In the previous section, MERGE format was discussed in connection with TI's *Programming Aids III*. When an Extended BASIC program is saved with the MERGE option (disk only), a data file is written such that each record in the file contains a coded representation of one line of BASIC code. This file can then be loaded into program memory with the MERGE command. Because the file is a data file, it can also be generated by a BASIC program. If all of the rules of MERGE format are observed, the file is indistinguishable from one created by saving a program with the MERGE option and can be loaded into program memory with the MERGE command. Thus an Extended BASIC program can, in effect, write another Extended BASIC program.

One can think of a variety of contexts in which this program generation capability could be used. For instance, a program might allow preparation of music or graphics in an interactive, "high level" format and then use this data to write a BASIC program or subroutine which produces the music or graphics display.

File Structure

The MERGE format file consists of sequentially organized records, each corresponding to one line of BASIC code. Records are of variable length with a maximum length of 163 display format characters. The OPEN statement for a MERGE format file might be:

```
OPEN #1:"DSK1.FILENAME",VARIABLE 163
```

Record Structure

Records in the file each represent a line of BASIC as strings of ASCII characters. The ASCII codes of the first two characters represent the line number, the last character designates "end-of-line", and the BASIC statement(s) are represented in coded form in between.

Let's consider first how line numbers are represented. You are probably aware that code numbers are associated with the character patterns used to display information. The character associated with a code can be obtained with the CHR$ function; PRINT CHR$(65) displays the pattern of the character with ASCII code 65 on the screen—the letter A. (ASCII, by the way, stands for American Standard Code for Information Interchange.)

While some ASCII characters, like the letter A, have an associated pattern, others do not. However, any of the 256 ASCII characters can be accessed with the CHR$ function and subsequently used in strings just like any of the more familiar characters. PRINT CHR$(32)&CHR$(255) displays two characters. Neither has a pattern, so neither can be seen, but the computer is able to recognize each character nevertheless.

A character consists of a "byte," and a byte can be thought of as an eight-place binary number. Just as the decimal number system contains 10 digits (0-9), the binary system contains two digits, 0 and 1. In the decimal system, the first place to the left of the decimal point counts in units of one. Each successive place counts in units of the number base 10 multiplied times the units of the preceeding place—i.e.,1's, 10's, 100's, 1000's, etc. Similarly the first place in a binary number counts 1 and successive places in units of the number base 2, multiplied times the units of the preceeding base; i.e., 1's, 2's, 4's, 8's, 16's, etc. Thus the eight-place binary number 00110001 is equivalent to 0+0+32+16+0+0+0+1 or 49 in decimal. The binary number 11111111 is equivalent to 255 (128+64+32+16+8+4+2+1), and this is the largest ASCII code because it is the largest number that can be represented with a byte. The 256 ASCII characters are thus numbered from 0 through 255.

The decimal equivalent of ASCII code is used to represent the line number, but with only one byte, the largest line number which could be represented is 255. To allow representation of high line numbers, a second character is added giving a total of 16 binary places. Applying the same principle used above, the places count (from right to left) in units of 256 (128*2), 512, 1024, etc. When placed in the first two positions of a MERGE format record, CHR$(2)&CHR$(8)—i.e., 00000010 00001000—would represent the line number 520 (512+8). A quick method of determining the decimal representation of any two characters is to multiply the code of the first by 256 and add the code of the second. In the above example, 520=2*256+8.

The highest allowable line number in TI BASIC is 32767 (01111111 11111111 or CHR$(127)&CHR$(255)). Adding the left digit gives the end-of-file mark used in MERGE format, equivalent to a line number of 65535. These two bytes, CHR$(255)&CHR$(255) must be in the first two positions of the last record in a MERGE format file.

Just as these two characters signal the end of the file, the byte CHR$(0) is used to signal the end of each line. This character must be the last one in each record.

MERGE Format Code

This brings us to the question of what to put between the line number and end-of-line mark and before the end-of-file mark, viz., the coded BASIC statements. Many elements which comprise Extended BASIC statements are listed in Table 3 together with their ASCII character tokens. In MERGE format, the BASIC elements listed are represented by a single ASCII character. For instance, CHR$(156) represents PRINT, CHR$(130) the statement separator, CHR$(213) the LEN function, etc. In order to prepare BASIC statements in MERGE format, however, one must also know how to represent variable names, numeric and string constants, and line numbers occurring within statements.

The easiest of these to represent is the variable name; the normal ASCII representation for each character of the name is used. Consider the line:

```
10 PRINT XYZ
```

The MERGE format record used to represent this line would be:

```
CHR$(0)&CHR$(10)&CHR$(156)&"XYZ"&CHR$(0)
```

That is, seven bytes would be concatenated in a string and written in the appropriate disk file record. The first two bytes represent the line number; the next, the keyword PRINT; the next three, the variable name; and the last, the end-of-line mark. Assuming that the complete file corresponds to the requirements of MERGE format in other respects, when loaded into program memory with the MERGE command LISTing, the program will show it to contain the line intended.

Numeric constants and unquoted string constants are handled differently from variable names: Each number of unquoted string must be preceded by two identifying bytes. The first is CHR$(200), the character which signals the beginning of an unquoted string. Following CHR$(200), a byte must be included to indicate the number of subsequent characters in the string or number. This byte is simply the character with the code equal to the length of the string—i.e., if the string were five characters long, CHR$(5) must be included; if 12 characters, CHR$(12). For example, consider the statement,

```
10 PRINT X+345
```

The statement would be represented in MERGE format with 11 bytes as follows:

```
CHR$(0)&CHR$(10)&CHR$(156)&"X"&CHR$(193)
&CHR$(200)&CHR$(3)&"345"&CHR$(0)
```

Here, CHR$(200)&CHR$(3)&"345" first indicates that an unquoted string is to be encountered, then indicates how long that string is, and finally gives the string.

Quoted strings are handled in much the same way, except that CHR$(199) is used instead of CHR$(200):

```
10 RUN "DSK.1.FILENAME"
```

would be represented as

```
CHR$(0)&CHR$(10)&CHR$(169)&CHR$(199)&
CHR$(13)&"DSK1.FILENAME"&CHR$(0)
```

Notice that quote marks are *not* explicity included in the string representation. They are automatically provided for by the use of CHR$(199).

Finally, line numbers included in program statements such as GOTO and GOSUB must consist of two bytes coded in the same way as the line number bytes which begin each record. Moreover, these two bytes must be preceded by CHR$(201) to indicate that they are to be interpreted as a line number. The statement:

```
10 GOTO 200
```

would be represented as follows:

```
CHR$(0)&CHR$(10)&CHR$(134)&CHR$(201)
&CHR$(0)&CHR$(200)&CHR$(0)
```

Program Generation

Although MERGE format programs can be generated with the above technique, its use would be cumbersome—to say the least. The following method simplifies the process considerably.

For the moment, let's put aside the question of generating the portion of the character string associated with the BASIC statement. Assume that this string is generated and assigned to the string variable LINE$. Each time a LINE$ string is constructed, two line number bytes must be added to the beginning, an end-of-line byte to the end, and the whole thing must then be written as a record in the MERGE format file. The easiest way to handle the operations which follow the construction of LINE$ is to use a subroutine. Given a starting line number, LN, the following subroutine constructs the two-byte ASCII line number representation and writes the file record. It then increments the line number by 10.

```
9000 PRINT #1: CHR$(INT(LN/256))&CHR$
(LN-256*INT(LN/256))&LINE$&CHR$(0) ::
LN=LN+10 :: RETURN
```

After the BASIC statement portion of the record is assigned to LINE$, a simple GOSUB 9000 takes care of all the rest.

The construction of LINE$ strings can be simplified by assigning ASCII character codes to string variables with easy to remember names. For instance:

```
100 REM$=CHR$(154)::FOR$=CHR$(140)::NEXT$
=CHR$(150)::IF$=CHR$(132)::THEN$=CHR$(176)
::TO$=CHR$(177)
```

Some string functions are followed by a "$" and are reserved words. But in TI BASIC, they can be embedded in a variable name so that one could use variable names like @SEG$, @STR$, etc., for storage of the appropriate ASCII character. Punctuation, arithmetic operators, and characters 199-201 also must be assigned "creative" string variable names: Q$ for quoted string, UQ$ for unquoted string, CM$ for comma, etc.—whatever will be easiest for you to remember.

The next level of simplification involves user-defined functions to include more than one byte whenever possible. For example, it is clear that CALL will always be followed by an unquoted string; CALL COLOR, CALL SPRITE, CALL SOUND, etc. For that matter, the unquoted string token will always be followed by a byte indicating string length. Construction of strings which in-

clude the call keyword can therefore be simplified by defining function appropriately:

```
110 DEF UQ$(X)=CHR$(200)&CHR(X)::CALL$(X)
=CHR$(157)&UQ$(X)
```

atement like CALL SCREEN (2) can then be written:

```
120 LINE$=CALL$(6)&"SCREEN"&LP$&UQ$(1)&
"2"&RP$::GOSUB 9000
```

(if CHR$(183), the left parenthesis, had previously been assigned to LP$ and 182, the right parenthesis, to RP$)

By making the function definitions a little more complex, the statement can be even further simplified:

```
110 DEF UQ$(X$)=CHR$(200)&CHR$(LEN(X$))&X$
120 DEF CALL$(X$)=CHR$(157)&UQ$(X$)
```

makes it possible to write CALL SCREEN (2) like this:

```
130 LINE$=CALL$("SCREEN")&LP$&UQ$("2")&
RP$::GOSUB 9000
```

t's beginning to look a lot like BASIC.

Built-in functions can similarly be defined to facilitate onstrucion of MERGE format strings. For instance,

```
40 DEF INT$(X$)=CHR$(207)&LP$&X$&RP$
```

lo one to write X=INT(Y/256) as

```
50 LINE$="X"&EQ$&INT$("Y"&DIV$&UQ$
"256"))::GOSUB 9000
```

f CHR$(190) had been previously assigned to EQ$ and HR$(196) TO DIV$)

Similarly, line numbers occurring within statements, ich as GOTO or GOSUB, can be simplified with the llowing function:

```
50 DEF LN$(X)=CHR$(201)&CHR$(INT(LN/256))
CHR$(LN-256*INT(LN/256))
```

that the statement GOTO 200 can be written simply as

```
70 LINE$=GOTO$&LN$(200) :: GOSUB 9000
GOTO$ had previously been assigned CHR$(134))
```

Using string variable names and user-defined string nctions, you can create your own custom "language" use in writing MERGE format records.

The following program may help to tie up the concepts esented; it is a trivial example of a music program ne tor. The program writes CALL SOUND tements in the MERGE format file "DSK1.BASIC" the user presses a single key.

```
100 REM **MUSIC PROGRAM GENERATOR**
110 REM
120 REM
130 REM
140 REM ASSIGN STRING VARS
150 REM
160 CM$=CHR$(179)::  LP$=CHR$(183)::  RP
    $=CHR$(182)
170 REM
180 REM DEFINE STRING FUNCTIONS
190 REM
200 DEF UQ$(X$)=CHR$(200)&CHR$(LEN(X$)
    )&X$  ::  DEF CALL$(X$)=CHR$(157)&UQ
    $(X$)
210 REM
220 REM ASSIGN FIRST LINENO
230 REM
240 LN=100
250 REM
260 REM OPEN MERGE FILE
270 REM
280 OPEN #1:"DSK1.BASIC",VARIABLE 163
290 REM
300 REM DISPLAY INSTRUCTIONS
310 REM
320 DISPLAY AT(7,1)ERASE ALL:"TO ENTER
     A NOTE PRESS ONE OFTHE FOLLOWING
    KEYS:"
330 DISPLAY AT(12,5):"A B C D E F G":
    :  DISPLAY AT(20,7):"PRESS P TO STO
    P"
340 REM
350 REM ACCEPT KEY INPUT
360 REM
370 CALL KEY(0,KEY,STATUS)::  IF KEY=80
     THEN 500 ELSE IF KEY<65 OR KEY>71
     THEN 370
380 KEY=2*KEY-130 ::  IF KEY>2 THEN KEY
    =KEY-1 ::  IF KEY>7 THEN KEY=KEY-1
390 REM
400 REM COMPUTE FREQUENCY
410 REM AND PLAY NOTE
420 REM
430 FREQ=INT(440*(2^(1/12)))^KEY+.5)::
    CALL SOUND(500,FREQ,0)
440 REM
450 REM FORM MERGE STATM'T
460 REM
470 LINE$=CALL$("SOUND")&LP$&UQ$("500"
    )&CM$&UQ$(STR$(FREQ))&CM$&UQ$("0")
    &RP$  ::  GOSUB 490
480 GOTO 350
490 PRINT #1:CHR$(INT(LN/256))&CHR$(LN
    -256*INT(LN/256))&LINE$&CHR$(0)::
    LN=LN+10  ::  RETURN
500 REM WRITE END OF FILE
510 REM
520 PRINT #1:CHR$(255)&CHR$(255)
530 CLOSE #1
540 REM
550 REM LOAD INSTRUCTIONS
560 REM
570 DISPLAY AT(12,1)ERASE ALL:"TO LOAD
     THE PROGRAM:"  ::  DISPLAY AT(15,2)
    :"1> ENTER 'NEW'"
580 DISPLAY AT(17,2):"2> ENTER 'MERGE
    DSK1.BASIC'"  ::  DISPLAY AT(19,2):"
    3> ENTER 'RUN'"  ::  END
```

HOW EXTENDED IS EXTENDED BASIC?

Nothing caused as much excitement and anticipation in the TI-99/4A community as the announcement (which now seems like an eternity ago) that Extended BASIC would be forthcoming. Well, now that the new programming language is being gobbled up by hungry Home Computer users, the question on everyone's mind is, naturally enough, "Was it worth waiting for?"

For the answer to this, and to help put the new software in proper perspective, we should first examine TI's claims for the language (in the introduction to the reference manual): "Texas Instruments Extended BASIC. . .has the features expected from a high level language plus additional features not available in many other languages, including those designed for use with large, expensive computers." The key words here are "expected" and "not available." Features such as DISPLAY AT, ACCEPT AT, PRINT. . .USING, IMAGE, ON ERROR, multiple statement lines, expanded IF-THEN-ELSE statements, PEEK, Boolean operators, and assembly language subroutine calls are indeed "expected." Unfortunately, they were expected in the *ordinary* TI BASIC, since they're *standard* features of various Microsoft BASICs found in other machines. But just as plain, old, ordinary TI BASIC has its share of surprises that *aren't* commonly found in other BASICs (e.g., CALL SAY, RESEQUENCE, *complete* EDIT, TRACE, and BREAK utilities, plus its marvelously simple character definition and color assignment facilities), TI Extended BASIC also has its own unique bag of tricks *not found* on other machines. And this bag of tricks includes some mighty impressive feats of computing magic.

But before we get into these *extended* features, let's examine some of the obvious changes from TI BASIC. First, there's the matter of a slight reduction in usable RAM. The maximum program size in Extended BASIC is 864 bytes smaller than in TI BASIC. Although this represents only about a 6% reduction, *any* reduction in user memory is significant if it prevents certain applications from being RUN. And, in fact, as little as 500 bytes is frequently the critical amount of extra memory needed. (Witness the several programs in this volume that cannot be loaded or RUN with the disk controller's power on—even with the CALL FILES(1) command that frees all but the 500 bytes for the disk system.) So programmers without the 32K RAM expansion should try wherever possible to make up the loss with Extended BASIC's built-in memory saving features: multiple statement lines (with more allowable characters per line), expanded IF-THEN-ELSE statements, multiple variable assignments, trailer comments that immediately follow statements (instead of separate REMs), repetition of strings with the RPT$ function, and the use of MIN and MAX functions.

The loss of user-definable characters in the character sets 15 and 16 is another departure from the TI BASIC standard. These custom characters are no longer available to programmers since the memory area is needed to keep track of sprites. Therefore, a TI BASIC program that doesn't use these character sets is supposed to RUN in Extended BASIC in most circumstances—unless, of course, you've done something that will obviously cause trouble, such as accidentally using a TI Extended BASIC keyword as a variable in your TI BASIC program (e.g., DIGIT, ERASE, ERROR, IMAGE, MERGE, MAX, MIN, SIZE, WARNING, etc.) [See the July/August 1981 issue of *99'er Magazine* for an analysis of what *is* and *isn't* interchangeable.—Ed.]

Now, let's take a peek (no pun intended) into the "bag of tricks" I mentioned earlier. A good place to start is with Extended BASIC's exciting new graphics capabilities. Nine new subprograms (plus 2 redesigned ones) provide the ability to create and thoroughly control the shape, color, and motion of smoothly-moving high-resolution graphics. These are the true *sprites*—graphics that can be displayed and moved at any of 49,152 positions (192 rows × 256 columns) rather than the 768 positions (24 rows × 32 columns) CALLed by the VCHAR and HCHAR statements of TI BASIC. But that's only the beginning. Sprites can be set in motion with simple X and Y velocity components and will continue their motion without further control; they can grow and shrink at will, be relocated or "hidden", and even pass over and blot out fixed objects and other sprites to give the illusion of depth and 3-D animation. [This is a function of the three-dozen stacked image planes of the Home Computer's video display processor chip—a unique graphics display explained more fully in "3-D Animation."—Ed.]

Although games aficionados and educators have every right to be overjoyed with the new sprites capability, TI-99/4A users who are more interested in business, scientific and professional applications will be drawn to other Extended BASIC features. First on the list is the impressive subprogram capability. Several options exist for passing values (and entire arrays) between main and

subprograms. There's also built-in protection to prevent subprogram's *local* variables from affecting the *main* variables. Additionally, commonly used subprograms may be SAVEd on a separate disk, and later MERGEd. This will allow programmers to build up a library of "universal" subprograms that can be called upon to supply the appropriate cartridges for new programming tasks—without time-consuming re-coding and debugging.

If this new subprogram flexibility is not enough for your most demanding tasks, how about "program chaining," where one program can load and RUN another program from a disk. This means that multi-part programs of almost *unlimited* size can now RUN on the TI-99/4A if they are broken into pieces and each segment is allowed to RUN the next. And at any point in this chain, a "menu" may be inserted, allowing the user to choose with a *single keystroke* the particular program to be RUN. Imagine the possiblities!

Those of you with a speech synthesizer, or thinking of purchasing one, will be happy to learn that Extended BASIC includes a speech editor. You will no longer need the separate Command Cartridge (with a retail price of about $45). What's more, with the combination of CALL SPGET, the capability of subroutine MERGEs, and the data for the code patterns (that TI supplies in the appendix of the reference manual), you can now easily add the suffixes ING, S, and ED to the roots of words in the resident vocabulary. And if TI ever supplies users with their master file of coded speech patterns and rules for combining them, it will be possible to create your own new words. As of now, TI provides only one cryptic statement: "Because making new words is a complex process, it is not discussed in this manual."

Incidentally, this capability of having the computer say what *you* want it to say rather than being limited to a fixed vocabulary will, in fact, be implemented through a related approach. I'm referring to the "text-to-speech" capability of the forthcoming *Terminal Emulator II* Command Cartridge, which is programmable in TI BASIC. Since only one Command Cartridge at a time can be attached to the TI-99/4A, text-to-speech cannot be used with the *Extended BASIC* Command Cartridge. [See "Text to Speech on the Home Computer."—Ed.]

The final two features I'm going to cover in this overview provide a fair degree of software protection and open the door to additional language capabilities. Consequently, these are the particular features that may have the most profound impact on the entire TI-99/4A community—ultimately determining the quality and quantity of most of the commercial software for this machine.

Extended BASIC programs can be SAVEd in a PROTECTed form to guard against software piracy. This irreversible feature allows a program to be RUN or loaded into memory only with an OLD command. A program thus PROTECTed cannot be LISTed, EDITed, or SAVEd. If the program was originally SAVEd and PROTECTed on a disk, you must still use the protect feature of the *Disk Manager* Command Cartridge to completely "lock up" the software by preventing it from being copied as well.

Extended BASIC has the capability to CALL and RUN assembly language programs if the 32K RAM expansion peripheral is attached to the computer. Since Assembly Language has a much faster execution speed than BASIC, many applications programs that are unfeasible to write in either TI BASIC or TI Extended BASIC (and Extended BASIC is not significantly faster than its predecessor) can now be written in TMS9900 Assembly Language, LOADed into the expansion memory peripheral, and RUN on a TI-99/4A. This paves the way for some fairly sophisticated applications programs that can now be targeted for TI-99/4A users. [See the related assembly language sections in this book.—Ed.]

Even though a TI-99/4A with Extended BASIC and the memory expansion peripheral can CALL and RUN Assembly Language programs and subroutines, it cannot be used to *write* them *at present*. And instead of a direct implementation of the POKE command, TI gave users an indirect implementation. To load data directly into memory locations, they can use CALL LOAD with the optional fields specifying a starting address followed by data bytes. The TMS9900 Assembler, available on the *Editor/Assembler* Command Cartridge and its accompanying diskettes, allows Home Computer owners to write their own Assembly Language programs and call them up through Extended BASIC. Besides this obvious use of an assembler, it opens up other exciting possibilities: More exotic languages can be written in TMS9900 Assembly Language especially for TI-99/4A implementations. FORTH, for instance, is now available.

The bottom line is more software tools for developers and more economic incentive for them to produce valuable programs that can be protected against most piracy. This means that the TI-99/4A user community will be seeing a lot more useful software enter the market. Being able to run this software should more than justify the $100 (retail) price for this filled-to-capacity 36K byte TI *Extended BASIC* Command Cartridge with accompanying 224-page reference manual. Therefore, the answer to the title's rhetorical question, "How Extended is Extended BASIC?" is apparently, "Extended enough. . . ."

POCKET TOWER OF HANOI

You are in an ancient temple at the center of the earth where three diamond needles bear eighty golden rings of graduated sizes. At the beginning of time the rings were all on one needle; but now the temple monks are transferring the rings, one at a time, from needle to needle, never setting a ring on a smaller ring. When they have moved all eighty rings to one of the other two needles, the world will end . . .

Possibly you have seen a children's toy along these lines—four or five disks of various colors and sizes, drilled to fit on three wooden pegs. The object is to start with the disks on one peg, and by moving one at a time—and never setting a disk on a smaller one—transfer the entire pile to another peg. If you don't have one of these in your closet, here is a pocket program of the puzzle for you and your friends.

When the program is run, four "rings" (they will actually look more like short bars) will appear on the left of the screen. There is room on the screen for three piles of rings. (To make the game pocket-sized, the pegs were left out.) To move a ring from one pile to another, press key 1, 2, or 3 to designate which pile (left, center, or right) to take the ring from, and then press 1, 2, or 3 to designate which pile to move the ring to. That's all there is to it.

The program works this way: rings are represented by the numerals 1, 3, 5, and 7. Peg (1), Peg (2), and Peg (3) are variables in which the presence of rings on the three pegs (or piles) are recorded. Thus in line 200, which is part of the initial setup portion of the program, Peg (1) is given the value 1.357 corresponding to the presence of all four rings on the first peg. The leftmost numeral is the one on top.

At the beginning, pegs #2 and #3 are empty. When a ring is moved from one peg to another, the values of the "peg( )" variables change accordingly. For example, if our first move is to place the top ring from peg #1 onto peg #2, then Peg (1) changes from 1.357 to 3.57 and Peg (2) changes from 0 to 1.

These changes are performed in line 450 (where the "size" of the ring being moved is figured out) and in lines 500 and 510 where the values of the "peg( )"s are actually changed. "From" and "too" identify the pegs. They are given values when the keys 1, 2, or 3 are pressed. The three "top( )" variables are strictly for the graphic display; they record the positions of the tops of the piles on the screen. Conveniently, the rings are 1, 3, 5, and 7 characters wide.

```
100 REM  *********************
110 REM  *  TOWER OF HANOI  *
120 REM  *********************
130 REM
140 REM
150 REM
160 REM
170 DIM PEG(3),TOP(3)
180 CALL COLOR(7,1,1)
190 CALL COLOR(8,2,2)
200 PEG(1)=1.357
210 PEG(2)=0
220 PEG(3)=0
230 TOP(1)=10
240 TOP(2)=14
250 TOP(3)=14
260 CALL CLEAR
270 CALL HCHAR(10,6,88,1)
280 CALL HCHAR(11,5,88,3)
290 CALL HCHAR(12,4,88,5)
300 CALL HCHAR(13,3,88,7)
310 CALL KEY(3,FROM,STATUS)
320 IF STATUS=0 THEN 310
330 CALL KEY(3,DUMMY,STATUS)
340 IF STATUS=-1 THEN 330
350 FROM=FROM-48
360 CALL SOUND(100,110,3)
370 CALL KEY(3,TOO,STATUS)
380 IF STATUS=0 THEN 370
390 CALL KEY(3,DUMMY,STATUS)
400 IF STATUS=-1 THEN 390
410 TOO=TOO-48
420 CALL SOUND(100,262,2)
430 IF (FROM<1)+(FROM>3)+(TOO>3)+(TOO<
    1)THEN 310
440 IF (PEG(FROM)=0)+((PEG(TOO)<>0)*(P
    EG(FROM)>PEG(TOO)))THEN 310
450 SIZE=INT(PEG(FROM))
460 TOP(TOO)=TOP(TOO)-1
470 CALL HCHAR(TOP(FROM),3+(FROM-1)*9+
    .5*(7-SIZE),87,SIZE)
480 TOP(FROM)=TOP(FROM)+1
490 CALL HCHAR(TOP(TOO),3+(TOO-1)*9+.5
    *(7-SIZE),88,SIZE)
500 PEG(FROM)=10*(PEG(FROM)-SIZE)
510 PEG(TOO)=.1*PEG(TOO)+SIZE
520 GOTO 310
```

"Status" is used as part of the "call key" routine tell the machine when a key has been released so that the program can go ahead. Now read through the program and see if you can follow what is happening.

Stacks

The piles of rings in this program are particularly graphic illustrations of the *stack*, a ubiquitous and very important idea in practically every kind of software. Like the rings in these piles, things stored on *software stacks* (subroutine return addresses, interrupts, whatever . . .) come off the stacks in reverse order to the way they went on. Because the items stored on our pegs are only single numerals, we are able to use a simple "trick"[1] to represent each of our three stacks. We just construct a number for each digit we want to represent. The 99/4A employs numbers accurate to 13 decimal places using a radix-100 representation, so we can push and pop numerals onto and off the left end of these with abandon, multiplying and dividing by 10 without fear of a roundoff error.

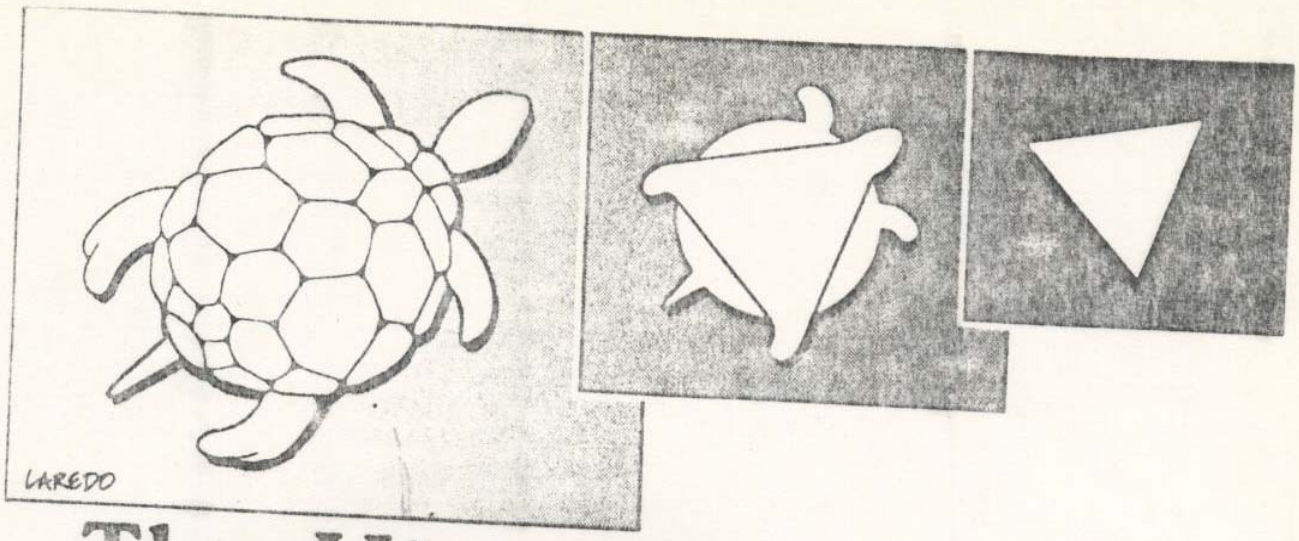
[1] If you find an application for this "trick" in a program of your own, you be entitled to call it a "method." (A "method" is a trick used twice).

4
LOGO

4

LOGO

A learning environment on your Home Computer.



The History of LOGO

LOGO—a powerful, high-level computer language designed for educational purposes especially as a programming language suitable for young children—is now available on Texas Instruments' TI-99/4A Home Computers. For more than a dozen years, the LOGO Group at the Massachusetts Institute of Technology has been developing the LOGO language and related computer programming activities. Under the leadership of MIT Professor Seymour Papert, LOGO activities have been used with children as young as nursery school age, with MIT undergraduates, and with many students of all ages in between. The philosopy of LOGO's developers has been: "No threshold, no ceiling." A beginner can make the computer do something meaningful and interesting in the very first programming session. Yet at the other extreme, LOGO is suitable for very advanced programming projects.

The philosophy of LOGO has been derived primarily from two sources: The developmental theories of the late Swiss psychologist, Jean Piaget (with whom Seymour Papert worked for several years before coming to MIT), and ideas from a modern scientific field called Artificial Intelligence. From Piaget comes the idea of creating learning environments in which most of what children learn can occur naturally—in the same way children learn to speak their native language, walk or run, and play ball. From Artificial Intelligence comes ideas about ways to use programming languages to aid thinking and problem-solving. Programming a computer in LOGO is seen as the act of teaching the computer a set of new commands, based on what it already knows how to do. Each user is, in effect, creating his or her own computer language, to suit his or her own purposes. Readers interested in learning more about these ideas should read *Mindstorms*, a recent book by Seymour Papert, in which he develops and extends the vision of the relationship between computers and learning that led to his development of LOGO.

LOGO activities are designed to allow use of the computer in a way that is personally meaningful to the user. Activities developed by the MIT LOGO Group have included using a computer to control the behavior of a robot turtle, to draw pictures and explore geometric environments on a TV screen, to create computer animations, invent interactive computer games, compose, arrange, and play music, and produce "poetry." The best known LOGO activity is using a simulated robot turtle on a TV screen to produce geometric designs and cartoon-like drawings. Hundreds of children have learned computer programming and problem-solving skills and developed mathematical expertise while writing programs for the turtle.

The LOGO language includes commands to make the turtle move and draw pictures. A student drawing with the turtle can make it move around on the TV screen by typing familiar commands such as FORWARD and BACK or RIGHT and LEFT. The information which beginners need to control the turtle is already present in their own body knowledge of how to move forward or back and how to turn right and left. Programming becomes an extension of something a learner already knows—rather than something requiring the mastery of an elaborate technical language or a complex coordinate system. The turtle becomes for the learner what Seymour Papert has called "an object to think with." Students using the computer as a programming tool become more aware of both their own body motion and the behavior of the computer.

The version of LOGO developed collaboratively by Texas Instruments and the MIT LOGO Group for the TI-99/4A includes an entirely new graphics environment called a "Sprites World." Sprites are small objects that can move rapidly around the screen, changing shape, color, speed and direction. Large numbers of sprites can appear at the same time to produce exciting animated designs or to be used as elements in programs to create video games. Because of its inherent attraction for so many people and because of the geometric and problem solving ideas embedded in it, the Sprites World promises to be one of the most exciting computer-based learning environments yet invented.

The World of the Turtle

Let's take a closer look at what actually happens when someone learns to program a computer using the LOGO turtle. The turtle responds to simple commands typed at the

keyboard: FORWARD 100, BACK 50, RIGHT 90, LEFT 45, etc. FORWARD 100 moves the turtle forward "100 turtle steps," drawing a line on the TV screen in the process. LEFT 45 makes the turtle rotate 45 degrees to its own left. People learning LOGO find it natural to "identify" with the turtle, imagining themselves going through its motions as it carries out a particular task. At the same time, controlling the turtle becomes a metaphor for controlling the computer itself: Like the turtle, the computer responds to an ordered series of command, and "TO" procedures that are defined as series of commands.

The ways in which the actions of the turtle can lead to geometric designs, as well as the method used to define procedures, is illustrated in the following simple examples. The turtle can draw a square by repeating the commands FORWARD 100 RIGHT 90 four times. A *procedure* can be defined by choosing a name (BOX, for example) and typing in a series of commands in order.

```
TO BOX
FORWARD 50
RIGHT 90
FORWARD 50
RIGHT 90
FORWARD 50
RIGHT 90
FORWARD 50
RIGHT 90
END
```

To execute BOX enter the following:

```
TELL TURTLE
BOX
```

When the new command, BOX, is typed, the turtle immediately draws the shape shown in the figure. (The small triangle shown in the figure represents the turtle by showing its position and heading). A similar procedure, TRI, can be defined as follows:

```
TO TRI
FORWARD 50
RIGHT 120
FORWARD 50
RIGHT 120
FORWARD 50
RIGHT 120
END
```

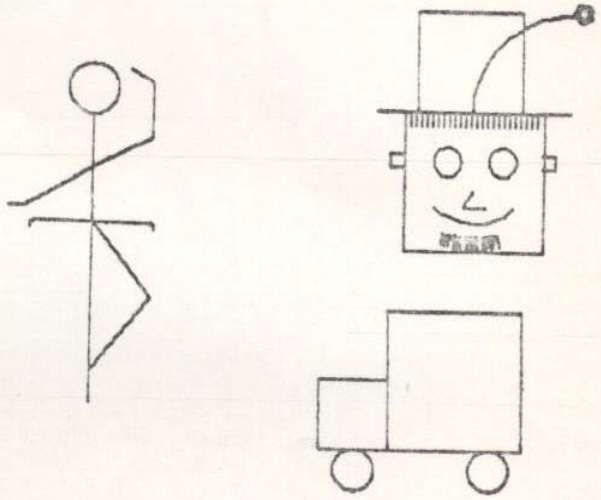
To execute TRI enter the following:

```
TELL TURTLE
TRI
```

A student who has defined procedures such as BOX and TRI is beginning to "teach the computer" his or her own private language. BOX and TRI can now be used in the same way as other LOGO commands. They can be used to create other drawings such as a simple "house" or an abstract geometric "flower."

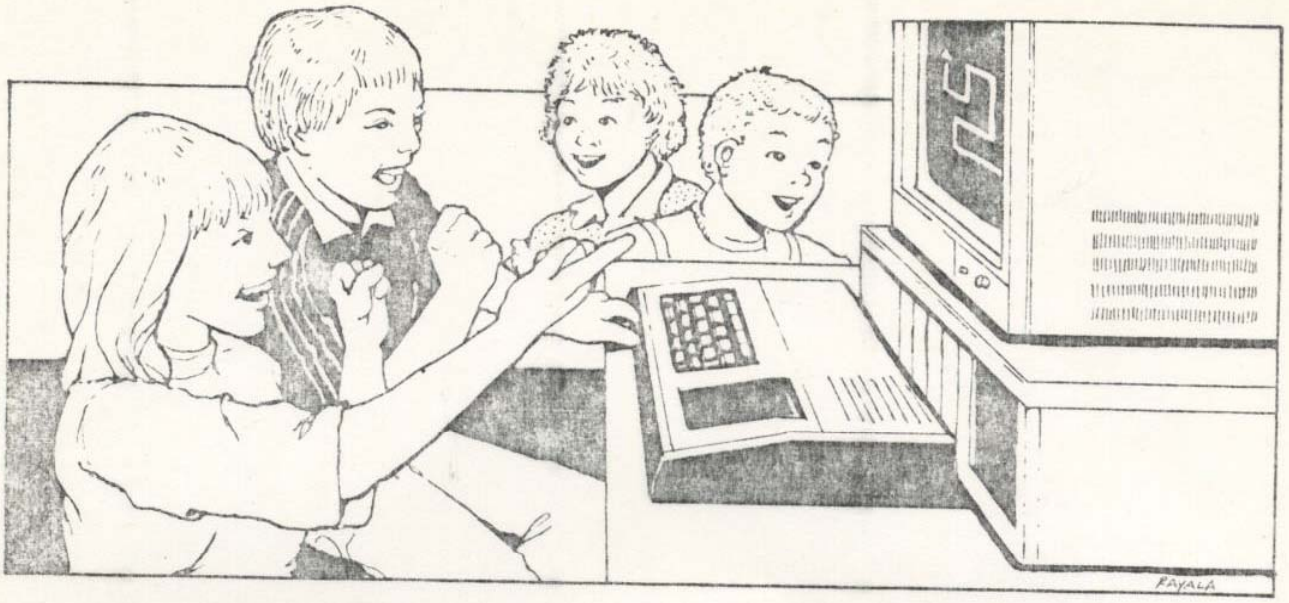
This approach to geometry and programming provides the basis for a rich universe of activities known as Turtle Geometry, which includes cartoon drawings (simple and complex), geometric designs, mathematical theory building, and computer games. Extensions of Turtle Geometry have proven fruitful when used with advanced high school students or MIT undergraduates. The universe of Turtle Geometry provides a conceptual framework for such aspects of mathematics as the relation between shapes and angles, coordinate systems, positive and negative numbers, the use of variables, symmetry and similarity, and even calculus and differential geometry. The computer programming involved in beginning LOGO activities can include procedures and subprocedures, the naming of procedures and variables, procedural hierachy, recursion and iteration, the use of conditional logic, and the development of problem-solving strategies.

Within the universe of Turtle Geometry, there is room for different students working individually to create their own sub-universes or *microworlds*. They can do this with their own limited (but expandable) sets of concepts and related activities and projects. To teach LOGO is really to help learners create, explore, and extend their own microworlds.



I have used turtle geometry as an *example* of what can be done with LOGO because it is easy for a reader to visualize the commands and to see how they lead to procedures that produce the results in the pictures—just as it is for young children. Children learning LOGO have actually carried out many other types of projects as well: moving turtles, finding their way around race-tracks or mazes, animated cartoons, interactive computer games such as Nim or Tic-Tac-Toe, programs which generate sentences or poetry (or even play Mad-Libs), and programs to translate English into Morse Code, or vice-versa. As LOGO becomes available to owners of TI-99/4A computers, I hope that these pages can be a forum for describing *your* LOGO projects. Since there will soon be more LOGO users than ever before, we can expect more and different LOGO projects to emerge. One of the best ways to build the culture of LOGO is for users to share project ideas through the pages of books such as this or magazines such as *99'er Home Computer Magazine*.

Although TI LOGO is a recent entry to the LOGO World, a prototype version has already been tested with hundreds of students between the ages of three and nine at the Lamplighter School in Dallas, Texas, and by students in fifteen elementary and junior high schools in New York City. Using the Sprites World of animated graphics activities, these students are busily creating a new universe of LOGO activities to delight and educate a new generation of computer users. In an age in which computers are omnipresent in society, and in which universal computer literacy is a pressing national need, computer-based learning environments like LOGO have become essential to the process of growing up literate in the last decades of the twentieth century.



The Lamplighter LOGO Project

"A child is not a vessel to be filled, but a lamp to be lighted." The quote from Alexandrov is on the plaque outside the Lamplighter school. That sign advises any visitor that the school is very unusual.

The curriculum at Lamplighter is individually tailored to meet the needs of each student. Individualization is applied in science, language arts, math, drama, music, art, French, and physical education. The Lamplighter is strongly supported by the parents of its students and by its alumni, with graduates of Lamplighter frequently dropping by to see their former teachers. Such alumni loyalty might not be considered unusual, except that the Lamplighter classes begin with preschool (age 3) and end with the fourth grade-level!.

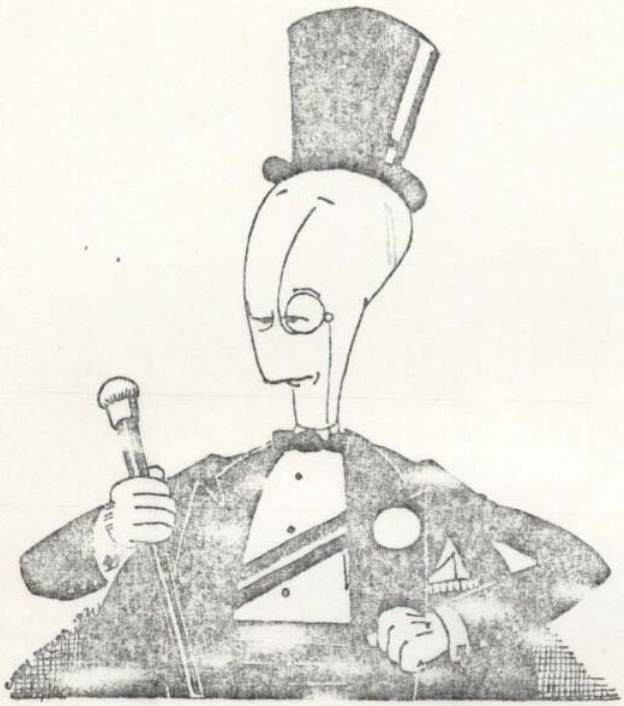
The physical arrangement of the school reinforces its approach to learning. Classrooms have only three walls; the fourth side of each class opens onto an airy, bright shared space. Class rooms are clustered around these shared-spaces by grade-level. Inside each classroom there are tables and chairs for writing work and, on one side, a small tiered well which is used for many other activities (e.g., reading, French, music, or story telling). The staff, the facilities, the students, and the parents all contribute to make Lamplighter a very special private school.

Lamplighter has been a leader in the use of new technology for learning. Calculators, *Speak & Spells*, Systems 80 units, and Little Professors are abundant throughout the school. Students regularly use these learning tools and other learning games found in the shared spaces. Teachers make extensive use of slides, films, and video and audio tapes. When Mr. Erik Jonsson (co-founder of Texas Instruments and Lamplighter Board of Directors Chairman and benefactor) first proposed introducing computers into Lamplighter, his idea was well received. Mr. Jonsson had earlier been in contact with Dr. Seymour Papert of the Division for Study and Research in Education (DSRE) at MIT, and found the LOGO language and philosophy of learning intriguing. Papert's initial explanation that LOGO allowed students to program computers and not vice-versa, enjoyed a favorable reception from the Lamplighter faculty. Later, as Papert elaborated on the LOGO philosophy, it became clear the LOGO was very much in accord with the philosophy and practice of Lamplighter.

In the fall of 1978, Papert and several others from DSRE made a series of preparatory visits to Lamplighter to arrange for the introduction of LOGO to the school. The plan was to begin LOGO training for first the faculty and then the students by using the Digital LSI-11 LOGO (in use at the Brookline, Massachusetts, project) and later, bring TI LOGO into the school as it developed.

Shortly after the first visit by Papert, Lamplighter rented the first of two LSI-11's that were to be used in the initial two years of the project. Training sessions helped the initial core of Lamplighter faculty (representing nursery school, second grade, third grade, and fourth grade) become familiar with LOGO. This "Computer Group" then began working with third and fourth grade students. Shortly thereafter, a second LSI-11 was rented, and by the end of the spring term every third and fourth grade student had had at least one hour of LOGO instruction on a computer.

The third and fourth graders considered it a treat to work on the computer—partly because these special computer activities allowed them to miss classes, and partly because they genuinely enjoyed working with LOGO. One student's remark reflects the sentiments of many of these pupils. After he had spent an hour working at figuring line lengths, turn angles, and sections of arcs in order to construct a computer picture of a cat, he thanked me for "getting out of math class."



In the summer of 1979, the Computer Group was expanded, and two workshops were held to refresh the teachers' memories. Subsequently, a 10 day workshop at MIT introduced the teachers to more elaborate LOGO programming and allowed them to participate in discussions on the relationships between learning and LOGO. Then, as the new school year started, the teachers were really surprised to discover how little the fourth graders had forgotten about LOGO. These students generally recalled all of the commands they had learned three months earlier—even though they had had no contact with LOGO in the interim!

Midway through the fall semester of 1979, several early prototypes of TI LOGO were tested at Lamplighter and revised by the MIT LOGO laboratory personnel in consultation with Lamplighter and Texas Instruments. In January 1980, the pace of computing at Lamplighter accelerated as an updated version of TI LOGO was implemented on the TI prototypes. By the end of January, a dozen prototypes were in use at Lamplighter, and a very few students continued to use the LSI-11 LOGO. Most pupils, in fact, switched to the TI prototypes even though that meant relearning much of LOGO.

In the middle of the spring semester, a few more prototypes arrived and all the machines were upgraded to a later version of TI-based LOGO. Before the school year ended, all of the third and fourth graders had had at least one hour on the new machines. One of the rented LSI-11's was then returned (though few noticed its departure). At that time, several fourth graders were writing elaborate programs which made use of recursion to create "movies" or "rainbows" (changing colors), or elaborate scenes. Some students were so taken with LOGO that their parents happily bought them their own computers (at that time, TI LOGO was not yet commercially available); other students became enthralled with their ability to produce perfectly printed letters and numerals on a keyboard and later received typewriters as presents from their parents.

By September 1980, a total of 50 TI LOGO prototypes were in operation at Lamplighter. The version of LOGO on these units was very close to that which TI is now marketing. Then, late in the fall, the second LSI-11 was returned, but its loss went *completely* unnoticed because all of the faculty and student interest was already focused upon the TI LOGO prototypes. Since September, the Computer Group has continued to work individually with third grade students. In addition, the rest of the faculty is being trained in LOGO, and it has been introduced into all of the classes as part of the regular school curriculum.

The teams at each grade level decided the best way to introduce LOGO into their classes and worked out various procedures for that introduction. For example, one teacher developed special simplified LOGO programs for the preschool children which required less typing in order to produce interesting effects. And personifications of LOGO constructs made LOGO easier for first and second graders to understand. Currently, students can be seen at every shared-space LOGO machine during lunch-hour, before school, after school, and whenever other school activities are completed. For the rest of the semester, LOGO will be used in class by the teachers as they feel it is relevant for their lessons and will continue to be available (as are the other learning aids) to students during free periods.

The Lamplighter LOGO project was not intended to be a formal experiment. Since there are no control groups, strong causal claims for LOGO's effects are inappropriate. Several cognitive and psychological assessments, however, were made at the beginning of the project and will be made again at the conclusion of the present school year. And, there already have been some indications of student attitude and behavior change. This is best exemplified by the way in which the pupils express their keen interest in acquiring new LOGO knowledge.

It's always interesting to observe what motivates children to learn. Because LOGO is so extensive, Lamplighter teachers find it impossible to show students all the commands in the initial sessions. As a result, students have taken the discovery of more LOGO commands as a sort of treasure hunt and this new, "unauthorized," LOGO information is disseminated through an "underground network" among the students. During a training session in which teachers were learning to use MAKESHAPE (the LOGO command with which users make their own shapes on a 16 × 16 grid), some students were secretly watching them. Shortly afterward, a hand-copied "underground" LOGO manual with clear and concise directions for the use of MAKESHAPE was found on the floor of a classroom; at the same time, a number of students began using MAKESHAPE.

Students have discovered other information accidentally. One student typed MC instead of MS for MAKESHAPE; this put him in MAKECHARACTER mode. In this mode, LOGO users can modify old characters, or make new characters. The student proudly shouted out his discovery to his classmates, who quickly confirmed his results and spread the news. New information has diffused from grade to grade or class to class or from parent to child in a similar manner.

Sharing among peers is the overwhelming response of Lamplighter students to new LOGO information. Pupils eagerly and proudly explain their accomplishments in LOGO. At first, however, there were a few exceptions. A couple of students were secretive about some LOGO information and effects. One student made the screen's background color black so that no one could read what he typed; another tried to *sell* LOGO programs to his classmates! After they discovered that other students could find different ways to achieve the same effects and were willing to share, they started sharing as well.

In at least one case, LOGO seems to be responsible for a *major* behavioral change. Late last year, a fourth grader who had not been performing well academically, and who had been somewhat disruptive in class, started programming in LOGO. As he played on the computer, his typing became very fast (QWERTY keyboards are quite properly regarded by the Lamplighter children as a stupid arrangement with which they reluctantly work), and his program became sophisticated. He was heard to remark, "I can't believe how fast my fingers are typing." He also could not believe how much fun school had become. Not only did he do well with LOGO, but he also became an attentive, productive student.

Figure 1.

```
TO LAZY8
FORWARD 4
RIGHT 10
TEST HEADING = 0
IFF LAZY8
FORWARD 4
LEFT 10
END
```

TELL TURTLE
LAZY8

Comment:

TEST checks the heading of the Turtle. If it's not 0 (North), the Turtle continues to draw the LAZY8.

After finishing the right-hand circle, the heading becomes 0 and the left-hand circle is drawn.

To really understand why the left-hand circle ever gets completed, you have to know something about microprocessors and stack operations. In keeping with the scope of this section, however, a simple anthropomorpic explanation will have to suffice at this time. Other sections will take an in-depth look at the technical aspect of the language.

Think of the job of drawing the LAZY8 as being given to a group of little workmen inside the computer. The first workman carries out the first four lines then decides he needs a rest before continuing. Notice that in his initial contract TO LAZY8 he has agreed to eventually carry out the FORWARD 4 and LEFT 10 specifications. The work must go on while he rests, so he subcontracts out the next stage to another little man. This workman also carries out the first four lines, then he too decides to rest. So before he gets to the FORWARD 4 LEFT 10 tasks, he decides to subcontract out the balance of the work on the right-hand circle. This process goes on with enough little workmen (36 in this case) until HEADING = 0. At that time, the last little man carries out his FORWARD 4 and LEFT 10 tasks, and gives the job responsibility back to the next-to-last workman who also carries out his remaining FORWARD 4 and LEFT 10 tasks. This reverse process of finishing the last two tasks and relinquishing responsibility goes on until the original contractor finishes his original job with a single FORWARD 4 and LEFT 10, thus completing the left-hand circle in the LAZY8—Ed.

At present, most of the third and fourth graders—and even some of the first and second graders—are writing LOGO programs. And this includes some fairly sophisticated programs which use recursion and the concept of state transparency. A few children even acquired the skill of using *subprocedures*—i.e., breaking a complex program down into its several component parts. This is one of the most important features of procedural languages such as LOGO. Most students had discovered recursive programming, or "cursives" as a few called it. In recursive programs, one of the program lines calls for a new stack to execute the program again. You do this by including the name of the program within the program itself. All the recursive programs written by the students, however, had the recursive step in the last line. [When the recursive step occurs in the last line before END, the procedure is said to have "tail-end recursion." For an example of somewhat more sophisticated usage, see the LAZY8 procedure in Figure 1—Ed.]

A number of programs produced exciting video scenes. In EXPLODE, 32 differently colored balls splay out from the center of the screen before repeating the entire procedure. One third grader saw how he could place a program which printed a message inside EXPLODE, and thus combined recursion and subprocedures. RAINBOW had one or more sprites continuously change colors for an attractive visual effect. There were also programs which had the TV monitor take on a series of sixteen colors, and programs which changed the background of the screen to black and created unusual perceptual illusions by shooting light-colored shapes across the screen. Some even had jets, rockets, or airplanes spouting fires from their engines.

Other children wrote programs which put shapes together to create scenes, such as a home with a car driving down the street in front of the home. Most students had written utilitarian programs like VANISH (Figure 2) which caused the sprites to move off screen, take on the clear color, carry an empty shape, and which caused all the printing to be cleared from the screen.

Figure 2.

```
TO VANISH
TELL :ALL
CARRY 0
SETCOLOR 0
SETSPEED 0
SETHEADING 0
END
```

After spring break, several things happened which caused a quantum leap in the computer work of the students. First, the children were shown how to save their programs and shapes on cassette tape. Until then, the students had to write in their computer notebooks anything they wanted to save. That meant that any elaborate shape had to be reproduced on a grid in an arduous manner, and long programs or complex programs required a very long time typing. (Remember these children are elementary pupils with little typing experience before computers!).

Students had not used much of their work as foundations for future work simply because loading the old material took so much of their time. Now, with the recorders, they could use and improve each session's programs just by taping and playing back a cassette. Also, they could design and SAVE complex shapes instead of seeing them lost when the computers were shut off.

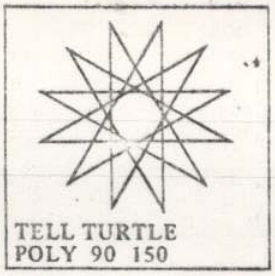
The children were also shown the TELL TURTLE mode. This opened up all of the turtle geometry features of LOGO.

(Turtle geometry is such a powerful idea that some Pascal systems have adopted it.) This newly acquired mode, coupled with the previously learned SPRITE MODE, allowed the students to produce many interesting programs and visual effects. As a result of these new developments, many of the students soon exhibited a feeling of mastery over the computers.

In the final eight weeks of school there was an exponential explosion in the complexity of the students' programs and in their ease with the machines. They quickly learned to use variables as inputs, and consequently "discovered" the famous turtle geometry POLYgon program which can generate any regular polygon. (See Figure 3.) Then one student found that changing the angle of the turn on each recursion could produce beautiful patterns—including a striking nested curl in a star pattern. Many students now began putting programs together in subordinate and superordinate structures. Programs contained the unique LOGO controls of TEST, IFT, and IFF, as well as the conditionals IF-THEN-ELSE, plus BOTH and EITHER for conjunctive and disjunctive branching. One of the third graders wrote a CAI (Computer-Assisted Instruction) program to quiz his first grade friends on addition facts using these control commands! He then added visual displays of the addends, and encouraging remarks when a student made a mistake, or a colorful scene as a reward for the correct answer.

Figure 3

```
TO POLY :DISTANCE :ANGLE
HIDETURTLE
FORWARD :DISTANCE
RIGHT :ANGLE
POLY :DISTANCE :ANGLE
END
```



Using combinations of several user-drawn shapes, students began constructing very elaborate composite pictures. One third grade student also discovered how to change the characters associated with each console key [by redesigning the characters on a grid "tile" with the MAKECHAR primitive—Ed.], and decided to tease the teacher. She replaced the 3 with a 2, and then called a teacher for a demonstration. While instructing the computer to print 3 + 3 (which now looked like a request for the sum of 2 + 2), she remarked to the teacher: "Look how dumb this computer is. It doesn't know 2 + 2."

The activity among the third grade students was exciting to witness. One began programming dramas in which text was printed at the bottom of the screen while the story was enacted in SPRITE and TELL TURTLE modes at the top of the screen. One other third grader was so intrigued by the space shuttle's landing that on the same afternoon of the landing, he began working on a shuttle program. First, he used MAKESHAPE to construct a faithful replica of the shuttle, complete with USA monogram, black-and-white coloring, and auxiliary rocket engines. Then he worked for part of the afternoon and a little of the next morning to write and debug his programs. His final superprocedure launched the shuttle with flames shooting from the engines, jettisoned the auxiliary tanks, orbited the shuttle among planets in outer space, returned the shuttle to a dry lakebed runway, taxied it to the end of the runway, and stopped it for a perfect landing. His programs are shown here in Figure 4

Figure 4

Note:
BG = BACKGROUND
FD = FORWARD
SC = SETCOLOR
SH = SETHEADING
SS = SETSPEED

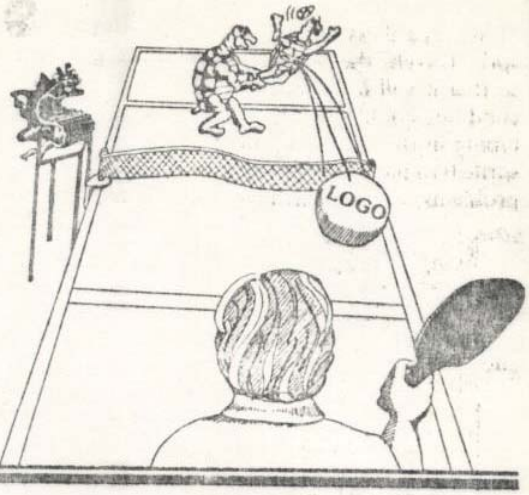
```
TO FIRE
TELL 4
WAIT 30
SC 0
WAIT 15
SC 6
FIRE
END

TO TANK
TELL 1 CARRY 20 SC 15 HOME
SH 90 FD 16
TELL 2 CARRY 21 SC 15 HOME
TELL 3 CARRY 22 SC 15 HOME
SH 270 FD 16
TELL 4 CARRY 23 SC 6 HOME
SH 270 FD 32
END

TO SHUTTLE
TANK
PLANE
FIRE
END

TO PLANE
TELL 5 CARRY 24 SC 15 HOME
SH 0 FD 16 SH 90 FD 16
TELL 6 CARRY 25 SC 15 HOME
SH 0 FD 16
TELL 7 CARRY 26 SC 15 HOME
SH 0 FD 16 SH 270 FD 16
TELL 8 CARRY 27 SC 15 HOME
SH 0 FD 8 SH 270 FD 16
TELL 9 CARRY 4 SC 11 HOME
SH 180 FD 47 SH 90 FD 12
TELL 10 CARRY 4 SC 8 HOME
SH 0 FD 45 SH 270 FD 23
TELL [1 2 3 4 5 6 7 8]
SH 90 SS 20
TELL BG SC 1
END
```

[Note: Listings of TI LOGO procedures are just that—listings of *procedures*. There's no way to print out a transcription of the data needed to MAKESHAPE and MAKECHAR as can be done with the HEX Codes in TI BASIC and Extended BASIC. The only way to show the graphics that a program contains is to *show it as drawn* on a series of "tiles" on the grids that appear on screen when the shapes and characters are first designed. This is similar to CHARDEF routine in *Programming Aids 1*. The listing of the Space Shuttle program was included (without the tiles) in this article to demonstrate the simplicity of the language structure.—Ed.]

The gains made by the Lamplighter children with LOGO have indeed been impressive. They confirm Papert's dictum [*Mindstorms*, Seymour Papert, Basic Books 1980] that children should program computers and not vice-versa. It's obvious that LOGO has indeed furthered Lamplighter's goal of igniting the imaginations and intellects of its children. But more importantly, LOGO has the potential to fire up imaginations *everywhere*.

Who is LOGO For?

Its not just for Turtles anymore . . .



Recently the question of LOGO's relevance for children and its relevance for adults has been stated as an implicit either/or issue. That the issue ever arose means that people (including me) who write about LOGO have not done their jobs as fully as they should. Perhaps the notion that LOGO was just for children developed because of the total attention children invest in LOGO. The position that LOGO is too complex for children may have arisen because published programs seem magic unless one actively explores them (including seeing what happens when the programs are changed). Presenting a program as a *fait accompli* to be copied, run, stored, and used like any other software is contrary to the philosophy of education behind LOGO.

LOGO is for humans. When Papert asked me if I felt comfortable with my LOGO, I said that LOGO is like a hologram—when you grasp just the smallest part of it, you have a small, but complete picture; and later as your understanding grows, you still have a complete picture, albeit larger. From that perspective, people can always learn more from LOGO and do more with LOGO even though they are able to use LOGO after the briefest of introductions. This feature of LOGO is what Papert alludes to in his slogan, "Low threshold, no ceiling."

The LOGO slogan invites empirical verification. In my self-observations and studies of other adults, I have noticed that there are common, identifiable LOGO-developmental stages. Among these are the discovery of heuristics (i.e., powerful ideas), improved understanding of numbers, appreciation of angles and heading, and awareness of states and state independence. Probably the greatest gain people share in working with LOGO is the realization that one can find out on the computer, rather than ignoring the question or looking the answer up somewhere. This is so obvious that it might appear trivial; it is not. All learning theorists agree that active learning is preferable to passive learning. This presents a dilemma for those writing about LOGO: How do you capture the open activity of a LOGO learning enterprise in a closed article?

The purpose of *this* article, however, is to reflect the development of a LOGO game, and in that development show how an apparently complex program is child's play, even for adults. At the same time, I hope that the development will point to variations and will entice you into active exploration. The program was initiated by a student in a course I taught.

The program was supposed to be a "Pong" type game. As you follow its growth, find the point, if there is one, where the program stops being a children's program.

The game begins not as a program, but simply a collection of conditions.

```
TELL 0
CARRY :BALL
SETCOLOR :BLUE
SETHEADING 90
SETSPEED 15
HOME
```

These commands set a ball speeding left-to-right across the middle of the screen.

The idea grows into a program as the ball is set to "bouncing" off left and right boundaries. This is accomplished any of several ways:

```
TO BOUNCE1
TELL 0
TEST XCOR > 85
IFT RIGHT 180
TEST XCOR < -85
IFT RIGHT 180
BOUNCE1
END
```

But BOUNCE1 sometimes doesn't work—occasionally the sprite is "caught" at one end or the other. What happens is that the sprite slips past one of the boundaries (e.g., the computer is at line 2 of the program as the sprite moves left through X coordinate equal to −85); by the time the computer reaches line 4, the sprite is well left of X coordinate −85. Then the computer turns the sprite right 180 (a right 180 functions equivalent to a left 180). Before the sprite can move beyond the −85 X coordinate, the computer checks line 4 again, turns the sprite 180 and sends it still further to the left. Of course, when the computer reaches line 4 a third time,the sprite is still left of −85; the poor sprite is stuck beyond the left-hand boundary! This bug could be eliminated with a second type of BOUNCE program:

```
TO BOUNCE2
TELL 0
TEST XCOR > 85
IFT SETHEADING 270
TEST XCOR < -85
IFT SETHEADING 90
BOUNCE2
END
```

that the CHECK program is invoked only if the ball is beyond XCOR 85. Therefore, part of the scoring and noises can be controlled after line 3 of BOUNCE2 by rewriting the CHECK program:

```
TO CHECK
TELL 0
TEST PADDLETOUCH
IFT CALL :PLS + 1 'PLS : IN -
; CREASE ... PLAYER'S SCORE
IFT NOISE ; NOISE FOR THE
; PLAYER'S POINT
IFF SETHEADING 270
IFT CALL :CPS + 1 'CPS : ELSE
; INCREASE COMPUTER'S SCORE
IFF BEEP WAIT 10 NOBEEP ; A
: SHORT BEEP FOR COMPUTER'S
; POINT
TYPE [YOUR SCORE IS ] PC 32
TYPE :PLS PC 32
TYPE [THE COMPUTER'S SCORE IS
] PC 32
PRINT :CPS
WAIT 90 CS ; ADDED TO PREVENT
; EXTRA SCORING ON EACH SERVE
END
```

```
TO NOISE
REPEAT 5 [BEEP WAIT 3 NOBEEP W
AIT 3 ]
END
```

It is necessary to set up an initial value for both the computer's score and the player's score as was done with :Y. Since this is done just once, it belongs in SETUP. [The initial score is 0 to 0—as in the proverbial "soothsayer's" prediction or score before it begins. . . .] So SETUP is revised:

```
TO SETUP
TELL 0
CARRY :BALL
SETCOLOR :BLUE
HOME
SETHEADING 90
SETSPEED 15
TELL [1 2 ]
CARRY :BOX
SETCOLOR :BLACK
SETHEADING 0
SXY 100 0
TELL 2
SY 16
CALL 16 'Y
CALL 0 'PLS
CALL 0 'CPS
END
```

This game, like most LOGO projects, is open-ended. It could be altered so that a winner is named at a score of 21, revised for two players, changed to use joysticks or changed so that the ball has topspin. With each addition, it is necessary to make sure that the initial conditions are established only once, that procedures to be repeated are placed inside a recursive program, and that there are no Recursion Interface Bugs.

```
TO BOUNCE2
TELL 0
TEST XCOR > 85
IFT CHECK
TEST XCOR < -85
IFT SETHEADING 70
TEST YCOR > 90
IFT SETHEADING 135
TEST YCOR < -85
IFT SETHEADING 45
END
```

LOGO's POWERFUL SURPRISES!

PART 1: Language Structure and Syntax

LOGO was developed by Seymour Papert and his associates at the MIT Artificial Intelligence Laboratory in order to study the way people might learn in a computer-rich environment. It was designed to be a language so simple to use that a person could manipulate objects or concepts by just thinking about what he or she wanted to accomplish, and not have to worry about programming. Such a language might stimulate a person to explore, to learn, and to grow.

The idea was to provide certain *primitive* commands and operations that could be combined to form more *complex* commands and operations. These more complex ones could then be used exactly like the primitive ones. Thus it would be possible to construct a single command to accomplish anything that could be accomplished using the primitive concepts. Additionally, *recursion*—whereby a command could call and activate itself—was allowed.

LOGO is a relative of LISP, the list processing language used in artifical intelligence. LOGO and LISP share the capability of manipulating numbers, words (character strings without a space), and lists. A *list* is a recursively defined object: It is an ordered set of objects, each of which may be a number, a word, or a previously defined list. In LOGO, a *procedure* is represented by a list; there are commands to access a list that represents any procedure, and to define *new* procedures from lists which might be the result of some manipulation. Furthermore, a procedure may have inputs and may have an output, and is activated by specifying its name (a word) followed by its inputs (which may be numbers, words, or lists). Defined procedures as well as primitive commands and operations all have exactly the *same* syntax. This is why LOGO is so simple to use. Its power comes from its list processing capabilities.

I hope that the description given so far has made it apparent that LOGO is *not* just for children. Although LOGO can be used in elementary ways, it is much more than FORWARD 20 RIGHT 90. LOGO is a language for all people who want to learn and expand their capacities.

The LOGO Turtle

The first experiments with LOGO were with junior high school students who could appreciate manipulation of words. Then a Turtle was created whose movements could be understood by very young people.

The Turtle was originally a robot that could be commanded to move about the floor. It had a pen which could be either up or down. In an experiment at the University of Pittsburgh Learning Center several years ago, one young person used LOGO to command the floor turtle to draw an alphabet of large letters. He also taught it to act like an airplane, and "fly" between cities on a large map. The plane had the possibility of going out of control, with the turtle going into a spiral and spinning on the floor. The turtle is now usually a small triangle on a terminal screen, but it can still do such things, albeit on a smaller scale.

At the youngest levels, LOGO is being used to teach a feeling for distances and angles. At levels through college it is being used to advance a new subject in mathematics called "Turtle Geometry." Some interesting theoretical results have come about. (A wealth of examples and exercises is contained in *Turtle Geometry* by Abelson and diSessa, where procedures are expressed in a language almost exactly the same as LOGO.) Recursive designs such as snowflake curves, space filling curves and trees are applications of LOGO's power.

TI LOGO

TI LOGO is marketed as a language for children, and it was a pleasant surprise to discover that TI LOGO has all of the list processing capabilities built into it. All the recursive designs presented in *Turtle Geometry* can be drawn. (The TI Turtle is, however, limited to 192 different

8×8 pixel character positions. Thus, if a figure is very dense, it can't be very large.)

The documentation that comes with TI LOGO doesn't make it easy to discover LOGO's power. Many of the commands needed for manipulating all but the simplest lists are not documented.

At this point, it may be helpful to briefly describe just what is available to a person who sits down to use TI LOGO. The TI Turtle is an object that lives on a coordinate screen with horizontal coordinates from −119 to +120 and vertical coordinates from −46 to +97. The bottom six lines of the screen are used for text. The turtle can be assigned a position, and "knows" where it is. It can be assigned a heading (from 0 to 360 as the points of a compass) and knows its heading. Its heading can be changed by a given angle, and it can be moved a given amount either in the direction of or opposite to the direction of its heading. It can make a dot at any position. The pen can be down, up, or in "reverse" modes, and it can draw in any of 15 colors.

Unique to the TI version of LOGO are *sprites*—objects familiar to those with TI Extended BASIC. There are 32 sprites (numbered 0 to 31) with each assigned to a 16×16 pixel shape. Users may design and store 26 of these and can direct any collection of sprites to assume simultaneously an attribute such as shape, color, position, heading, speed, or velocity. The commands which control the turtle act similarly on the sprites. Motion is controlled by assigning a speed (in the current direction) or a velocity (horizontal and vertical components). Not only can attributes be assigned, but they can also be obtained as the output of operations because a sprite always knows its own number, shape number, color number, position (on the full screen), heading, speed, and velocity.

Papert has described *Velocity Turtles* (which can have velocities) and *Acceleration Turtles* (whose velocities can be incremented). Sprites can be both. Using sprites we can even simulate Papert's "Dynaturtle"—an acceleration turtle which does not change direction when it is rotated, but changes velocity only by accelerating in the direction it is facing, thus obeying Newton's laws of motion. A dynaturtle therefore behaves like the ship in the popular *Asteroids* arcade game. The example procedures that follow this article will demonstrate a dynaturtle which can have the force of its "thruster" changed, and which can simulate an environment with friction.

TI LOGO also has 256 tiles (numbered 0 to 255) that can be given arbitrary 8×8 pixel designs. We can assign tiles foreground and background colors and position them anywhere on the 24×32 character screen or on the current print line. Console characters are tiles, the number of each tile being the ASCII code of the character. (Note: The Turtle records its trace using tiles, so simultaneous use of the Turtle and nonprinting characters is limited.)

Numbers, Words, and Lists

A *number* in TI LOGO is an integer from −32,768 to 32,767. Numbers can be added, subtracted, multiplied and divided (integer quotient), calculations being modulo 32,768. The restriction to integer arithmetic is a definite limitation, but the limitation is not serious for most applications.

A *word* is a character string without a space. A feature of LOGO distinguishing it from other programming languages such as BASIC or Pascal is the capability of using a word *simultaneously* as (1) the name of a command or procedure, (2) a variable, and (3) data. For example, if the word X is to be used as the name of an action, X itself is used. When an object has been assigned to X, the object is denoted :X. The word X as data is denoted "X. Suppose that X has not been defined as an action and has not been assigned a value. LOGO will respond to X with TELL ME HOW TO X, to :X with :X HAS NO VALUE, and to "X with TELL ME WHAT TO DO WITH X.

A word can be assigned any kind of data—i.e., a number, word, or list as a value. This also distinguishes LOGO from BASIC or Pascal where the data type of a variable must be specified in advance. As a bizarre example, note that MAKE "MAKE "MAKE and MAKE "MAKE [MAKE] assign to MAKE first the word MAKE and then the list whose single member is the word MAKE.

A *list* is the most powerful data object in TI LOGO and is denoted by a left bracket followed by its members, then a right bracket. Examples of lists are [ ], the null list; [HOW NOW BROWN COW], a list of words; and [REPEAT 4 [FORWARD 20 RIGHT 90]], a list whose members are a word, a number, and another list.

Data Manipulation in LOGO

Commands which are powerful in manipulating data include the following: FIRST(F), LAST, BUTFIRST(BF), BUTLAST(BL), SENTENCE(SE), FPUT, LPUT, NUMBER?, WORD?, THING?, THING, WORD, MAKE, RUN, TEXT, DEFINE. The last three are used to execute a list of commands, to access the list which defines a procedure and to define a procedure represented by a given list. These are powerful commands, but to be able to make use of them it is necessary to be able to construct lists whose members themselves are lists. The following key (undocumented) commands, FPUT and LPUT, are helpful here:

FPUT *object list*—outputs a list whose first member is object, and whose following members are the members of list.

LPUT *object list*—outputs a list whose last member is object and whose members all but the last are the members of list.

If *object* is a word or a number, the results of these commands are the same as SENTENCE *object list* and SENTENCE *list object*, respectively. But if *object* is a list, FPUT *object list* adds *object* to the beginning of list while SENTENCE *object list* adds the *members* of *object* to the beginning of *list*. This is a crucial difference, making possible the construction of arbitrarily complicated lists. The other commands in the above list which are undocumented are as follows:

NUMBER? *object*—returns TRUE if object is a number, and FALSE otherwise.

WORD? *object*—returns TRUE if object is a word, and FALSE otherwise.

THING? "*name*—returns TRUE if name has been assigned a value, and FALSE otherwise.

THING "*name*—returns the object which has been assigned to name, if name has a value.

WORD *word1 word2*—returns the word formed by concatenating word1 and word2. (Compare with SENTENCE, below.)

SENTENCE *wordorlist1 wordorlist2*—(a documented command), returns a list determined by the inputs. If an input is a word, that word is put in the list. If an input is a list, its members are included in the list.

Some of the undocumented commands were found by accident; others by studying the documentation for MIT LOGO. Still others were known to Jim Muller, president of the Young Peoples' LOGO Association (YPLA). We encourage readers to share other discoveries with us.

A Calculating Example

As a simple example, consider the problem of teaching LOGO to act like a calculator. If one enters 2 + 3, the response is TELL ME WHAT TO DO WITH 2 + 3. Here, desired output is 5, which is the result of executing PRINT 2 + 3. The problem is solved by using SENTENCE to form the list [PRINT 2 + 3] and then using RUN to execute the list. A solution is the following:

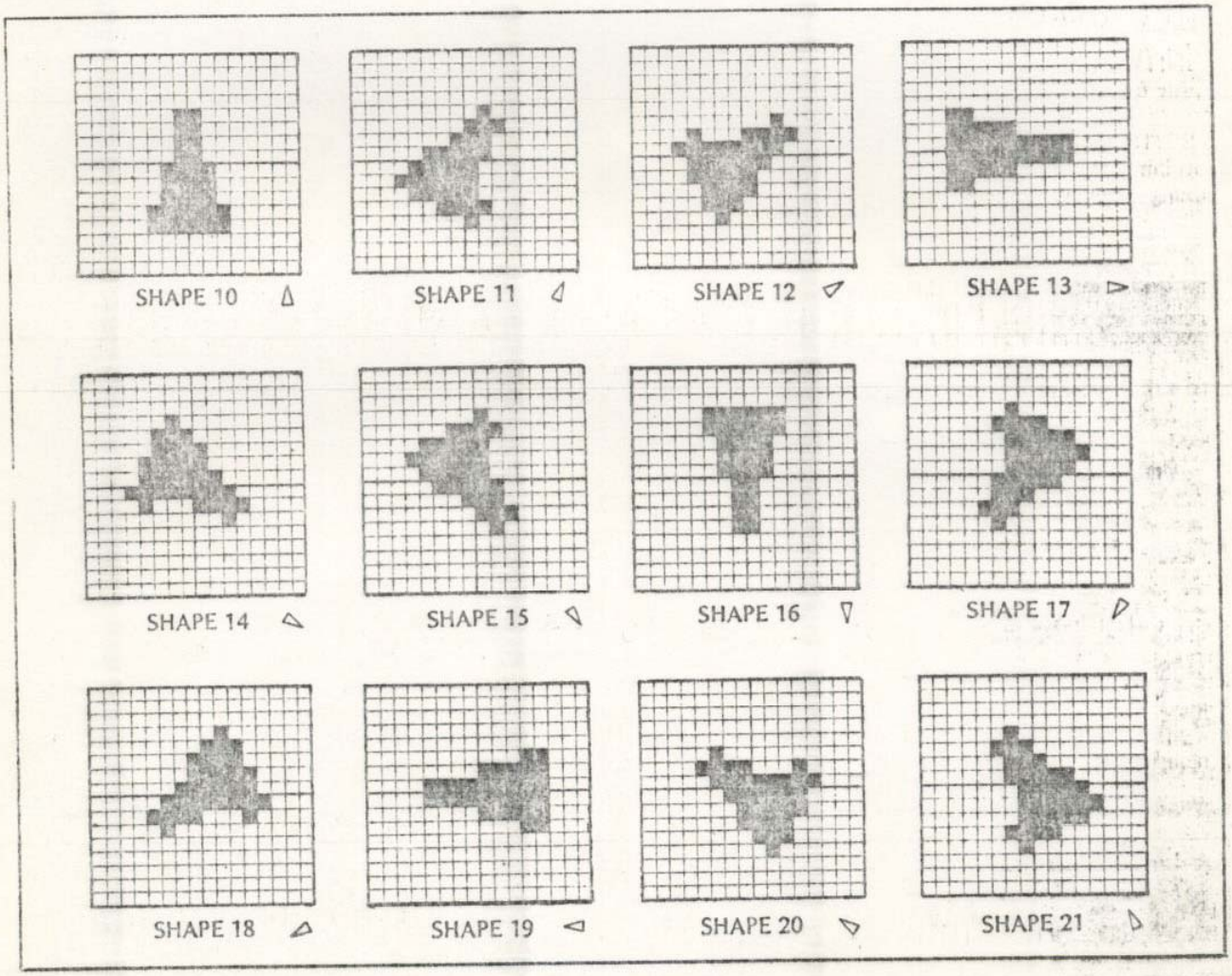
```
TO CALCULATE
MAKE "Y READLINE
TEST :Y = []
IFF RUN SENTENCE "PRINT :Y CAL
CULATE
END
```

After you enter CALCULATE, the computer accepts arithmetic expressions and prints out the resulting value until just ENTER is pressed. The recursion then "unwinds," and the procedure stops.

The power of a list processing language such as TI LOGO becomes apparent the more you use it. Yet for learning, all of these advanced capabilities *don't* have to be utilized. This is what makes the language so versatile—its built-in power that is accessible on demand. And it is this versatility that allows teachers to tailor LOGO for special applications, and reassures all students that with LOGO there is always more to learn.

PART 2: Constructing a DYNATURTLE

The instructions for using the dynaturtle are obtained by typing HELP. The dynaturtle itself is activated by typing DYNATURTLE. The procedure starts out drawing a circle and displaying a white dynaturtle. Touching the E key



causes a "thruster" to impart motion to the dynaturtle with speed 3. Each touch of the E key adds a velocity with magnitude 3 to the dynaturtle. Touching S or D makes the dynaturtle face 30 degrees left or 30 degrees right from its former heading. Velocities add like vectors. If the dynaturtle is not facing in the direction of its motion, the force of the thruster will cause it to head in a direction intermediate between its heading and direction, exactly as if it were a rocket in space obeying Newton's laws.

Touching F will turn friction on. In this state, the dynaturtle will be sluggish and come quickly to a stop after each kick. It will therefore be necessary to increase the force of the thruster. To do this, touch K. You can then enter a number, say 10 or 20, and touch ENTER. The dynaturtle will now be given an increase in velocity with magnitude 10 or 20 with each touch of E. Touching F again will turn friction off. You will find the dynaturtle now very difficult to control. Touch K again and readjust the thrust.

When friction is off, the dynaturtle is seen to act just like the ship in the *Asteroids* arcade game. When friction is on, it behaves as if it were riding on a rough surface—appearing to skid as you direct it around the circle.

Description of Procedures

DYNATURTLE activates the procedures INITIALIZE, SETDYNATURTLE and CONTROL.

INITIALIZE draws a circle and initializes the thruster (sprite 0).

SETDYNATURTLE positions the dynaturtle and gives it its initial shape (shape 10). The secret of the dynaturtle's turning capability is that the twelve shapes (shape 10 through 21) contain designs for the dynaturtle, each rotated 30 degrees from the preceding.

CONTROL is the main loop. Friction is always checked to see if it is on. If it is on, CHECKFRICTION decreases the dynaturtle's speed. If one of the control keys is pressed, the action is taken and control branches to label A. This procedure keeps running until Q is touched.

KICK reads the velocity of sprite 0, which is always kept heading in the direction the dynaturtle is facing. This velocity is then added to the velocity of sprite 1, which carries the shape of the dynaturtle.

ROTRIGHT adds 30 degrees to H, which maintains the heading of the dynaturtle and causes sprite 1 to carry the shape with next highest number, unless that number is larger than 21. If sprite 1 is carrying shape 21, it assumes shape 10. In this way, the dynaturtle appears to be rotating to the right by 30 degrees.

ROTLEFT is similar to ROTRIGHT but gives the effect of rotating the dynaturtle to the left.

SETFRICTION simply makes the value of the word FRICTION? true if it is false, and false if it is true.

SETKICK gets a number from the console and assigns it as the speed for sprite 0. The velocity for sprite 0 (x- and y- coordinates) is used to impart an acceleration to sprite 1. Note the command SS FIRST READLINE. The primitive READLINE outputs a list, and SS requires a number for input. The desired number is the first (only) member of the list entered.

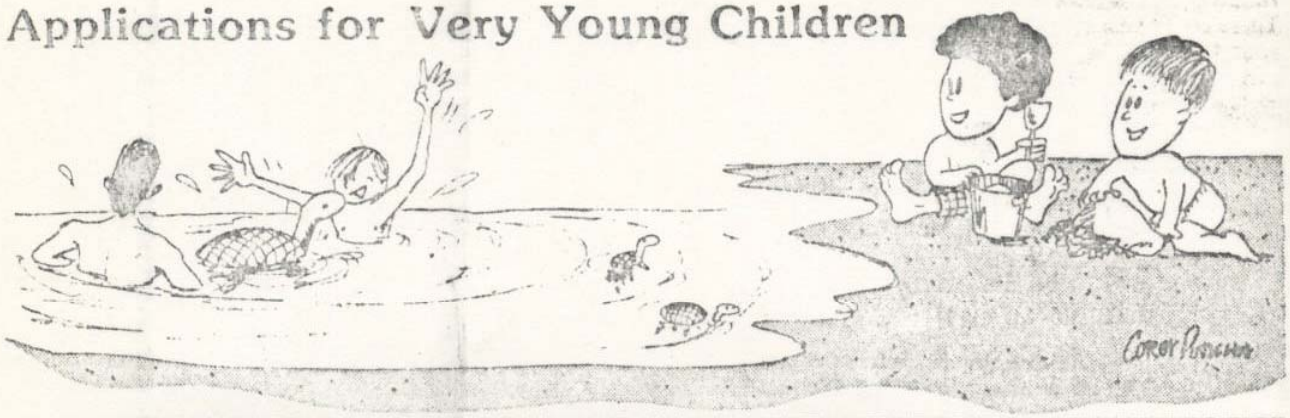
```
TO DYNATURTLE
INITIALIZE
SETDYNATURTLE
CONTROL
END

TO KICK
TELL 0
SH :H
MAKE "DVX XVEL MAKE "DVY YVEL
TELL 1
SV XVEL + :DVX YVEL + :DVY
END

TO CONTROL
A:
CHECKFRICTION
TEST RC?
IFF GO "A
MAKE "X RC
IF :X = "E THEN KICK
IF :X = "S THEN ROTLEFT
IF :X = "D THEN ROTRIGHT
IF :X = "F THEN SETFRICTION
IF :X = "K THEN SETKICK
IF :X = "Q THEN STOP
GO "A
END

TO CHECKFRICTION
IF :FRICTION? THEN TELL 1 IF S
PEED > 0 THEN SS SPEED - 1
END

TO ROTLEFT
TELL 1
MAKE "H :H - 30
IF SHAPE = 10 THEN CARRY 21 EL
SE CARRY SHAPE - 1
END
```

```
TO HELP
NOTURTLE CS
PRINT [THE DYNATURTLE IS AN OB
JECT ]
PRINT [WHICH OBEYS NEWTON'S LA
WS ]
PRINT [OF MOTION. ]
PRINT [ ]
PRINT [IT LIVES ON A SURFACE W
HICH ]
PRINT [CAN BE SMOOTH OR ROUGH.
]
PRINT [ ]
PRINT [CONTROLS: ( TOUCH KEYS
)]
PRINT [ ]
PRINT [E: GET KICK FROM THRUST
ER ]
PRINT [S: TURN LEFT 30 DEG ]
PRINT [D: TURN RIGHT 30 DEG ]
PRINT [F: TURN FRICTION ON / O
FF ]
PRINT [K: SET THRUSTER KICK ]
PRINT [Q: QUIT ]
PRINT [ ]
PRINT [TRY TO GO AROUND CIRCLE
]
PRINT [ ]
PRINT [TYPE "DYNATURTLE" ]
PRINT [ ]
END

TO CIRCLE :SIDE
REPEAT 36 [FD :SIDE LT 10 ]
END
```

```
TO ROTRIGHT
TELL 1
MAKE "H :H + 30
IF SHAPE = 21 THEN CARRY 10 EL
SE CARRY SHAPE + 1
END

TO SETFRICTION
TEST :FRICTION?
IFT MAKE "FRICTION? "FALSE PRI
NT [FRICTION OFF ]
IFF MAKE "FRICTION? "TRUE PRIN
T [FRICTION ON ]
END

TO SETKICK
TYPE [FORCE OF KICK? ]
TELL 0
SS FIRST READLINE
END

TO SETDYNATURTLE
TELL 1
SX 50 SY 0
SS 0 SH 0 CARRY 10
SC :WHITE
END

TO INITIALIZE
MAKE "H 0 ; HEADING OF DYNAT
TELL TURTLE ; DRAW CIRCLE TO G
O AROUND
CS HT
SX 50 CIRCLE 8
TELL 0 ; INITIALIZE THRUSTER
SS 3
MAKE "FRICTION? "FALSE
END
```

EXT-E-N-D-I-N-G LOGO

Applications for Very Young Children



Seymour Papert designed LOGO to have a low threshold so that even young children could benefit from it. Unfortunately, the present technical requirement that LOGO input and output be *text-bound* limits LOGO to children who can read and type.

It is apparent that learning in a LOGO environment offers greater potential for pre-verbal children than for verbal humans. This is simply because there is so much more for them to learn. But it is equally apparent that the reading/typing prerequisite is an artificial barrier to that same environment. The ultimate solution—a computer which can comprehend *and* generate speech—is not yet available. (Programs such as TI text-to-speech, however, can do a reasonable job of talking.) Still, there are ways LOGO can be adapted to children who are only able to recognize alphabetic characters or typewriter keyboard symbols.

Even before LOGO was implemented on the DEC LSI/11 or the TI-99/4, people in the MIT LOGO lab worked on simplified LOGO systems. One approach yielded a special LOGO input device which translated symbol cards[1] inserted into slots through a light scanner into ASCII code. Although prototypes of the "slot machine" card reader worked well enough, the idea was never developed commercially. A second approach was followed by Bob Lawler, a graduate student at the time, who wanted his two children to be able to use LOGO. He wrote a number of excellent LOGO programs which allowed *very* young children to draw with the turtle, to play shoot-out games with the turtle, or to design elaborate turtle pictures. Lawler's programs were written for a large, mainframe computer version of LOGO, but his ideas are compatible with TI LOGO. In fact because of the excellent color graphics of TI LOGO, his ideas may involve children more effectively when set up on the TI-99/4A. The essence of his programs was to simplify access to turtle geometry by simplifying and combining commands. One simplification allows pupils to move the turtle forward or backward a fixed amount, or to turn the turtle left or right a fixed amount by pressing any of the four "arrow" keys:

```
TO DRAW
TELL TURTLE
SIMPLIFY
END
```

[1] The cards carry labels like ↱ for RIGHT 90, ↓ for FORWARD 10, as well as symbols for recursion and sub-procedure calls.

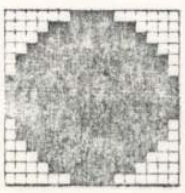
```
TO SIMPLIFY
CALL RC "ANSWER
IF :ANSWER = "D RIGHT 30
IF :ANSWER = "E FORWARD 10
IF :ANSWER = "X BACK 10
IF :ANSWER = "S LEFT 30
SIMPLIFY
END
```

With slightly more sophisticated children, or as children work with DRAW, we can add other commands by merely inserting them into SIMPLIFY. For example:

```
IF :ANSWER = "C CLEARSCREEN
IF :ANSWER = "Q STOP
IF :ANSWER = "0 PENERASE
IF :ANSWER = "1 PENDOWN SETCOLOR 1
IF :ANSWER = "4 PENDOWN SETCLOR 4
IF :ANSWER = "U PENUP
```

Then, as students master the fuller set of DRAW commands and learn the idiosyncracies of QWERTY typewriter code, they can be introduced to TELL TURTLE without using DRAW any further.

Coleta Lewis, a teacher at the Lamplighter school in Dallas, adapted sprites for use by nursery school children. (The Lamplighter school pioneered the use of TI LOGO with children; see "The Lamplighter LOGO Project.") Two "games" her children played allowed them either to move a garage around the screen, move a car around the screen, and vary the colors of each separately, or to construct a face and then color in the parts of the face. Programming sprites for very young children is not much more difficult than DRAW. As one example of a sprite game for youngsters, consider a game of blocks. It is fairly easy to create a universe of blocks with simplified sprite commands. First, it is necessary to make up some "blocks" using MAKESHAPE. A circle (shape 4) and a square (shape 5) already exist. A good set of blocks ought to have a triangle.



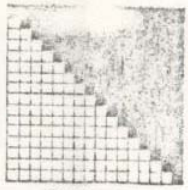
SHAPE 4

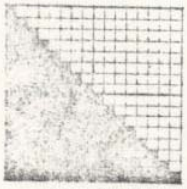
SHAPE 5

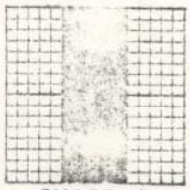
It would, however, be better to build triangles with four different orientations because the shapes of sprites cannot be rotated.
These could be assigned shape numbers 6, 7, 8, and 9. A good block set should also have a rectangle. To show the two orientations, shape numbers 10 and 11 could be designed as shown.

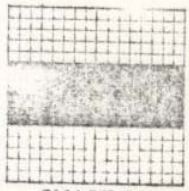

SHAPE 6


SHAPE 7


SHAPE 8


SHAPE 9


SHAPE 10

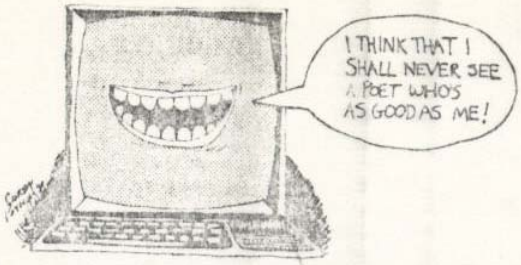

SHAPE 11

Other shapes could be easily added. Children should be able to color the blocks, move the blocks around, change the shape of any of the blocks, and bring more blocks onto the screen. In addition, when a child begins working with one of the blocks, he or she should be able to identify which one it is. (One way this identification can be aided is to have them briefly flash colors.) The following programs implement these ideas:

```
TO BLOCKS
; THIS PROCEDURE CLEANS UP THE
  SCREEN AND GETS BUILD STARTED
TELL :ALL
SETCOLOR 0 CARRY 4
HOME
CLEARSCREEN
TELL 0
BUILD
END

TO BUILD
;
; THIS PROCEDURE TAKES A
; SINGLE KEY TYPED AS INPUT
; AND USES IT TO CHANGE
; SPRITE SHAPE, COLOR,
; NUMBER AND LOCATION
;
CALL RC "ANSWER
IF :ANSWER = "N NEXT
IF :ANSWER = "B BACKONE
IF :ANSWER = "C SETCOLOR COLOR
 + 1
IF :ANSWER = "E SETHEADING 0 F
ORWARD 10 ; MOVE THE SPRITE
;
; TEN STEPS UP
;
IF :ANSWER = "N SETHEADING 180
 FORWARD 10 ; MOVE THE SPRITE
;
; TEN STEPS DOWN
;
IF :ANSWER = "S SETHEADING 270
 FORWARD 10 ; MOVE THE SPRITE
;
; TEN STEPS LEFT
;
IF :ANSWER = "D SETHEADING 90
FORWARD 10 ; MOVE THE SPRITE
;
; TEN STEPS RIGHT
;
IF :ANSWER = "F FORM
IF :ANSWER = "Q STOP ; STOPS
;
; THE PROGRAM
;
BUILD
END
```

```
TO NEXT
;
; THIS PROCEDURE RECRUITS
; THE NEXT SPRITE
;
TEST YOURNUMBER = 31
IFT PRINT [THERE ARE NO MORE S
PRITES LEFT.] STOP
IFF TELL ( YOURNUMBER + 1 ) [
SCH CARRY 4 STOP
END

TO BACKONE
;
; THIS PROCEDURE MOVES BACK
; TO THE   EVIOUSLY
; RECRUITED SPRITE
;
TEST YOURNUMBER = 0
IFT PRINT [THIS IS THE FIRST S
PRITE.] STOP
IFF TELL ( YOURNUMBER - 1 ) FL
ASH STOP
END

TO FORM
;
; THIS PROCEDURE ALLOWS
; SHAPES TO BE CHANGED
;
TEST SHAPE = 11 ; 11 IS THE
;
; THE LAST BLOCK AND 4 THE
; F   IN THE BLOCK SET
; MADE SO FAR - - CHANGING
; THIS NUMBER WOULD MAKE
; A LARGER SET
;
IFT CARRY 4 ; START OFF WITH
;
; THE FIRST BLOCK IN THE
; SET
;
IFF CARRY ( SHAPE + 1 )
;
; CHANGE THE SHAPE TO THE
; SHAPE IN THE SET
;
END
```

```
TO FLASH
;
; THIS PROCEDURE INVOKES
; EITHER FLASH0 OR FLASH1 SO
; THAT THE SPRITE WILL FLASH
; BRIEFLY TO IDENTIFY THE
; CURRENT SPRITE
;
CALL COLOR "START
IF :START = 0 FLASH0 STOP
FLASH1 :START STOP ; CAUSES
;
; THE SPRITE TO FLASH AND
; FINISH FLASHING WITH ITS
; ORIGINAL COLOR ONLY WHEN
; THE COLOR ISN'T COLOR 0
; ( INVISIBLE )
END
TO FLASH1 :START
REPEAT 5 [SETCOLOR 0 WAIT 20 S
ETCOLOR :START WAIT 20 ]
END

TO FLASH0
REPEAT 5 [SETCOLOR 1 WAIT 20 S
ETCOLOR 0 WAIT 20 ]
END
```



The LOGO Poet:
USING RECURSION FOR LIST HANDLING

Since TI LOGO's graphics capabilities are so vast and so easy to use, there is a tendency to overlook its other features. List handling is a case in point: By combining some of LOGO's list primitives—such operations as FIRST, BUTFIRST, (or the converse LAST, BUTLAST)—with recursive [see adjoining *A Primer on Recursion and List Primitives*] OUTPUT lines, we can easily write programs to reverse a list, alphabetize a list, or even compose poetry. The several examples that follow will, I hope, demonstrate to you the powerful simplicity and list-manipulation potential of the language.

Verifying the presence or absence of a word in a list is a problem commonly encountered in list processing. The MIT LOGO group refers to this as the "MEMBER?" problem because the program is to answer the question, "Is a specified word in a specified list?" Some aspects of the program are obvious. For example, once the answer is obtained (whether TRUE or FALSE), it should OUTPUT to the user or program which called for the answer. It is also obvious that if the list is empty, the word is not in the list. Given just this much information, it is possible to frame a MEMBER? program:

```
TO MEMBER? :WORD :LIST
IF :LIST = [ ] OUTPUT "FALSE
.
.
END
```

apart, following Polya, notes that one way of solving a complex problem is to ignore the complex whole and focus on those parts which can easily be solved. [See *Mindstorms: Children, Computers, and Powerful Ideas* by Seymour Papert—available from the 99'er Bookstore.] In the "MEMBER?" problem, if the *first* word in the list were the target word, then it would be easy to detect it and solve the problem using the LOGO primitive FIRST*, which returns the first word in a list:

```
TO MEMBER? :WORD :LIST
IF :LIST = [ ] OUTPUT "FALSE
IF FIRST :LIST = :WORD OUTPUT
"TRUE
.
.
END
```

Now all that remains is solving for those cases in which the word is in an *interior* position or is *absent* from the list.

Were the word *second* in the list, the problem would be solved by adding a line using the LOGO primitive BUTFIRST, which returns all but the first word in a list of words:

```
IF FIRST BUTFIRST :LIST = :WORD OUTPUT
"TRUE
```

since the *second* word in the list is the *first* word in a list which excludes the first word. Similarly, the third word becomes the FIRST of the BUTFIRST* of the BUTFIRST of the list, the fourth word is the FIRST of the BUTFIRST of the BUTFIRST of the BUTFIRST of the list. It would be possible to write a separate line for each of those positions as well as the fifth, sixth, seventh or any other potential word position. However, a program that did this would quickly grow ponderous. Fortunately, in LOGO this is unnecessary. Notice that for each position an additonal BUT FIRST is all that is needed. The problem therefore requires only a *single* recursion line to complete the program:

```
TO MEMBER? :WORD :LIST
IF :LIST = [ ] OUTPUT "FALSE
IF FIRST :LIST = :WORD OUTPUT
"TRUE
OUTPUT MEMBER? BUTFIRST :LIST
:WORD
END
```

Now when we run the program by typing MEMBER? [A QUICK BROWN FOX] "FOX, the first stack checks to see if the list is empty or if the first word in the list matches the target word, FOX. Then it awaits the results of a second stack which runs MEMBER? with the truncated list and the target word. The second stack then awaits the results of a third stack which runs MEMBER? on BROWN FOX and "FOX. That stack then awaits the results of MEMBER? FOX "FOX which returns "TRUE (from the match in the second line). "TRUE is returned to the second stack which outputs "TRUE to the first stack which outputs "TRUE to the program which first called it (or to top level). In the event that there were no matches, one of the stacks would eventually run MEMBER? on an empty list and would output "FALSE.

Another common problem is to count the number of words in a list of words. As before, this problem is solved by outlining the obvious elements of the solution and the simplest case.

```
TO COUNT "LIST
OUTPUT some number
END
```

The simplest case occurs when the list is empty.

```
TO COUNT :LIST
IF :LIST = [ ] OUTPUT 0
OUTPUT some number
END
```

When a list has just *one* word in it, the program should recognize that and OUTPUT 1. Since a list with just one word is one word away from an empty list, The LOGO

* FIRST returns the first word in a list of words, or the first letter in a list of words, or the first letter in a word. LAST returns the last letter in a list of words, or the last letter in a word. BUTFIRST returns all but the first word in a list of words, or all but the first letter in a word. BUTLAST returns all but the last word in a list of words, or all but the last letter in a word.

operation BUTFIRST applied to that list would yield the empty list. If there were *two* words in a list, then obviously the list is just *two* words away from an empty list. If a recursive line were put into the program which (a) applied BUTFIRST and (b) added 1 to the count for every application of BUTFIRST, the program would count the words in the list.

```
TO COUNT :LIST
IF :LIST = [ ] OUTPUT 0
OUTPUT ( COUNT BUTFIRST :LIST
END
```

For another example, consider a program which will reverse a list. The simplest case would be a list with no words.

```
TO REVERSE
IF :LIST = [ ] OUTPUT [ ]
END
```

The next simplest case would be a list with just one word. For such a list we could have the program OUTPUT the SENTENCE or the word and an empty list.

```
TO REVERSE
IF :LIST = [ ] OUTPUT [ ]
OUTPUT SENTENCE ( LAST :LIST )
( REVERSE BUTLAST :LIST )
END
```

This solution can be applied to longer lists as well!

For a final example, let's use LOGO to "write" random poetry. As a first effort at LOGO poetry, we'll attempt some "free verse" by instructing a poet to string words together randomly from a list we select. First, we will need a program like SELECT to output a selected item from a list.

```
TO SELECT :N :LIST
IF :N = 1 OUTPUT FIRST :LIST
OUTPUT SELECT :N - 1 BUTFIRST
:LIST
END
```

Then we need a program to generate random numbers for SELECT. Because LOGO's RANDOM primitive provides the integers through nine, if our list is less than ten, we can get a COUNT of it and use that COUNT.

```
TO NUMB :LENGTH
CALL RANDOM "N
TEST BOTH :N > 0 :N < ( :LENGT
H + 1
IFT OUTPUT :N
IFF OUTPUT NUMB :LENGTH
END
```

By first typing

CALL COUNT :LIST "LENGTH

we can then use NUMB for the value of LENGTH. If we then type:

TYPE SELECT (NUMB :LENGTH) [*a list of words*]

the computer types one of the words in the list. We can write that as a program:

```
TO VERSE :LIST
TYPE SELECT ( NUMB :LENGTH )
LIST
END
```

A PRIMER ON RECURSION AND LIST PRIMITIVES

It is easier to understand recursion in LOGO if one imagines that each LOGO program is a job for a contractor to perform. Each contractor is a specialist and can do only *one* job. Every contractor follows strict working rules; these rules say that when the contractor sees STOP, he must stop, when he sees OUTPUT, he must pass back some information and then stop. Of course, when a contractor reaches an END, he also stops. When a contractor sees the name of any LOGO program inside of the program he is completing, he subcontracts that job out to another contractor. Thus, in COUNT [A, B, C], the first contractor reads the first line of the program, but the condition isn't met, so he moves to line two. There he is told to OUTPUT 1 + the COUNT of [B, C]. Since he can't do another program, he subcontracts the job. The subcontractor reads line 1 of COUNT and since it doesn't apply, he reads line 2. He is told to OUTPUT 1 + the COUNT of [C]. He can't do that, so he also subcontracts that job. The third contractor notes that line 1 doesn't apply and line 2 tells him to OUTPUT 1 + the COUNT of [ ]. He also must subcontract the job out, and so the fourth contractor reads line 1 of COUNT. Since the list is empty, he OUTPUTs 0 and passes the job back to the third contractor; he in turn adds 1 and then OUTPUTs 2. The first contractor adds 1 to that and then OUTPUTs 3, which is the correct answer. With this explanation, you should now be able to analyze a program which gives you the answer to a number X raised to N power.

```
TO EXPONENT :X :N
.
.
.
END
TO EXPONENT :X :N
IF :N = 0 OUTPUT 1
.
.
.
END
TO EXPONENT :X :N
IF :N = 0 OUTPUT 1
IF :N = 1 OUTPUT :X
OUTPUT (EXPONENT :X :N - 1) * :X
END
```

To turn this into a line of poetry, we should have a random number of such randomly picked words with a random number of spaces between words (E. E. Cummings's style) and then a carriage return:

```
TO SPACE
REPEAT RANDOM [PRINTCHAR 32 ]
END

TO LINE :LIST
REPEAT RANDOM [SPACE VERSE :LI
ST ]
PRINT SELECT ( NUMB :LENGTH )
:LIST
END
```

Note: PRINTCHAR 32 puts the character with ASCII code 32, a *space*, on the screen.

If we want continuing lines of poetry, we can write a recursive program:

```
TO LINES :LIST
LINE :LIST
LINES :LIST
END
```

Now, putting this all together we get:

```
TO POET :LIST
CALL COUNT :LIST "LENGTH
LINES :LIST
END
```

Now we can try converting POET into a program which produces either rhyming verse,, blank verse, or a finite number of lines of verse. One way to modify POET to produce rhymed verse is to give it two different lists—one of words for the interior words of each line of verse, and another of rhyming words for the last word in each line. Then the program can be changed so that only rhyming words are placed in end positions.

```
TO LINES :LIST :RHYMES
;
; LINES MUST ACCOMMODATE TWO
; LISTS
;
LINE :LIST :RHYMES
LINES :LIST :RHYMES
END

TO POET :LIST :RHYMES
;
; THE SECOND LIST MUST NOW BE
; GIVEN THE PROGRAM
;
CALL COUNT :LIST "LENGTH
CALL COUNT :RHYMES "LENGTHR
;
; NECESSARY TO FIND OUT HOW
; MANY RHYMING WORDS THERE
; ARE
;
LINES :LIST :RHYMES
END

TO LINE :LIST :RHYMES
;
; LINE MUST PUT RHYMES ONLY AT
; THE END OF EACH LINE
;
REPEAT RANDOM [SPACE VERSE :LI
ST]
PRINT SELECT ( NUMB :LENGTHR )
:RHYMES
;
; PUTS A RHYME AT THE END
;
END

TO SPACE
PC 32
END

TO COUNT :LIST
IF :LIST = [ ] OUTPUT 0
OUTPUT 1 + COUNT BUTFIRST :LIST
END

TO NUMB :LENGTH
CALL RANDOM "N
TEST BOTH :N > 0 :N < ( :LENGT
H + 1
IFT OUTPUT :N
IFF OUTPUT NUMB :LENGTH
END

TO SELECT :N :LIST
IF :N = 1 OUTPUT FIRST :LIST
OUTPUT SELECT :N - 1 BUTFIRST
:LIST
END

TO VERSE :LIST
TYPE SELECT ( NUMB :LENGTH ) :
LIST
END
```

You probably recognize that the problem of generating rhyming verse is one form of the problem of teaching the computer to write text which follows a specified rule (in this case each line must rhyme). The more general application of rules to text is nothing less than grammar. One of the grade school pupils in the Brookline project wrote a textbook rule program like POET which generated random sentences. After she saw the effects of changing parts of speech she exclaimed enthusiastically that she now understood what a noun was.

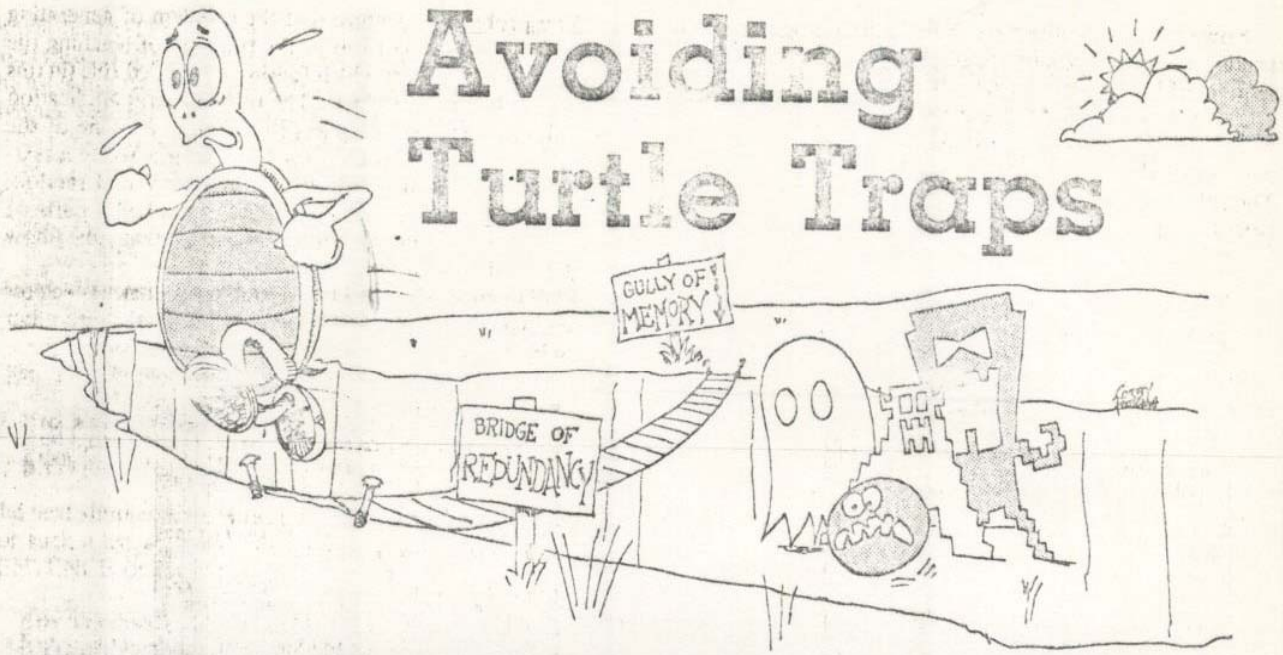
POET can also be quickly adapted to a sentence generator which young people can play with to make grammar meaningful.

```
TO SENTENCES
PRINT [TYPE A LIST OF ARTICLES
 AND THEN PRESS ENTER.]
CALL READLINE "ART
PRINT [TYPE A LIST OF NOUNS AN
D THEN PRESS ENTER.]
CALL READLINE "NOUNS
PRINT [TYPE A LIST OF ADJECTIV
ES AND PRESS ENTER.]
CALL READLINE "ADJ
PRINT [TYPE A LIST OF VERBS AN
D THEN PRESS ENTER. NOW WATCH.
]
CALL READLINE "VERBS
GRAMMAR :ART :NOUNS :ADJ :VERB
S
END
TO GRAMMAR :ART :NOUNS :ADJ :V
ERBS
TYPE SELECT ( NUMB ( COUNT :AR
T ) ) :ART
SPACE
TYPE SELECT ( NUMB ( COUNT :NO
UNS ) ) :NOUNS
SPACE
TYPE SELECT ( NUMB ( COUNT :VE
RBS ) ) :VERBS
SPACE
TYPE SELECT ( NUMB ( COUNT :AD
J ) ) :ADJ
SPACE
TYPE SELECT ( NUMB ( COUNT :NO
UNS ) ) :NOUNS
PRINT "
WAIT 30
GRAMMAR :ART :NOUNS :ADJ :VERB
S
END
```

SENTENCES can be made a better grammarian by adding distinctions of number and gender where appropriate; it can be made a more sophisticated language generator if GRAMMAR is altered to allow for conjunctions and subordinate clauses. All of these changes and more can be programmed by students as they learn both the specifics of grammar[1] and the mathematics of LOGO.

[1] Papert would probably argue that most students know the grammar which schools attempt to teach, but that the students do not have verbal labels for syntactical rules and parts of speech, and do not see the relevance of the labels once they are told them. A sentence generator program can make grammar "speech syntonic."

Avoiding Turtle Traps



Seymour Papert and his colleagues purposefully decided to structure LOGO to facilitate the writing of good computer programs. The concept of good programming is not superficially apparent. Of course, a program should accomplish its intended goal, but all programmers recognize that any goal can be achieved by many different types of programs. Beyond simply "working," there are a number of criteria by which programs can be judged. Programs which have multiple applications are generally better than single-purpose programs. Programs which are easier to debug and which can be understood by people other than the authors (or which can be understood by the authors at a future time) are more desirable. Pragmatically, programs which run faster or with fewer bits of memory are better than slower or more memory-intensive programs. Finally, some programs are aesthetically more appealing than others.

It is possible to find examples of program applications in which two or more of the criteria are in conflict. However, it is more often the case that the criteria are in accord. All of the criteria except for aesthetics are straightforward and relatively objective. Still, writing aesthetic programs is so satisfying that aesthetics will be considered first here.

For instance, you can write GO in LOGO, but programs with many different branches from GO commands are particularly inelegant: Why write poor programs when good ones are easier to write? Also inelegant are programs with hundreds of lines of code, especially when the code contains several repetitions of a series of commands. And programs with many inputs are generally less aesthetic than those with fewer inputs. Compare the aesthetics of two programs which count the number of words in a list:

```
TO COUNT :N :LIST
IF BUTFIRST :LIST = [ ] OUTPUT
:N
COUNT :N + 1 BUTFIRST :LIST
END
```

This program requires typing as input along with COUNT and the list in question requires a starting value of :N – 0.

```
TO COUNT :LIST
IF :LIST = [ ] OUTPUT 0
OUTPUT ( COUNT BUTFIRST :LIST
( 1 + 1 )
END
```

This program requires no superfluous input.

Elsewhere in this chapter, there is a fairly complex program, DYNATURTLE, which creates a turtle that obeys Newtonian laws of motion. Despite the complexity of the program, DYNATURTLE is relatively elegant: DYNATURTLE has only three lines, which are INITIALIZE, SETDYNATURTLE, and CONTROL. Each of those lines is, in turn, a brief program which serves a unique function. Contrast DYNATURTLE's elegance with a spaghetti-pole BASIC program which would achieve the same effects. Such a program would be long and littered with extensive GO-TO's.

A subtler example of elegant and inelegant programs can be made from the GRAMMAR program. The program was modified from a earlier POET program and was written:

```
TO GRAMMAR :ART :NOUNS :ADJ :V
ERBS
TYPE SELECT ( NUMB ( COUNT :AR
T ( ) ) :ART
SPACE
TYPE SELECT ( NUMB ( COUNT :NO
UNS ) ) :NOUNS
SPACE
TYPE SELECT ( NUMB ( COUNT :VE
RBS ) ) :VERBS
SPACE
TYPE SELECT ( NUMB ( COUNT :AD
J ) ) :ADJ
SPACE
TYPE SELECT ( NUMB ( COUNT :NO
UNS ) ) :NOUNS
PRINT "
WAIT 30
GRAMMAR :ART :NOUNS :ADJ :VERB
S
END
```

Notice how much of each line is repetitive. A better LOGO program would have taken advantage of that redundancy and used a broader application program:

```
TO WORDS :X
TYPE SELECT :NUMB :COUNT :X
TYPE :X
END
```

Then GRAMMAR could be written:

```
TO GRAMMAR :ART :NOUNS :ADJ :VERBS
WORDS :ART
SPACE
WORDS :NOUNS
SPACE
WORDS :VERBS
SPACE
WORDS :ADJ
SPACE
WORDS :NOUNS
PRINT
WAIT 30
GRAMMAR :ART :NOUNS :ADJ :VERBS
END
```

The second GRAMMAR program is more elegant and is shorter. It achieved greater simplicity by taking out of GRAMMAR all of the repeated functions and placing them in WORDS. All of the functions carrying out the program WORDS are directed at placing a single word from a designated set of words. The specification of the set and type of words is left for GRAMMAR, the program surrounding WORD. A common format for many well-written LOGO programs is:

```
TO DOSOMETHINGSPECIFICALLY :SPECIALINPUT
GENERALPURPOSEPROGRAM :GENERALINPUT
END

TO GENERALPURPOSEPROGRAM
  :GENERALINPUT
LOGO commands :GENERALINPUT
END
```

On occasion it is necessary to string together several general-purpose programs inside a specific-purpose program. In that case, the general program often requires that there be some set-up steps and some "fix-up" steps before and after the general program. Such programs have the form:

```
TO GENERALPURPOSE
SETUP
GENERALFUNCTIONS
FIXUP
END
```

Mathematicians may indeed recognize a similarity between the concept of elegance and aesthetics in programming and the expression of algebraic functions. There are many ways to express algebraic functions, but it is often more useful and always more elegant to express such functions in a form which collects common factors and simplifies terms even where such simplification may require a set-up or a quick fix-up manipulation along with the factoring.

There are two other major aspects to consider in order to write better LOGO programs. One is writing programs which don't run out of memory; the other is writing them to run as fast as possible. It is important to understand the major feat accomplished by Texas Instruments and by the MIT LOGO Lab in putting LOGO on the 99/4. LOGO is a very high level computer language which requires large amounts of memory. The architecture of microcomputers limits the speed with which large amounts of memory can be addressed. The TI LOGO which emerged from the joint efforts of TI and MIT represents an effort to compress code to the minimum memory requirement without compromising its applications. There are two tricks which they built into TI LOGO to make LOGO feasible on a micro. If you use these tricks you can gain even greater satisfaction from your computer. The first feature is an automatic garbage-collector. A garbage collector is a part of the operating system which takes used memory and makes that memory available for further uses. Of course, the garbage collector should not destroy and overhaul all of memory's work. The way that the automatic garbage-collector in LOGO recognizes when a unit of memory has served its purpose is by checking the instructions written in the memory. Below are examples of programs which permit or exclude the collector:

```
TO POLYGON :SIDE :ANGLE
FORWARD :SIDE
LEFT :ANGLE
POLYGON :SIDE :ANGLE
END
```

In this program, the garbage collector notes that each time POLYGON is entered (referred to as the *level* of POLYGON), there are no further commands or instructions after the line POLYGON :SIDES :ANGLES (called the recursive call line). Thus the piece of memory that was used to store POLYGON at that level is collected for reuse. If all memory gets used up in TI LOGO, the message "OUT OF SPACE" appears, but POLYGON will never generate that message because it will never run out of memory.

```
TO SIDE :LENGTH
FORWARD :LENGTH
END
```

This program will never run out of memory in TI LOGO because the program terminates.

```
TO POLYGON :SIDE :ANGLE
FORWARD :SIDE
LEFT :ANGLE
IF HEADING = 0 STOP
POLYGON :SIDE :ANGLE
PENUP
END
```

This program could use up all available memory before it reaches its stop conditions because the garbage collector cannot refurbish the memory used to execute this POLYGON at any level. The program leaves work to be done (namely PENUP) once control is passed back to the level of POLYGON.

Unfortunately, the garbage collector is not empowered with the authority to decide if any instructions following the recursion call are worth keeping, and so the following POLYGON program could run out of memory:

```
TO POLYGON :SIDES :ANGLES
FORWARD :SIDES
LEFT :ANGLES
POLYGON :SIDES :ANGLES
END
```

The only difference between the first POLYGON program and the one here is the empty line following the recur-

sion call and before END. The garbage collector sees that there is a line of commands and cannot tell that the line is useless, so it is barred from refurbishing the memory! Empty lines use up memory and can block garbage collection (depending on their location), so empty lines should be eliminated from your programs.

Finally, the operating system can work faster when fewer sprites are being used, i.e., programs which use no sprites run faster than programs which use sprites. The more sprites in use (generally), the slower the system operates. The reason for the slight degradation in response time is obvious—the system has to check to see which, if any, sprites must be displayed or moved. The system checks on its sprites by looking up the highest number of sprite called upon. For example, TELL 31 or TELL SPRITE 31 would cause the system to check on every sprite from 31 on through to sprite 0. Such a check is necessary (from the user's perspective) only if all 32 sprites are being used. If only one sprite is needed, then the user should type TELL 0 or TELL SPRITE 0 and the system would skip the checkup on sprites 1 to 31, thus saving a small amount of time.

Student Reactions to a Four Week LOGO Class
By Gene Branum

Students pick up these principles quickly. For instance, Gene Branum, a student in a four-week LOGO course, reflects on this experience:

"The expectations of the students varied—we wanted to know more about computers, we wanted a different Jan-term experience, or maybe just a free Jan-term. Whatever the motivation, all came away affected in some way by our experience. All experienced both the frustration of failure and the flush of triumph as the computer finally 'did what it was supposed to.'

"The format for our experience was a four-week mini-term (Jan-term) at Austin College. Our class met; five days a week for two hours, and we were required to spend at least one hour of work on our own as well. This requirement was easily met; as one student put it, 'It was not unusual to spend four hours at a time' on the computer. Needless to say, the experience was very intense, and there was a great deal of self-teaching. This was felt to be one of the greatest strengths of the course.

"Professor Hank Gorman did a fine job of teaching the basics early in the course. As he told us his expectations, we scoffed. After two weeks, he told us, we would be drawing cartoons and making up games. Even though his leadership was great, the majority were insecure about 'the machine.' Our confidence, however, grew with experience and familiarization.

"The two greatest aspects of the course for all of us were (1) the team experience and (2) experience in general problem solving skills. The true strength of LOGO is that students, working together, can teach each other massive amounts of material. The realization that *everyone* had problems put us all on the same level. Sharing ideas and solutions became important for everyone because no one could work totally independently. Many social experiences allow students to interact, but LOGO is one of the few that forces students to *think together.*

"Without exception, all of the students involved in the course commented that, after LOGO, they knew better how to approach a complex problem. Dr. Gorman spent several class periods on problem solving skills: decomposition, recursion, naming, multiple descriptions, and the 'little men.' These skills not only aided our search for solutions to LOGO problems, but also for problems that require a thinking solution. The overriding principle of LOGO is that the simple builds to the complex, which is its major strength as a system for any age-group.

"While it was widely agreed upon that none of us 'mastered' LOGO, each of us developed confidence in our abilities to control the computer and make it do what we requested. The LOGO experience allowed everyone to use logical approaches to problem solving and gain valuable hands-on experience in a discipline that continues to increase in importance."

The following programs, which students wrote during this course, show an emerging appreciation for elegance, speed, and simplicity in programming. Except for correction of typographical errors, their work hasn't been edited in an attempt to find still more elegent ways of achieving their programs' goals. Note, however, that they all grasp the essentials of esthetic programming.

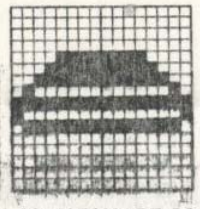
Space Pylon Racer

Once set up, the player guides his saucer through pylons. Two shapes must be made first (check graph paper). The keys control the saucer. E moves it upward, X moves it downward, D moves it forward, S moves it backward, F speeds it up, A slows it down. If the ship hits a pylon, the beep sounds.

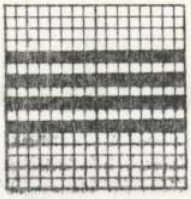
Use arrow keys to change direction.
Use F for fast speed.
Use A for slow speed.

```
TO GAME
CS
TELL :ALL SS 0 SC 0 CARRY 0
SET
TELL 0
SPR
END
```

```
TO CHECK
TEST BOTH ( BOTH XCOR > - 65 X
COR < - 55 ) ( BOTH YCOR > 74
YCOR < 90 )
IFT GO -B
TEST BOTH ( BOTH XCOR > - 65 X
COR < - 55 ) ( BOTH YCOR > 54
YCOR < 70 )
IFT GO -B
TEST BOTH ( BOTH XCOR > 85 XCO
R < 115 ) ( BOTH YCOR > 74 YCO
R < 90 ) -B
IFT GO
TEST BOTH ( BOTH XCOR > 85 XCO
R < 115 ) ( BOTH YCOR > 54 YCO
R < 70 )
IFT GO -B
TEST BOTH ( BOTH XCOR > 15 XCO
R < 45 ) ( BOTH YCOR < - 70 YC
OR > - 86 )
IFT GO -B
TEST BOTH ( BOTH XCOR > 15 XCO
R < 45 ) ( BOTH YCOR < - 50 YC
OR > - 66 )
IFT GO -B
B:
IFT BEEP WAIT 15 NOBEEP
END
```



MAKESHAPE 20
Saucer

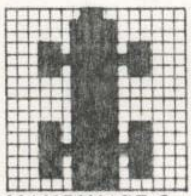


MAKESHAPE 21
Pylon

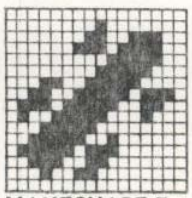
```
TO SET
TELL 0 CARRY 20 SC :RED SXY -
100 0
TELL [1 2 3 4 5 6 ]
CARRY 21 SC :BLACK
TELL 1 SXY -50 80
TELL 2 SXY -50 60
TELL 3 SXY 30 ( -| 80 )
TELL 4 SXY 30 ( -| 60 )
TELL 5 SXY 100 80
TELL 6 SXY 100 60
END

TO SPR
CONTROL
CHECK
SPR
END

TO CONTROL
IF RC? = "TRUE THEN TEST 3 = 4
  ELSE STOP
CALL RC "E TELL 0 SH 0 STOP
IF :Z = "X TELL 0 SH 180 STOP
IF :Z = "D TELL 0 SH 90 STOP
IF :Z = "S TELL 0 SH 270 STOP
IF :Z = "F TELL 0 SS 10 STOP
IF :Z = "A TELL 0 SS 5 STOP
END
```

Spinout

This program was designed as a cartoon to depict two Indianapolis-style racing cars racing, crashing, burning, and being towed away. The central program, SPINOUT, contains 7 subprograms. These short programs make the central program neat and concise.



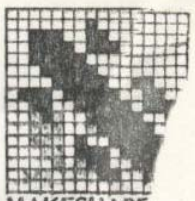
MAKESHAPE 6



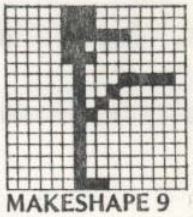
MAKESHAPE 7

```
TO SPINOUT
WAVE
MOVE
WAIT 350
SWERVE
WAIT 20
SPIN
WAIT 50
BURN
TOW
END

TO SETUP
TELL 0
CARRY 6
SC 6 SS 0 SH 0
HOME
TELL 1 CARRY 6
SC 4
SS 0 SH 0
HOME
TELL 0 SX 15
END
```

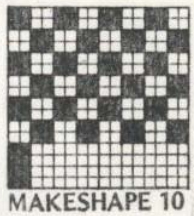
```
TO WAVE
TELL 3 CARRY 9
SC 1
TELL 4 CARRY 10
SC 1
TELL 3 SX -75 SY 0
TELL 4 SX -60
SY 10
SETUP
TELL 4 REPEAT 4 [SY 5 WAIT 10
SY 10 WAIT 10 SY 15 WAIT 10 SY
 10 ]
BEEP
WAIT 30
NOBEEP
END

TO MOVE
TELL 0
SS 12
TELL 1 SS 19
TELL [3 4 ] SC 0
END

TO SWERVE
TELL 1
REPEAT 10 [SX -5 WAIT 5 SX 0
WAIT 5 SX -2 WAIT 5 SX 10 WAI
T 6 ]
SX 15
END

TO SPIN
CALL [0 1 ] "TEAM
TELL :TEAM SS 12
REPEAT 5 [CARRY 6 WAIT 5 CARRY
 8 WAIT 5 CARRY 6 WAIT 5 CARRY
 7 WAIT 5 CARRY 6 WAIT 5 CARRY
 8 WAIT 5 ]
END

TO BURN
REPEAT 10 [CB 6 WAIT 5 CB 11 W
AIT 5 CB 9 WAIT 5 ]
TELL :TEAM
SC 1 SS 0
CB 7
END

TO TOW
TELL 5 CARRY 11
SC 9
SX 100
SY -70
WAIT 100
SH 270
SS 10
WAIT 115
SS 0 WAIT 205
CALL [0 1 5 ] "T
TELL :T
SH 90
SS 10
WAIT 175 SC 0 SS 0
END
```



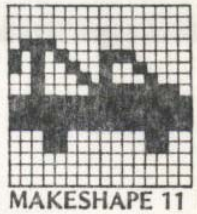
MAKESHAPE



MAKESHAPE 9



MAKESHAPE 10



MAKESHAPE 11

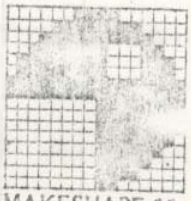
Munchie

Munchie illustrates how one can program a sprite to move to certain locations where an object may be found. After testing coordinates within the procedure, it eats that object and continues on until it eats all objects. You move Munchie by using arrow keys, and set speed by using keys 0, 5, and 1. You should stop Munchie when passing over the object to be eaten.

```
TO MUNCHIE
SETUP
MOVE
END
```

```
TO SETUP
TELL 1 CARRY :PLANE
SC :BLUE SXY -50 0
TELL 2 CARRY :TRUCK
SC :GREEN SXY 50 30
TELL 3 CARRY :ROCKET
SC :RED
SXY -10 (-70)
END
```

MAKESHAPE 10

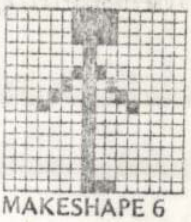


MAKESHAPE 11

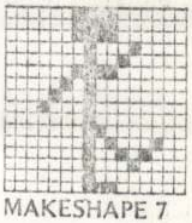
```
TO MOVE
TELL 0 SC :WHITE
CARRY 10 BEEP WAIT 5 CARRY 11
NOBEEP
TEST RC?
IFF CALL "A "M
IFT CALL "RC "M
IF :M=1 "S SH 270
IF :M=2 "S SH 0
IF :M=3 "S SH 90
IF :M=4 "S SH 180
IF :M=0 SS 0
IF :M=1 SS 5
IFF CARRY 10 BEEP CARRY 11 NOB
EEP
CHECK
END

TO EAT3
REPEAT 25 [BEEP WAIT 2 NOBEEP
WAIT 2 ]
TELL 3 SC 0
TELL 0 SS 5
MOVE
END

TO EAT2
REPEAT 25 [BEEP WAIT 2 NOBEEP
WAIT 2 ]
TELL 2 SC 0
TELL 0 SS 5
MOVE
END

TO EAT1
REPEAT 25 [BEEP WAIT 2 NOBEEP
WAIT 2 ]
TELL 1 SC 0
TELL 0 SS 5
MOVE
END

TO CHECK
TEST BOTH XCOR > -55 XCOR < -
45
IFT TEST BOTH YCOR > - YCOR
< 5
IFT TELL 0 SS 0 EAT1
IFF CARRY 10 BEEP WAIT 5 CARRY
11 NOBEEP
TEST BOTH XCOR > 45 XCOR < 55
IFT TEST BOTH YCOR < 35 YCOR >
25
IFT TELL 0 SS 0 EAT2
IFF CARRY 10 BEEP WAIT 5 CARRY
11 NOBEEP
TEST BOTH XCOR < -10 XCOR > -
20
IFT TEST BOTH YCOR > -80 YCOR
< -70
IFT TELL 0 SS 0 EAT3
IFF MOVE
END
```

Fieldgoal Movie

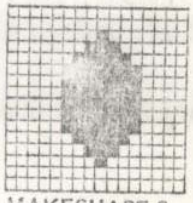
```
TO WIPE
CS
TELL :ALL
SC 0
SS 0
SXY 110 95
END

TO SETUP
WIPE
TELL 1 CARRY 8
SC :YELLOW
HOME
TELL 2 CARRY 6
SC :RED
SXY -5 8
TELL 3 CARRY 9
SC :BLUE
SXY 70 8
TELL 4 CARRY 12
SC :BLUE
SXY 70 23
TELL 5 CARRY 10
SC :BLACK
SXY 80 (-8)
END

TO KICK
SETUP
WAIT 120
TELL 2
CARRY 7
TELL 1
SH 90
SS 10
RISE
TELL 5
CARRY 11
REPEAT 10 [JUMP ]
END

TO RISE
TELL 1
CARRY 8
SH 0
SS 1
SH 45
WAIT 60
SH 90
WAIT 100
SH 135
WAIT 40
SC 0
SS 9
END

TO JUMP
TELL 2
CARRY 6
SY 14
WAIT 10
SY 8
WAIT 10
END
```



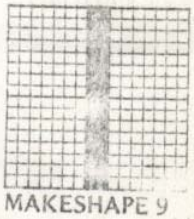
MAKESHAPE 6



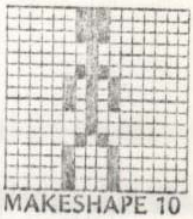
MAKESHAPE 7



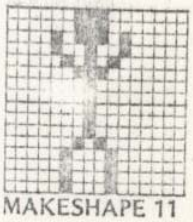
MAKESHAPE 8



MAKESHAPE 9

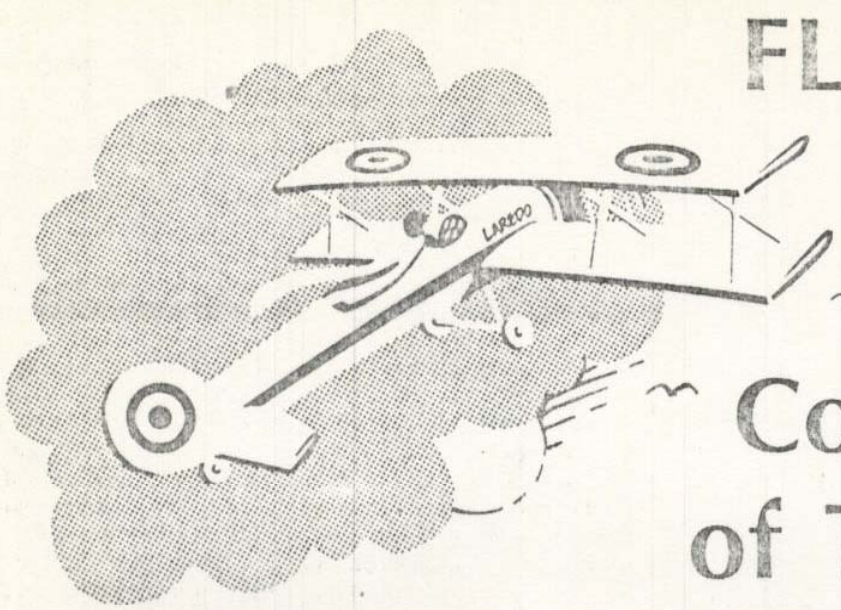


MAKESHAPE 10



MAKESHAPE 11

MAKESHAPE 12

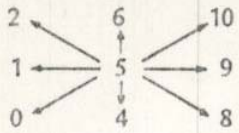


FLY AWAY with the JOY Commands of TI LOGO

You push the stick forward, and the aircraft begins to roll. You then gradually pick up speed and start moving down the runway. After reaching takeoff speed, you move the stick again, and suddenly you're airborne. Now you have control of the skies—to fly high or low, do loops and other maneuvers, and then land. But be careful with your speed! You don't want to stall and crash.

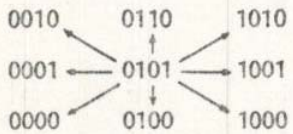
This isn't flight training or a simulator. It is a TI LOGO procedure that gives you the opportunity to fly by keyboard or joystick. It uses either the arrow keys or the JOY 1 and JOY 2 commands. The JOY commands return one of nine values depending on the position of the joystick, thus opening a wide range of possibilities for interactive games and other activities.

At first, it might appear that the nine values have little relationship to each other. You'll note that the three and the seven were omitted. However, the pattern of the values is quite interesting.

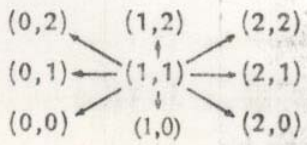


Moving from left to right in each row, you'll note that each digit is four more than the previous digit, i.e., 2 + 4 = 6, 6 + 4 = 10, etc. Moving from bottom to top in each column, observe that each digit is one more than the previous one.

These patterns begin to suggest why three and seven were omitted from the values assigned. However, to make the logic behind these patterns even more graphic, let's convert them to binary numbers.



Now look at the first two digits in each column. You'll note that they are the same, representing from left to right, 0, 1, and 2. Also, if you look at the last two digits in each row, you'll note that they are also the the same. Moving from bottom to top, they also represent 0, 1, and 2. So what we really have here is a distinctive coordinate system with real meaning, rather than what might first be perceived as a random placement of values.



Let X and Y be used to name these coordinates. The coordinates for the joysticks can be assigned with the command.

```
MAKE "Y (JOY1) / 4
MAKE "X (JOY1) - 4 * :Y
```

Now let's put these JOY commands to work in FLYAWAY, a procedure developed by Roger Kirchner, a fellow YPLA member. This is a procedure for one or two players that tests each player's ability to take off and safely land an airplane on the runway shown on the screen. Either the direction keys on the keyboard or the joysticks can control the plane.

The joystick commands are incorporated in the procedure, STICK S. Push the stick forward, and the aircraft increases its speed.

```
IF :X = 6 THEN FASTER
```

Pull the stick back and the aircraft slows down:

```
IF :X = 4 THEN SLOWER
```

To minimize the chance for error, direction commands are accessed by merely moving the joystick to the left or

right. It does not matter whether you hit position 0, 1, 2; the aircraft will turn left.

```
IF :X > 4 THEN TURNLEFT
IF :X < 6 THEN TURNRIGHT
```

Of course, it would be possible to add additional maneuvers using each of the nine joystick positions. This would require a much more sensitive touch to the joystick, but that could also add to the challenge of the flight.

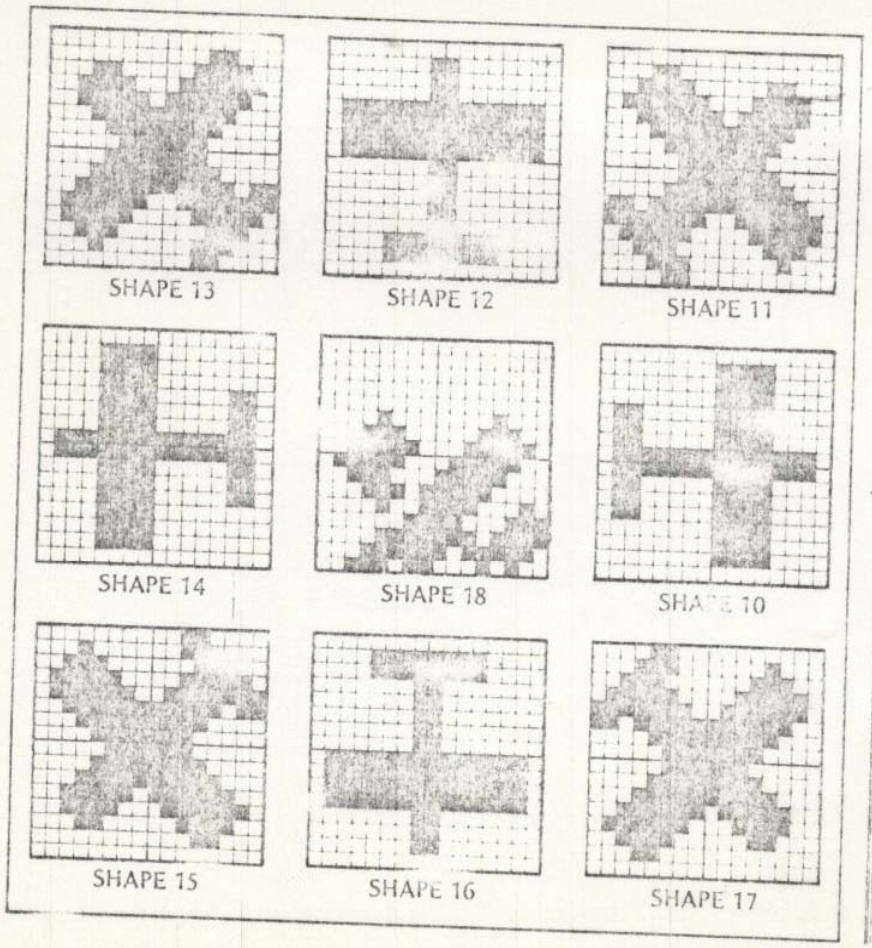
With each turn of the aircraft, a new shape is called to show that new position. These range from #10 through #18. The first shape, #10, is similiar to the Plane shape in TI LOGO. The next shape shows the aircraft at a 45° angle. The other shapes depict the plane in a 90° angle, 135° angle, at 180°, 225°, 270° and 315°. Shape #18 depicts the crash.

The CONTROLJOY and the CHECK P procedures control the aircraft in the air. CHECK P monitors the speed and "altitude" of the aircraft to test for the CRASH parameters.

FLYAWAY is an excellent graphics program that begins to tap the power of the LOGO language. It goes quite a bit further than merely drawing pictures with the computer. This is an important point to realize. But unfortunately, not all educators do: In a November issue of *Infoworld*, one educator stated that although he can understand young people enjoying maybe 50 to 100 hours of drawing pictures w LOGO, he would imagine they would then tire of language and move on to the other things.

Do chess players ever tire of chess? Do chess masters e ...lly feel that they have mastered the game? Probably n The moves of chess can be easily learned by primary gr youngsters, but entire *lifetimes* are spent learning the gar Certainly the graphics capabilities and the speed of LOGO are spectacular. Indeed, they tend to overshad the other attributes of the language. Where that happe it is most unfortunate because LOGO offers a young p son so much more than just graphics—much more th BASIC and some other high level languages.

For example, look closely at BASIC: It uses a fin number of commands that must be strung together statements that tend to hide the operation of the prograr Were the operation of BASIC programs easily discerne TRACE would be unnecessary. LOGO, on the other han is a virtually unlimited language. If the command doesr exist, use your imagination and create it! This is t marvellous challenge of TI LOGO—using your imagin tion and creativity to discover the *real* potential of the con puter. Way back in the dark ages before microcomputer Albert Einstein expressed a truth that is especially relevar to today's computer learning environments: "Imaginatio is more powerful than knowledge." Undoubtedly, Einstei would have approved of TI LOGO.




```
TO SETUP
CS
VANISH
PRINT [INSTRUCTIONS Y / N ]
IF RC = "Y THEN HELP MAKE "X R
C
CS
PRINT [NAME OF BLUE PILOT? ]
MAKE "PILOT1 READLINE
PRINT [NAME OF RED PILOT? ]
MAKE "PILOT2 READLINE
PRINT
PRINT [FLYING WITH KEY CONTROL
S OR JOYSTICK ( K / J ) ? ]
MAKE "MODE F READLINE
MAKE "XS ( - 100 )
MAKE "Y1S ( - 40 )
MAKE "Y2S ( - 60 )
MAKE "UPOFF ( - 30 )
MAKE "DOWNOFF ( - 72 )
CS
END

TO FLYAWAY
SETUP
RUNWAY
SETPLANE 1 SETPLANE 2
TEST :MODE = "J
IFF CONTROL
IFT CONTROLJOY
END

TO CONTROL
MAKE "X RC
IF :X = "E THEN TELL 1 FASTER
IF :X = "I THEN TELL 2 FASTER
IF :X = "X THEN TELL 1 SLOWER
IF :X = "M THEN TELL 2 SLOWER
IF :X = "S THEN TELL 1 TURNLEF
T
IF :X = "L THEN TELL 2 TURNLEF
T
IF :X = "D THEN TELL 1 TURNRIG
HT
IF :X = "K THEN TELL 2 TURNRIG
HT
IF :X = "Q THEN STOP
CHECK 1 CHECK 2
CONTROL
END
```

```
TO HELP
CS
PRINT [ "FLYAWAY" ]
PRINT [ ]
PRINT [BLUE PILOT USES KEYS E,
 S, D, X OR JOYSTICK 1 ]
PRINT [ ]
PRINT [RED PILOT USES KEYS I,
 J, K, M OR JOYSTICK 2 ]
PRINT [ ] PRINT [ ]
PRINT [CONTROLS: ]
PRINT [ ]
PRINT [FASTER: E I OR STICK FO
RWARD ]
PRINT [SLOWER: X M OR STICK BA
CK ]
PRINT [TURN LEFT: S J OR STICK
 LEFT ]
PRINT [TURN RIGHT: D K OR STIC
K RIGHT ]
PRINT [ ] PRINT [ ]
PRINT [SEE WHO CAN TAKE OFF AN
D LAND SAFELY FIRST ]
PRINT [ ]
PRINT [BON VOYAGE ]
END

TO VANISH
TELL :ALL HOME SH 0 SS 0 CARRY
 0
END

TO CONTROLJOY
STICK 1 CHECK 1
STICK 2 CHECK 2
IF RC? THEN STOP
CONTROLJOY
END

TO SETPLANE :P
IF :P = 1 THEN MAKE "YS :Y1S E
LSE MAKE "YS :Y2S
TELL :P SC ( 2 * :P + 2 ) SXY
 XS :YS SS 0 SH 90 CARRY 10
END
```

```
TO STICK :S
MAKE "X JOY :S
TELL :S
IF :X < 4 THEN TURNLEFT
IF :X > 6 THEN TURNRIGHT
IF :X = 6 THEN FASTER
IF :X = 4 THEN SLOWER
END

TO CHECK :P
TELL :P
IF SPEED > 10 THEN STOP
IF YCOR = :XS THEN STOP
TEST EITHER YCOR > :UPOFF YCOR
 < :DOWNOFF
IFT CRASH
IFF IF SPEED = 0 THEN WELCOME
ELSE STOP
WAIT 120
SETPLANE WHO
END

TO TURNRIGHT
RT 45
IF SHAPE = 10 THEN CARRY 17 EL
SE CARRY SHAPE - 1
END

TO TURNLEFT
LT 45
IF SHAPE = 17 THEN CARRY 10 EL
SE CARRY SHAPE + 1
END

TO SLOWER
IF SPEED > 0 THEN SS SPEED - 5
END

TO FASTER
IF SPEED < 100 THEN SS SPEED +
 5
END
```

```
TO PLUS :N
IF :N < 17 THEN OUTPUT :N + 1
ELSE OUTPUT 10
END

TO MINUS :N
IF :N > 0 THEN OUTPUT :N - 1 E
LSE OUTPUT 17
END

TO WELCOME
PRINT SE [NICE LANDING. ] PILO
T WHO
END

TO CRASH
PRINT SE [NOT SO GOOD. ] PILOT
 WHO
MAKE "I 10
MAKE "DROP YCOR - ( -50 )
A:
SY ( -50 ) + ( :DROP * :I ) /
 10
IF :I > 0 THEN MAKE "I :I - 1
GO "A
CARRY 18
SS 0
END

TO PILOT :N
IF :N = 1 THEN OUTPUT :PILOT1
ELSE OUTPUT :PILOT2
END

TO RUNWAY
TELL TILE 96 SC 12 15 ]
TELL TILE 104 SC :GREEN
MAKE "TILES [ 104 104 100 100 9
6 100 100 104 104 ]
MAKE "ROW 15
A:
IF :TILES = [ ] THEN STOP
MAKE "T F :TILES
MAKE "COL 0
REPEAT 32 [PT :T :COL :ROW MAK
E "COL :COL + 1 ]
MAKE "TILES BF :TILES
MAKE "ROW :ROW + 1
GO "A
END
```

Problem Solving
WITH LOGO

It is pleasureable to work with a language like LOGO because it gives us something to "think with," and it encourages us to think in what Papert has called "mind-sized bites." The solution of a problem can be identified with the definition of a procedure. If the problem is simple, we can specify the procedure directly. Otherwise, we try to specify it in terms of a small number of simpler procedures.

Often, this method leads to a complete solution of a problem. But sometimes, a problem is so complex that the method leads to an indefinite number of problems. A solution seems hopeless.

But suppose that new problems have the same form as previously encountered problems, and are simpler. The problem will be solved at least "theoretically," if the rules lead to a solution in a finite number of steps. Such a solution is said to be *recursive*.

One of the beauties of a language such as LOGO is that recursive procedure definitions are allowed. And writing a LOGO procedure not only gives a *theoretical* solution, but a practical one which can be carried out by executing the procedure. Of course, for the latter, one needs access to a TI-99/4A with TI LOGO (or some other implementation of LOGO).

In thinking through the solution of a problem, one often works "both ends." The big picture leads to smaller pictures. But also details occur which can be incorporated into procedures, which then make the solution of larger problems easier.

Translating the Pig Latin

As a concrete example of these ideas, consider the momentous task of translating an English word into Pig Latin. According to my children, the rule is to add "HAY" at the end of a word beginning with a vowel, otherwise to take the consonant sound from the front, add "AY" to it, and put it at the end. Thus "AND" translates to "ANDHAY", and "BREAK" translates to "EAKBRAY."

These rules lead immediately to a LOGO procedure for accomplishing the task:

```
TO TRANWORD :W
TEST MEMBER FIRST :W [A E I O
U]
IFT OUTPUT TRANVWORD :W
IFF OUTPUT TRANCWORD :W
END
```

This procedure reduces our problem to the solution of three simpler problems, which we might need to reduce further. The procedures we need are:

MEMBER *object list*
TRANVWORD *word*
TRANCWORD *word*

MEMBER returns TRUE if *object* is in *list* and returns FALSE otherwise. TRANVWORD translates *word* if it begins with a vowel. TRANCWORD translates *word* if it begins with a consonant. We can hope that MEMBER is a utility built into LOGO. It isn't, but this is no problem. Nearly anything that isn't a primitive can be built in.

At any stage in the solution process we can decide to work on big problems or focus on little ones. The solution of a problem isn't a linear process, even if solutions are usually presented as if the process were orderly and straightforward. The LOGO procedures document and organize progress.

Let's focus on the problem of deciding membership. If object is in a list, it is either the first item of the list, or else


```
TO HELP
CS PRINT [FOR PIGLATIN PRACTIC
E, ]
PRINT [TYPE "PIGLATIN" ]
END

TO PIGLATIN
CS
PRINTPIG [I WILL HELP ]
PRINTPIG [YOU LEARN PIGLATIN ]

A:
PRINT [ ]
PRINTPIG [TYPE A SENTENCE ]
MAKE "LINE READLINE
IF :LINE = [ ] THEN PRINTPIG [
THAT WAS FUN ] STOP
PRINTPIG :LINE
GO "A
END
```

```
TO TRANCWORD :K :W
IF :K = 1 THEN MAKE "VOWELS [A
E I O U ]
IF :K = 0 THEN MAKE "VOWELS [A
E I O U Y ]
TEST MEMBER FIRST :W :VOWELS
IFT OUTPUT WORD :W "AY
IFF OUTPUT TRANCWORD 0 ( WORD
BUTFIRST :W FIRST :W )
END

TO TRANWORD :W
TEST MEMBER FIRST :W [A E I O
U ]
IFT OUTPUT TRANVWORD :W
IFF OUTPUT TRANCWORD 1 :W
END
```

```
TO PRINTPIG :LINE
TEST :LINE = [ ]
IFT PRINTCHAR 13
IFF TYPE TRANWORD F :LINE PRIN
TCHAR 32 PRINTPIG BF :LINE
END

TO TRANVWORD :W
OUTPUT WORD :W "HAY
END

TO MEMBER :X :SET
IF :SET = [ ] THEN OUTPUT "FAL
SE
TEST :X = FIRST :SET
IFT OUTPUT "TRUE
IFF OUTPUT MEMBER :X BF :SET
END
```

it is the first of a truncated list, or it is not in the list. The definition is, naturally, recursive:

```
TO MEMBER :X :SET
IF :SET = [ ] THEN OUTPUT "FAL
SE
TEST :X = FIRST :SET
IFT OUTPUT "TRUE
IFF OUTPUT MEMBER :X BF :SET
END
```

With this definition, MEMBER FIRST :W [A E I O U] will return TRUE if :W begins with a vowel, and FALSE if it doesn't.

The definition of TRANVWORD is so simple we can write the procedure anytime. Let's do it now:

```
TO TRANVWORD :W
OUTPUT WORD :W "HAY
END
```

The (undocumented) primitive WORD takes two words as input and outputs the word formed by joining them.

The definition of TRANCWORD takes more thinking. We want it to be recursive. We want to move letters from the beginning to the end until the first letter is a vowel, and then add "AY". We are led to:

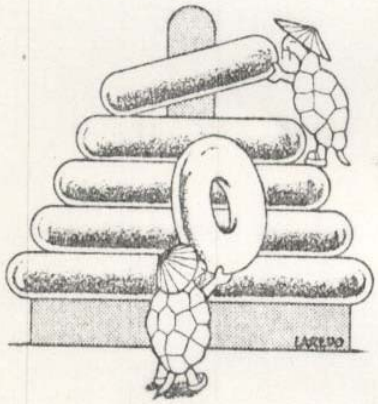
```
TO TRANCWORD :W
TEST MEMBER FIRST :W [A E I O
U ]
IFT OUTPUT WORD :W "AY
IFF OUTPUT TRANCWORD (WORD BU
TFIRST :W FIRST :W )
END
```

If we try (that is, think through, or execute in LOGO) TRANCWORD "BREAK, we find it will return EAKBRAY, as desired. And TRANCWORD "YOU returns OUYAY. But TRANCWORD "BY runs out of space because the recursion cannot end. Evidentally Y must be added to the list of vowels. But then TRANCWORD "YOU would return YOUHAY and not OUYAY.

Can you fix this bug? We want Y to count as a vowel only if it isn't the first letter. One solution is to use two inputs to TRANCWORD, one of which is a flag. This solution, as well as the generalization to translating a sentence, can be seen by reading the PIGLATIN procedure and the procedures it calls.

99'er

TOWER OF HANOI



Now we turn to a less frivolous example. The Tower of Hanoi is a puzzle familiar to many. It consists of three pegstands. One contains a "tower" of circular rings. The object is to move the tower from one peg to another, moving one ring at a time, and never putting a larger ring on top of a smaller one. There is rumored to be a Buddhist priest working on a puzzle with 64 rings; when he finishes, the world will end. If he makes one move per second, how much should we worry?

We can use LOGO to worry about this problem. We need a procedure, say NUMMOVES, which takes for input the number of rings and outputs the number of moves. Suppose we think of the task this way: Move the top n – 1 rings to an auxiliary peg, then move the largest ring, then move the smaller n – 1 rings onto the largest.

The way of viewing the problem leads to the following recursive definition for NUMMOVES:

```
TO NUMMOVES :N
TEST :N = 1
IFT OUTPUT 1
IFF OUTPUT 1 + 2 * (NUMMOVES
:N - 1 )
END
```

Trying this procedure, we find that NUMMOVES 2 = 3, and also that NUMMOVES 3 = 7. The reader might try to find a formula for NUMMOVES n, and also the value of NUMMOVES 64.

Of more interest is a procedure for actually solving the puzzles, and beyond that, for implementing the solution graphically. By the above reasoning, what we need is a procedure SOLVE with four inputs:

SOLVE *n peg1 peg2 peg3*

which would move the top *n* rings from *peg1* to *peg2* using *peg3*. Using the rules we obtain:

```
TO SOLVE :N :P1 :P2 :P3
TEST :N = 1
IFT GETRING :P1 SETRING :P2
IFF SOLVE :N - 1 :P1 :P3 :P2
SOLVE 1 :P1 :P2 :P3
SOLVE :N - 1 :P3 :P2 :P1
END
```

To have LOGO print out the moves in order, we need to implement two procedures called GETRING and SETRING:

In the meantime, let's implement GETRING and SETRING simply so we can test our solution:

```
TO GETRING :P
TYPE (PICK UP ) PRINTCHAR 32
TYPE :P PRINTCHAR 32
END
```

```
TO SETRING :P
TYPE ISET ON 1 PRINTCHAR 32
PRINT :P
END
```

Now, if we enter SOLVE 2 "A "B "C, the output will be:

PICK UP A SET ON C
PICK UP A SET ON B
PICK UP C SET ON B

The number of moves for three rings is 3, as expected. What will be the seven moves for SOLVE 3 "A "B "C? Try it!

We've looked at a LOGO procedure for solving the Tower of Hanoi as an abstraction. This procedure, SOLVE, prints out—as a list—the sequence of moves necessary for the solution. But given the graphics power of LOGO, we should be able to design a program—a series of procedures—which will represent the actual movement of rings from one peg to another graphically. And, in fact we *can* use LOGO's MAKECHAR command to define the required graphics, called *tiles*, and we *can* move these newly-defined tiles about, using LOGO procedures. So let's begin at the beginning.

Let A, B, and C be the three pegs. When we know which rings are on which pegs, we then know the particular state of the puzzle. In our LOGO implementation, the variables A, B, and C will be the names for lists which tell us which rings are on each peg. Our puzzle will have 8 rings. Let us number them 1 through 8 in order of increasing size. The beginning position, with all rings on peg A, is represented by :A = [1 2 3 4 5 6 7 8], :B = [ ], and :C = [ ]. Moving the top ring from A onto B results in the state :A = [2 3 4 5 6 7 8], :B = [1], :C = [ ]. In essence, a move consists of removing a number from the beginning of one list and adding it to the beginning of another list. At the same time, of course, the graphic representation ring must be erased and redisplayed in the correct position.

Let us first construct a procedure HANOI, which will allow us to play with the puzzle and then, when we want, solve it automatically.

```
TO HANOI
INITIALIZE
SETUP
PLAY
SETUP
SOLVE 8 "A "B "C
END
```

INITIALIZE should set colors and define constants. SETUP should display the puzzle with all the rings on peg A. PLAY should allow us to pick rings up and put them down by simply pressing the names of the corresponding pegs. Play might continue until 'Q' is pressed. The puzzle should then be redisplayed and solved automatically, beginning with the rings on peg B. The procedure SOLVE was developed in the previous section. Procedures SETUP, PLAY, and SOLVE will depend on workhorse procedures GETRING and SETRING. The requirements for INITIALIZE will become apparent as we make choices about representation.

Assume that INITIALIZE assigns the value 8 to N and :TOP is the number of the ring to be displayed. Then SETUP can be:

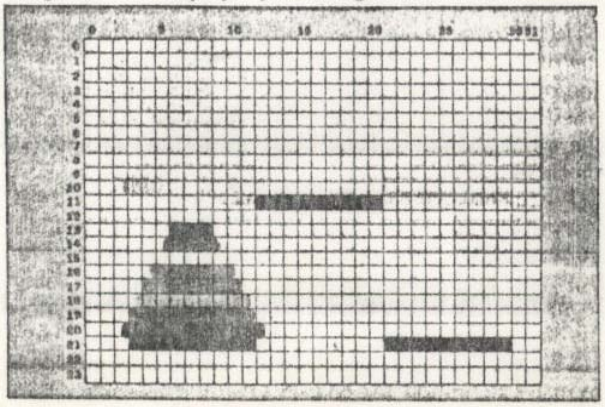
```
TO SETUP
CS
STAND "A
STAND "B
STAND "C
MAKE "TOP :N
MAKE "A [ ]
MAKE "B [ ]
MAKE "C [ ]
REPEAT :N [SETRING "A MAKE "TO
P :TOP - 1 ]
END
```

Using utilities MEMBER?, EMPTY, and ALARM, we can write PLAY in such a way as to validate all inputs. We want to accept either 'Q' or to stop PLAY the letters A, B, and C only. (VALID will be initialized to [A B C].) We also want to prevent an attempt to remove a ring from an empty peg. If an error is made, we will cause an alarm to be sounded. (See the listing for definitions of the utilities.)

```
TO PLAY
L1:
MAKE "X RC
IF :X = "Q THEN STOP
IF NOT MEMBER? :X :VALID THEN
ALARM GO "L1
IF EMPTY? THING :X THEN ALARM
GO "L1
GETRING :X
L2:
MAKE "X RC
IF NOT MEMBER? :X :VALID THEN
ALARM GO "L2
SETRING :X
PLAY
END
```

In this procedure, note that the value of X, :X, is the name of a peg, either A, B, or C. One might expect that the value of :X would be denoted : :X, but this denotes the value of ':X'. The primitive THING must be used. THING :X is the list named by :X.

In order to discuss GETRING and SETRING, we need to be specific about how to represent the graphics. We could use the turtle, but we choose tiles because this allows the most colorful display. The LOGO screen is divided into 32 columns numbered 0 to 31 from left to right, and 24 rows numbered 0 to 23 from top to bottom. We can place the rings on the display by locating them relative to their



pegstands. Let ABASE, BBASE, and CBASE name the coordinates for the centers of the pegstands. Reasonable choices are :ABASE = [7 21], :BBASE = [25 21], and :CBASE = [16 11]. Suppose a ring is the top one on a given peg. Its center has as its column coordinate the same column coordinate as the peg, and its row coordinate is equal to the row coordinate of the base minus as many rings as are on the peg. If we use TOP, COL, and ROW to contain the number of the top ring and its column and row coordinates respectively, we are led to:

```
TO SETRING :P
MAKE :P SE :TOP THING :P
MAKE "BCOORD THING WORD :P "BA
SE
MAKE "COL FIRST :BCOORD
MAKE "K COUNT THING :P
MAKE "ROW ( LAST :BCOORD ) - :
K
DISPLAYRING
END

TO GETRING :P
MAKE "BCOORD THING WORD :P "BA
SE
MAKE "TOP FIRST THING :P
MAKE "COL FIRST :BCOORD
MAKE "K COUNT THING :P
MAKE "ROW ( LAST :BCOORD ) - :
K
ERASERING
MAKE :P BF THING :P
END
```

In using these procedures, :P is a letter (A, B, or C). Thus WORD :P "BASE will return the word ABASE, BBASE or CBASE. Note how BF (BUTFIRST) and SE (SENTENCE) are used to change the value of :P (which will equal A, B, or C). By passing the name of the peg, we can change its value. This would not be the case if we passed the value of the peg to the procedure. (Computer scientists call this passing parameters "by reference" rather than "by value.")

We are left with the problem of actually displaying the pegs and displaying and removing the rings. The work will be done by STAND, DISPLAYRING, and ERASERING. We need to choose the tiles and colors.

The bases will use tile 96 and be black. The pegs will use tiles 104 and 105, and be white. Tile 104 is square, and tile 105 is rounded at the top. Recall that the number of rings is :N, and the division in LOGO is integer division.

```
TO STAND :P
MAKE "BCOORD THING WORD :P "BA
SE
MAKE "COL FIRST :BCOORD
MAKE "ROW LAST :BCOORD
MAKE "J :COL - :N / 2
REPEAT 1 + 2 * ( :N / 2 ) [PT
96 :J :ROW MAKE "J :J + 1 ]
MAKE "K :ROW - 1
REPEAT :N [PT 104 :COL :K MAKE
"K :K - 1 ]
PT 105 :COL :K
PT CHARNUM :P :COL :ROW + 1
END
```

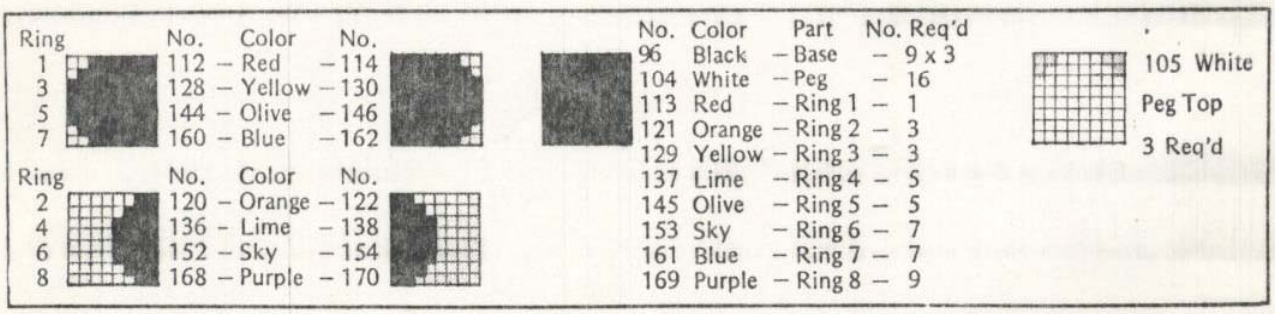
Tiles and colors for the rings will be chosen as follows: The shapes for the tiles are designed so that ring k appears to be k + 2 tiles wide, but it is actually 3 + 2*(k/2) tiles wide. The accompanying figure shows the number and shape of all the required tiles, which we will have to make using MAKECHAR.

Ring	Tiles	Color	Tiles wide
1	112,113,114	Red	3
2	120,121,122	Orange	5
2	128,129,130	Yellow	5
4	136,137,138	Lime	7
5	144,145,146	Olive	7
6	152,153,154	Sky	9
7	160,161,162	Blue	9
8	168,169,170	Purple	11

A ring appears when the right number of tiles of the right shape and color are displayed. A ring is erased by displaying blanks and the peg tile. For effect, the rings will be displayed from the center out and erased from the outside in.

```
TO ERASERING
MAKE "J 1 + :TOP / 2
REPEAT 1 + :TOP / 2 [PT 32 :CO
L - :J :ROW PT 32 :COL + :J :R
OW MAKE "J :J - 1 ]
PT 104 :COL :ROW
END

TO DISPLAYRING
MAKE "LT 104 + :TOP * 8
MAKE "MID 105 + :TOP * 8
MAKE "RT 106 + :TOP * 8
PT :MID :COL :ROW
MAKE "J 1
REPEAT :TOP / 2 [PT :MID :COL
- :J :ROW PT :MID :COL + :J :RO
W MAKE "J :J + 1 ]
PT :LT :COL - :TOP / 2 - 1 :RO
W
PT :RT :COL + :TOP / 2 + 1 :RO
W
END
```

We are almost ready to play with the puzzle. INITIALIZE (see listing) defines colors for the tiles, and assigns values to N, VALID, ABASE, BBASE and CBASE.



Before anything will happen, though, the tiles must be defined using MAKECHAR. (See figures.) Then, ENJOY! Recall that to manipulate the rings, you just need to press the letter of the peg from which you want to take, or to which you want to add a ring. Use the procedure HELP if you forget.

After you have had some fun with the puzzle, you might want to try a four peg variation. To implement a four peg version, do the following:
Change INITIALIZE to include:

```
MAKE "VALID [ A B C D ]
MAKE "ABASE [ 8 10 ]
MAKE "BBASE [ 24 10 ]
MAKE "CBASE [ 8 23 ]
MAKE "DBASE [ 24 23 ]
```

In SETUP, add:

```
MAKE "D [ ]
STAND "D
```

The puzzle should then contain four pegs: A, B, C, and D. It can be manipulated just like the three peg puzzle. The automatic solution will still use just three pegs. But as a worthy challenge, you might try to write a better version of SOLVE which takes advantage of the fact that there are two auxiliary pegs instead of just one. The puzzle should take fewer moves to solve. How many less than 2n − 1 moves are required if there are n rings and four pegs? I would be interested in any of your results. Then can five pegs be fit on the screen. . . ?

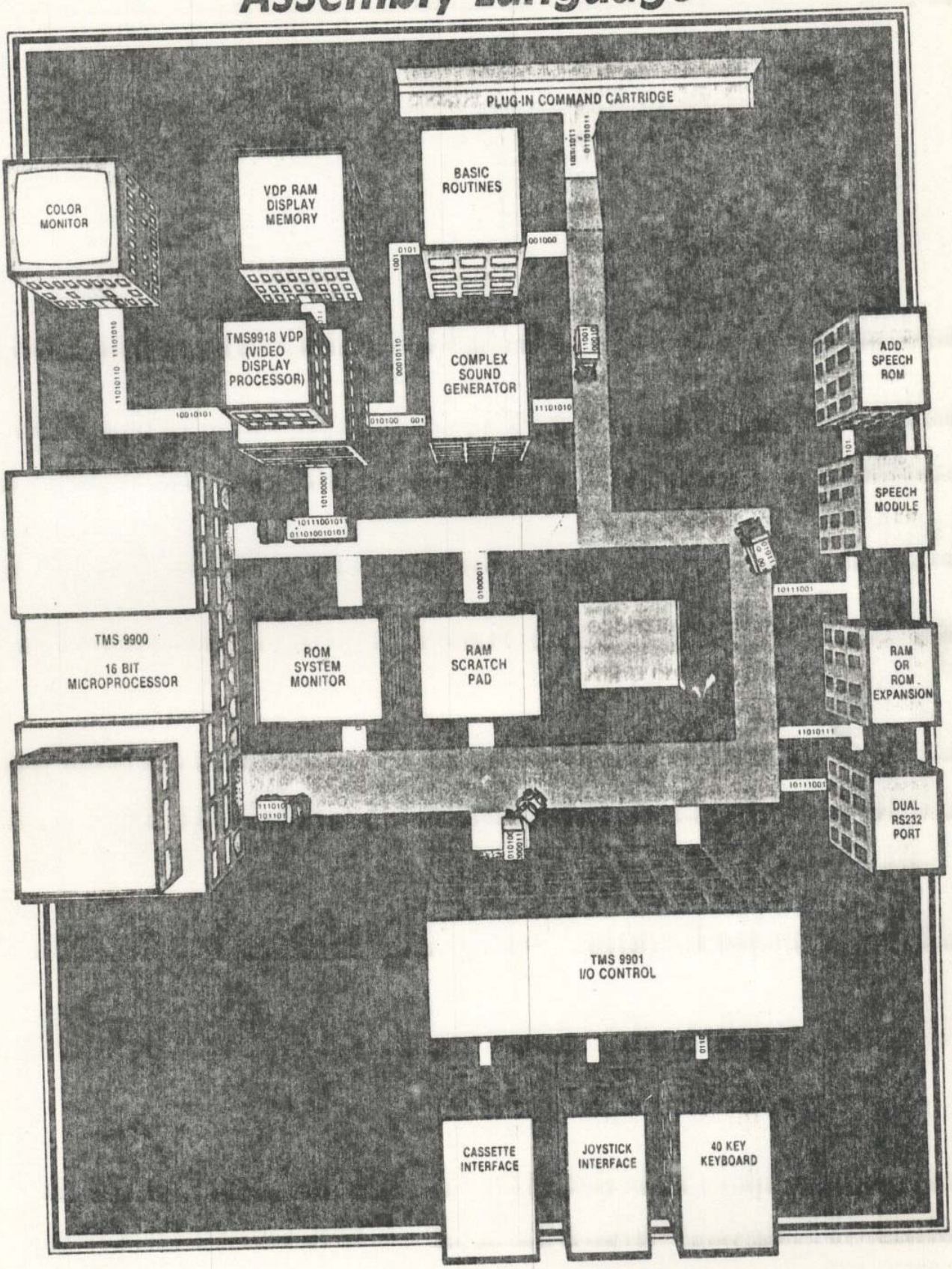
But if you are looking for a lesser challenge, or just want to experiment with a simpler puzzle, note that the number of rings is set in INITIALIZE and can be changed. Try this: Enter INITIALIZE, and then MAKE "N 5 (or some other integer). If you now enter SETUP, a puzzle with 5 rings will be displayed. Enter PLAY, and you can manipulate this puzzle until you press Q. Now enter SETUP again, and then SOLVE 4 "A "C "B. This will cause four rings to be moved automatically to peg C. Then enter PLAY and you can complete the puzzle by yourself. With LOGO, the procedures are your own to do with or modify as you please. Use your imagination, make up other puzzles, or just go ahead and play with this section's puzzle as is.

```
TO HANOI
INITIALIZE
SETUP
PLAY
SETUP
SOLVE 8 "A "B "C
END

TO INITIALIZE
TELL TILE 96 SC :BLACK
TELL TILE 104 SC :WHITE
TELL TILE 112 SC :RED
TELL TILE 120 SC :ORANGE
TELL TILE 128 SC :YELLOW
TELL TILE 136 SC :LIME
TELL TILE 144 SC :OLIVE
TELL TILE 152 SC :SKY
TELL TILE 160 SC :BLUE
TELL TILE 168 SC :PURPLE
MAKE "N 8
MAKE "VALID [A B C]
MAKE "ABASE [7 21]
MAKE "BBASE [25 21]
MAKE "CBASE [16 11]
END

TO PLAY
L1:
MAKE "X RC
IF :X = "Q THEN STOP
IF NOT MEMBER? :X :VALID THEN
ALARM GO "L1
IF EMPTY? THING :X THEN ALARM
GO "L1
GETRING :X
L2:
MAKE "X RC
IF NOT MEMBER? :X :VALID THEN
ALARM GO "L2
SETRING :X
PLAY
END

TO SETUP
CS
STAND "A
STAND "B
STAND "C
MAKE "TOP :N
MAKE "A []
MAKE "B []
MAKE "C []
REPEAT :N [SETRING "A MAKE "TO
P :TOP - 1 ]
END
```

```
TO ALARM
BEEP
WAIT 30
NOBEEP
END

TO MEMBER? :X :LIST
IF :LIST = [] THEN OUTPUT "FA
LSE
IF :X = FIRST :LIST THEN OUTPU
T "TRUE
OUTPUT MEMBER? :X BF :LIST
END

TO HELP
CS
PRINT [TYPE "HANOI" TO BEGIN.
]
PRINT []
PRINT [PUSH A, B, OR C TO REMO
VE OR SET DOWN A RING. ]
PRINT []
PRINT [TO QUIT, AND WATCH THE
PUZZLE SOLVED AUTOMATICALLY, P
USH Q. ]
END

TO STAND :P
MAKE "BCOORD THING WORD :P "BA
SE
MAKE "COL FIRST :BCOORD
MAKE "ROW LAST :BCOORD
MAKE "J :COL - :N / 2
REPEAT 1 + 2 * ( :N / 2 ) [PT
96 :J :ROW MAKE "J :J + 1 ]
MAKE "K :ROW - 1
REPEAT :N [PT 104 :COL :K MAKE
"K :K - 1 ]
PT 105 :COL :K
PT CHARNUM :P :COL :ROW + 1
END

TO SETRING :P
MAKE :P SE :TOP THING :P
MAKE "BCOORD THING WORD :P "BA
SE
MAKE "COL FIRST :BCOORD
MAKE "K COUNT THING :P
MAKE "ROW ( LAST :BCOORD ) - :
K
DISPLAYRING
END
```

```
TO SOLVE :N :P1 :P2 :P3
IF :N = 1 THEN GETRING :P1 SET
RING :P2 STOP
SOLVE :N - 1 :P1 :P3 :P2
GETRING :P1 SETRING :P2
SOLVE :N - 1 :P3 :P2 :P1
END

TO DISPLAYRING
MAKE "LT 104 + :TOP * 8
MAKE "MID 105 + :TOP * 8
MAKE "RT 106 + :TOP * 8
PT :MID :COL :ROW
MAKE "J 1
REPEAT :TOP / 2 [PT :MID :COL
- :J :ROW PT :MID :COL + :J :R
OW MAKE "J :J + 1 ]
PT :LT :COL - :TOP / 2 - 1 :RO
W
PT :RT :COL + :TOP / 2 + 1 :RO
W
END

TO ERASERING
MAKE "J 1 + :TOP / 2
REPEAT 1 + :TOP / 2 [PT 32 :CO
L - :J :ROW PT 32 :COL + :J :R
OW MAKE "J :J - 1 ]
PT 104 :COL :ROW
END

TO COUNT :LIST
IF :LIST = [] THEN OUTPUT 0 E
LSE OUTPUT 1 + COUNT BF :LIST
END

TO GETRING :P
MAKE "BCOORD THING WORD :P "BA
SE
MAKE "TOP FIRST THING :P
MAKE "COL FIRST :BCOORD
MAKE "K COUNT THING :P
MAKE "ROW ( LAST :BCOORD ) - :
K
ERASERING
MAKE :P BF THING :P
END

TO EMPTY? :LIST
IF :LIST = [] THEN OUTPUT "TR
UE ELSE OUTPUT "FALSE
END
```

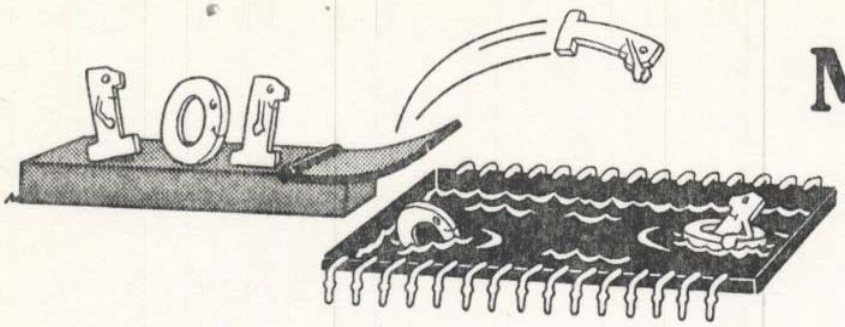
Assembly Language



5

Assembly Language

Faster than a speeding cursor! More powerful than Extended BASIC!—It's SUPER LANGUAGE!



TMS9900 Machine & Assembly Language

PART 1: Electrical Signals, Number Systems & CPU Architecture

If you're a reader of *99'er Home Computer Magazine*, you are probably aware that there is a difference between 8-bit and 16-bit computers . . . although just exactly *what* that difference is—other than "16 bits are twice as many as 8 bits"—might not be that obvious. My purpose in this series of articles is, therefore, to discuss the inner workings of your 16-bit computer by gradually introducing you to its operation and low-level programming in a language much closer to the way your computer operates without any BASIC interpreter slowing things down, or coming between you and the power of your machine.

The heart of any computer is its microprocessor, and the one we'll be examining is, naturally enough, the Texas Instruments TMS9900—the 16-bit chip around which this magazine is organized. To understand its operation, we first have to know something about electrical signals and number systems, so let's begin our discussion here.

Clocks, Pulses, Bits & Bytes

The electrical signals used by a computer are labeled high and low, or 1 and 0, respectively. One of these signals is called a *bit*. Inside the computer this corresponds to one wire. All of the wires together are called a *bus*. The computer reads and writes a part of the bus called the *data bus* at specific intervals, which are regulated by a *clock*. The signals that the clock produces to tell the computer when to read and write are called *clock pulses*.

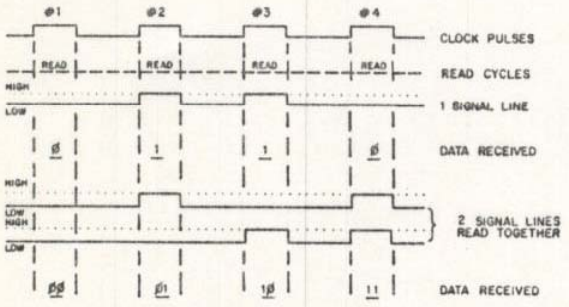


Chart 1

At each pulse of the clock, the computer reads a group of lines. Your normal, run-of-the-mill microcomputer uses groups of 4,8,or 16 bits. All the information read or written is called *data*. If the computer is reading or writing on 1 line, the data is called *serial*. If it is reading or writing on a group of lines together, the data is called *parallel*. 4 bits in parallel are called a *nybble*; 8 are a *byte*; and 16 has no name, but I propose to call it a *gobbyl*.

Look at Chart 1. The top line is the clock. In this example when the pulse is high, the computer reads the signal lines. Notice that when there is only one signal line, the data received can be only a 1 (when the line is high) or a 0 (when the line is low). There are only two possible codes you could see during one clock pulse. You would see a 1 or a 0.

Now look at what happens when you have two signal lines grouped together: 4 different codes are possible. On clock pulse #1 both lines are low (code 00); on pulse #2 the bottom line is low and the top one is high (code 01); pulse #3 has the bottom high and the top low (code10); and pulse #4 has both lines high (code 11).

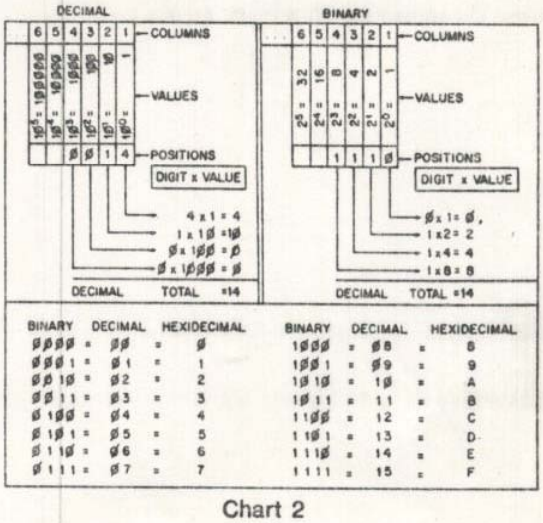


Chart 2

Number Systems

These codes could also be considered numbers. Counting with only 0's and 1's is called *binary* (from the Latin word for two) or *base two* counting. Ordinary, plain, vanilla numbers that we use everyday that are called *decimal* (from the Latin for ten, of course) or *base ten* numbers. Even though we have only the ten digits from 0 to 9, we can make very large numbers by using the same digits in different positions. Follow along on chart 2.

The position on the extreme right in a decimal number is the ones column. For that matter, the position on the extreme right in any base is the ones column. Why? Because you find the column value by taking the number of digits you have and raising it to the power of column minus one. For example, if you have ten digits, and the column is number 1 (from the right), then the value of that column is 10 to the 1 minus 1, or 10 to the 0 power. Any number to the 0 power is 1, so the first column is always ones in any base.

The second column is a different matter. In base ten it is 10 to the 2 minus 1, or 10 to the 1st power, or 10. So if you write 14 what you mean is 4 groups of ones and 1 group of tens. In base two the second column (from the right) would be 2 to the 2 minus 1, or 2 to the 1st, or 2. The second column or position in binary is the twos column.

The thing that makes the zero so neat is that it holds the position without giving it a value. Zero ones is zero! If you just left a blank there, people would have to write all their numbers in little boxes or pretty soon the columns would get all jumbled up. Is there one blank, or two? . . .or three?? Better use the zero.

The columns in binary numbers are just like the signal lines in a computer. In theory, the columns go on forever—and so do the numbers. Regardless of the base you are in, you can keep writing numbers forever! But wait! I just said that signal lines are usually groups of 4, 8, or 16. If signal lines are the same as columns, then there is a limit to the size of number a computer can understand. How big is the biggest number you can use?

To find out, raise the base to the same power as the number of positions you have. On chart 1 when we used two lines, that was 2 to the 2nd power, or 4 codes or numbers. With 4 lines, there are 16 (2 to the 4th); with 8 there are 256; and with 16 lines there are 65536.

The last code on the chart is 1111, which in decimal is 15. I said you could get 16 numbers with four lines, so where is the last number? Don't forget to count 0! 0 through 15 is sixteen numbers, 0 through 255 is 256 numbers, and so on.

There are other bases, or course. The numbers marked *hexadecimal* are from a base with 16 digits—the normal 10 digits from 0 to 9, plus the letters A to F. Use them just like any other digits. For instance, on the chart, 1111 binary is 15 decimal and F in hexadecimal (hex for short). The next number in hex is 10; in decimal it is 16; and in binary you have to add a new position (sixteens) and write 10000.

You can always add as many zeros to the front of a number as you want without changing it. However, if you make a binary number divisible into groups of four, an interesting thing happens: Each group of four can represent 16 codes or numbers. Since that is exactly the number of digits in the hex number system, you can substitute! This makes long binary numbers much easier to read, and doesn't change their values at all.

Try a few yourself. They're easy!

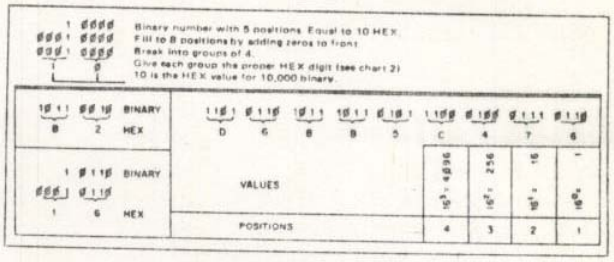


Chart 3

Hardware

The TMS9900 is called a 16-bit *CPU (Central Processing Unit)*. This means that when it fetches an instruction from memory, it gets 16 bits in parallel. And when it reads or writes data this is usually done in groups of 16 bits too. [In the TI-99/4A, however, this 16-bit group is converted into an 8-bit data bus.—Ed.] You may hear the term *word* used for 16 bits. If you are talking about a 16-bit machine, the term is correct. But remember, if you are talking about an 8-bit CPU, 8 bits (or byte) is a word; if the CPU is 32 bits, the word is 32 bits.

It is necessary for a programmer to know about only two kinds of memory. *Random-access memory (RAM)*, sometimes called *read/write memory*, is what stores the user's program, data, etc. The user or the computer can read or write in it. The memory location is chosen by the lines on the bus called *address lines*. The data that is being read or written appears on the data bus.

Read-only memory (ROM) comes in many varieties and works just like RAM except for one thing—it can't be written to. If you tell the computer to write, it will go through the motions of writing, but it doesn't work. The old data is still there.

Inside the CPU there are a few memory locations that are not addressed by the address bus. The chip itself knows where they are. These are called *registers*. All *machine language* and *assembly language* programming involves manipulating the data in these registers, because that is all that the computer really can do!

How many registers there are and how big they are varies widely. The chip manufacturer usually labels the registers and decides on a short code, called an *operation code* (op-code), for each of the manipulations that the chip can do. An *assembler* is a program that reads these op-codes and writes them into memory in the binary form that the CPU understands. When you write a program using the op-codes, you are writing in *assembly language*. If you write your own assembler you can devise your own op-codes. But because the manufacturer generally writes an assembler for his chip, you can use his op-codes.

About the only thing all CPUs have in common is a register called the *Program Counter* (PC). The address bus is just an extension of the PC. Each bit of the program counter is, in effect, connected to one signal line of the address bus. Since the TMS9900 chip was designed especially for dedicated control purposes (e.g., production lines inspection or phone switching) where the pro-

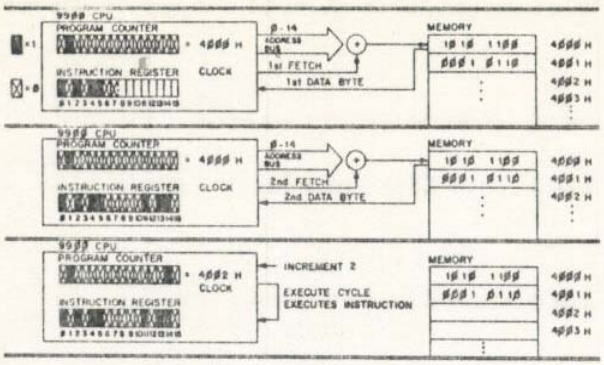


Chart 4

gram is always in ROM—and since at the time most ROMs were made for 8-bit computers—the address bus of the 9900 is a little unusual.

The bits of the PC allow the chip to address 65536 blocks of memory. The blocks could be any size, but as I said, most ROMs were in blocks of 8 because most computers had an 8-bit data bus. The PC in the 9900 has 16 bits. These are labeled 0-15, from (left to right), *most significant bit (MSB)* to *least significant bit (LSB)*. Why are there only 15 address lines? Follow on Chart 4 as we go along.

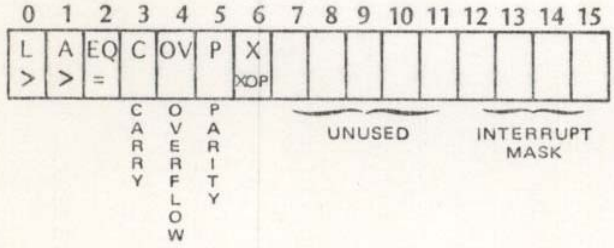
Normally the PC advances after each instruction or parameter it fetches so that it points to the next memory byte. But the 9900 needs 16 bits instead of the 8 available at each location in most ROMs. So the 9900 has two different fetch cycles: it reads the byte indicated by the PC on the first cycle, hooks the next byte to it on the second cycle, then increments the PC by two. To the user this all appears as one fetch, except that the PC is incremented by *two* instead of by *one* as expected. By eliminating the last bit, however, the address line appears to step normally. The drawback is that you can address only 32767 words. It's still 65536 bytes though.

PART 2: Registers, Programming & The Need For Assemblers

Status Register

Almost every CPU has some kind of *flag*(s). These are set (high) and reset (low) by actions performed in the manipulations of data. Different instructions affect different flags. Modern CPUs combine several flags into a single *Status Register*. The TMS9900 is no exception. Its Status Register (ST) is 16 bits long. Bits 7-11 are not used at present. The others are shown in the drawing below and are explained in the text.

TMS9900 STATUS REGISTER



Each of these conditions will be discussed in more detail as examples are shown. Until then, these simple descriptions will help.

The four bits labeled 12-15 can select up to 16 interrupt levels. All levels equal to or above the level indicated are enabled.

Bit 0 is set after any operation where the destination value (answer) is greater than the source (the first operand used; it remains unchanged). All 16 bits are used for the comparison.

Bit 1 is similar to bit 0 except that the values are compared as signed integers. The MSB (most significant bit) designates the sign of the integer, with a 1 meaning negative and a 0 meaning positive. The range is +32,767 to −32,768. Negative numbers are represented in a two's complement fashion.

Computer math is cyclic. This means that if you add 1 to the highest possible 16-bit number (FFFF hex), you go back to 0000 hex with a *carry* bit that is set. If you subtract 1 from 0000 hex without the carry, you get an *overflow*; but if the carry is set, you get FFFF hex. Therefore, −1 is FFFF hex in two's complement. To see its usefulness, let's add −1 and 1: FFFF hex plus 0001 hex equal 0000, the carry is set, and the answer is zero. In a nutshell, this whole business of two's complements and carry bits is simply a way to subtract by adding.

Bit 2 is set if the two operands are equal.

Bit 3 is set if a 1 is shifted out of an operand, or if a carry occurs in a math operation.

Bit 4 is set if the math requested cannot be done.

Bit 5 is set if the parity is odd, and reset if it is even. Odd parity means that there is an odd number of 1s in the binary representation of an operand.

Bit 6 is set after an *extended operation* has been completed. This is done because an interrupt is not checked for after completion of an extended operation. (You therefore may wish to have the software check for one if this flag is set).

The ALU

Most CPUs have an *Arithmetic/Logic Unit (ALU)* where the simple math is performed. An *accumulator*, a special register used by the ALU, usually contains the answers to the math. In the TMS9900 there is no accumulator because the destination address serves as the equivalent of an accumulator. This means, in effect, that any memory location *can* be the accumulator. There is an ALU on the TMS9900 chip, but its operation is intrinsic to the instructions.

Other Registers

Most CPUs have a few extra registers where quickly-needed values can be stored, as well as a register called a *Stack Pointer* which points to a section of memory where more data can be "piled" and then quickly accessed. These two concepts have been combined on the TMS9900 into a single *Workspace Pointer Register (WP)*. The WP points to a block of 32 bytes of the memory arranged as 16 workspaces (WS), each 16 bits long. The workspaces are synonymous with registers, and are used the same way. We can change the WP in several ways and can save the old WP when a new one is used. This allows us to return to the old one if we need to. This set-up, in effect, acts like an elaborate *stack*.

There are five different ways to use these WP registers to indicate an operand for an instruction. These *addressing modes* are as follows:

1. Workspace Register Mode code 00 —the data in the indicated register is the data used.
2. Workspace Register Indirect code 01 —the data in the register is treated as the address of the real data.
3. WS Register Indirect w/Auto-Increment code 11 —same as above, but the register is incremented upon completion.
4. Symbolic or Direct code 10 —the address of the data follows the instruction in memory.
5. Indexed code 10 Td or Ts equal 1-15 —same as above, but the value in the index register is added to the address.

There are three other addressing modes not dealing with registers *per se:* (1) The *immediate mode* has the data immediately follow the instruction code. In other words, the address of the data is the address immediately following the PC. (2) The *CRU mode* has the address of an external input/output (I/O) device determined by bytes 3-12 of register 12. (3) The *JMP instruction* (and all variations thereof) uses the last 8 bits of the instruction to determine where on a 256 byte page to jump. The PC indicates the center of the page, so the jump can be from PC – 128 to PC + 127. One byte is taken up by the jump instruction itself. The 8 bits store the relative jump in two's complement form.

Programming and the Need for Assemblers

If your CPU is the TMS9900, the simplest computer you could construct would be composed of a clock, a CPU, some memory, a few control switches, 16 data switches, 16 lights for read out, and 15 address switches. It would be crude and slow to program, but once programmed, it would operate as well as any other computer. But how could we program it?

Suppose we wanted to load register 1 with zero, and then increment it until its contents were equal to either 1024 (decimal) or the contents of register 2. The first step can be done several ways. Immediately loading register 1 with 0 comes to mind first. A little investigation of the instructions for the chip show that we could save a word of memory by using the *Clear* command. Figure 1 shows the register format for the various commands, and Figure 2 shows the *op codes* for the instructions.

Using this information, we can now determine the binary values of each word. Load Immediate uses the first 10 bits as the op code; the 11th bit is not used; and bits 12-15 select the register. This means the first word is

000000010000X0001, where X can be 1 or 0.

The second word is the value to load, and in this case would be all zeros.

Using our simplified computer, just flip each switch on if there is a 1 at the corresponding bit, off if there is a zero. Press the *Input* control switch (it might be called *Load*, or . . .), and the instruction is stored in whatever address the address switches are set to. Then add 1 to the address switches (which adds 2 to the PC) and set all the data switches to zero. Press *Input* again, and our complete instruction is ready.

FORMAT	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	OP CODE			B	T_d		D				T_s		S			
2	OP CODE										RELATIVE JUMP					
3	OP CODE						D				T_s		S			
4	OP CODE						C				T_s		S			
5	OP CODE								C				W			
6	OP CODE										T_s		S			
7	OP CODE												N			
8	OP CODE										N		W			
9	IMMEDIATE VALUE															
	OP CODE						D				T_s		S			

KEY

B 1=byte 0=word
T_d destination address mode
D destination address
T_s source address mode
S source address
C counter
W register number
N unused
RELATIVE JUMP from +127 to −128

T_d/T_s FIELD CODES

00 Register
01 Indirect
10 with R0, symbolic
10 with R1–R15, indexed
11 Indirect with increment

Figure 1.

If instead, we use the *Clear* instruction, we would use the single-operand general format with the first 10 bits being the op code. The next two bits indicate address mode, and the last 4 bits select the register. Since we want to clear the register itself (not the word it points to), the code is 00,and the whole instruction is 0000010011000001.

Even with a hex keypad and a small monitor program, it would be a very time-consuming process to piece together the binary words, and then convert to hex and type them in. Typing in 04C1 is easier than setting switches to

0000010011000001,

but putting together those op codes is just the tedious, boring kind of work that computers are supposed to free us of. So why not use them for that?

Why not, indeed. . .That's exactly what we'll do when we look at a TMS9900 assembler.

Figure 2.

Mnemonic	Op Code	Format	Status	Bits Affected	Meaning
A	1010	1	0–4		Add words
AB	1011	1	0–5		Add bytes
ABS	0000011101	6	0–4		Absolute Value
AI	00000010001	8	0–4		Add Immediate
ANDI	00000010010	8	0–2		And immediate
B	0000010001	6	- - -		Branch
BL	0000011010	6	- - -		Branch and Link (R11)
BLWP	0000010000	6	- - -		Branch, load WP
C	1000	1	0–2		Compare words
CB	1001	1	0–2, 5		Compare byte
CI	00000010100	8	0–2		Compare immediate
CKOF	0000001111000000	7	- - -		External Control
CKON	0000001110100000	7	- - -		External Control
CLR	0000010011	6	- - -		Clear
COC	001000	3	2		Compare Ones Corresp. (OR)
CZC	001001	3	2		Compare Zero Corresp. (AND)
DEC	0000011000	6	0–4		Decrement by one
DECT	0000011001	6	0–4		Decrement by two
DIV	001111	9	4		Divide
IDLE	0000001101000000	7	- - -		Computer idles
INC	0000010110	6	0–4		Increment by one
INCT	0000010111	6	0–4		Increment by two
INV	0000010101	6	0–2		Invert (complement)
JEQ	00010011	2	- - -	(ST2=1)	Jump if equal
JGT	00010101	2	- - -	(ST1=1)	Jump greater than
JH	00011011	2	- - -	(ST0 and ST2=1)	Jump high
JHE	00010100	2	- - -	(ST0 or ST2=1)	Jump high or equal
JL	00011010	2	- - -	(ST0 and ST2=0)	Jump low
JLE	00010010	2	- - -	(ST0=0 or ST2=1	Jump low or equal
JLT	00010001	2	- - -	(ST1 and ST2=0)	Jump less then
JMP	00010000	2	- - -	(none checked)	Jump unconditionally
JNC	00010111	2	- - -	(ST3=0)	Jump no carry
JNE	00010110	2	- - -	(ST2=0)	Jump not equal
JNO	00011001	2	- - -	(ST4=0)	Jump no overflow
JOC	00011000	2	- - -	(ST3=1)	Jump on carry
JOP	00011100	2	- - -	(ST5=1)	Jump odd parity
LDCR	001100	4	0–2, 5		Load CRU
LI	00000010000	8	0–2		Load immediate
LIMI	00000011000	8	12–15		Load immed. INT mask
LREX	0000001111100000	7	12–15		External control
LWPI	00000010111	8	- - -		Load immed. WP
MOV	1100	1	0–2		Move word
MOVB	1101	1	0–2, 5		Move byte
MPY	001110	9	- - -		Multiply
NEG	0000010100	6	0–4		Negate (2's comp.)
ORI	00000010011	8	0–2		OR immediate
RSET	0000001101100000	7	12–15		External control
RTWP	0000001110000000	7	0–6, 12–15		Return with WP
S	0110	1	0–4		Subtract word
SB	0111	1	0–5		Subtract byte
SBO	00011101	2	- - -		Set CRU bit to one
SBZ	00011110	2	- - -		Set CRU bit to zero
SETO	0000011100	6	- - -		Set ones
SLA	00001010	5	0–4		Shift left (0 fill)
SOC	1110	1	0–2	Words (OR)	Set ones corresp.
SOCB	1111	1	0–2, 5	Bytes (OR)	Set ones corresp.
SRA	00001000	5	0–3		Shift right (MSB fill)
SRC	00001011	5	0–3		Shift right circular
SRL	00001001	5	0–3		Shift right zero fill
STCR	001101	4	0–2, 5		Store from CRU
STST	00000010110	8	- - -		Store ST
STWP	00000010101	8	- - -		Store WP
SWPB	0000011011	6	- - -		Swap bytes
SZC	0100	1	0–2	Words (AND)	Set zero corresp.
SZCB	0101	1	0–2, 5	Byte (AND)	Set zero corresp.
TB	00011111	2	2		Test CRU bit
X	0000010010	6	- - -		Execute
XOP	001011	9	6		Extended operation
XOR	001010	3	0–2		Exclusive OR

IT'S SUPER LANGUAGE

PART 1: Fundamentals of Assembly Language Programming on the TI-99/4A

Before getting into the details of the TI-99/4A Editor/Assembler package, we should first consider what an assembler is and what it can do for us. Most readers are already familiar with the TI BASIC language, and many have already experienced the disk-oriented features of Extended BASIC. These BASICs are *interpreted* languages. When a BASIC program is being run, the BASIC interpreter converts (interprets) the BASIC statements, one statement at a time, into *machine language*—the binary ones and zeros that the computer understands. It then executes the statement it has just converted. Since a single BASIC statement usually generates several machine instructions, programs can execute relatively slowly. This is especially true in programs containing loops because each statement in a loop is interpreted each time it is encountered.

BASIC programs are simply input and RUN, but programming in assembly language involves an extra step which is not apparent in BASIC programming—namely the *assembler stage*: Assembly language programs must be input, then assembled and finally RUN. The assembler converts the assembly language statements (or *source* program) to machine language; it is the machine-language (or *object*) program which is RUN. Because there is no waiting for each statement to be interpreted at runtime, programs written in assembly language run extremely fast.

Another major difference between BASIC and assembly language is the difficulty of writing programs. A BASIC program is relatively easy to code because the instructions are English-like and the programmer does not have to worry about where variables reside in memory or have to understand the structure of the machine. Assembly language programs, on the other hand, are harder and more time-consuming to write because the instructions are machine-oriented (see "TMS9900 Machine and Assembly Language") and the programmer must understand the structure of the machine. Debugging assembly language programs is harder, too. But these difficulties are not necessarily disadvantages, because an understanding of the machine allows a programmer to create more efficient programs. Programming in assembly language is an education in itself, and is one of the best ways to learn how a computer works.

A programmer must consider these tradeoffs in choosing the best language for each application. In general, BASIC is faster to write and debug, but assembly language programs execute faster. Happily, TI has made it possible to choose *both* by enabling Extended BASIC programs to CALL assembly language subroutines. This means that a programmer can write mainly in Extended BASIC and use assembly language for portions of the program where faster execution is required (loops, and especially, sorts). Writing short assembly language subroutines to CALL from Extended BASIC programs is a good way to ease into assembly language programming, and after some practice you may find yourself writing entire applications in assembly language.

What follows is a preliminary look at the TI-99/4A Editor/Assembler package. It is, however, only an *overview* of the product. Other sections will go into more depth on specific features of the software.

Software Media and Required Hardware

The Editor/Assembler software resides in a Command Cartridge and on a disk. To run it, you'll need at least one

disk drive and the 32K expansion RAM. Both the Editor and the Assembler are selectable from menus, and most of the screens include easy-to-understand prompting messages.

The Editor

The Editor is used to input Assembly Language source programs initially, to update programs previously saved on disk and to print programs. The Editor's features compare favorably to those of larger systems.

There are two modes: Edit Mode and Command Mode. Edit Mode is always used to input a program for the first time, but either mode can be used to change existing programs after loading them from the disk or typing them in Edit Mode.

Edit Mode is entered directly from the menu. The screen is a 40 × 24 window on the source program. Function keys allow you to move this window to the right or left in 20-character increments, or up and down 24 lines at a time. (Since most of my Assembly Language programs have fewer than 40 characters per line, I tend to view the leftmost 40 characters and make heavy use of the up and down scrolling). The four cursor keys are enabled in Edit Mode, making it especially easy to correct typographical errors. Whole lines can be inserted into the text by moving the cursor to the adjacent line and pressing the Insert function key; a new blank line is inserted, and the user simply types in a new line. Similarly, a whole line can be deleted by moving the cursor there and pressing the Delete function key; the line is removed and the line numbers of the following lines are automatically decremented. There are also keys for inserting or deleting characters. A Tab key is also provided for tabbing to columns 8 and 16. Edit Mode makes it very easy to enter new programs because the user can both type the source program in a natural manner and correct errors and omissions as they occur. Edit Mode is exited via the Back function key, which puts the Editor into Command Mode.

Command Mode reminds me of the UCSD Pascal editor. The first line of the screen shows the Command Mode options: **E**scape, **F**ind, **R**eplace, **M**ove, **I**nsert, **C**opy, **D**elete, **S**how, and **A**djust. Line 2 is reserved for parameters to be input by the user, so in this mode the text window is 40 × 22. Most options require further information to be given on line 2, and very clear prompts given so the user knows what line to enter.

Each option is selected by typing the first character of the option name. For example, to find an occurrence of a string in the source program, the user enters **F**. The system responds with the prompt <count>< (start col, end col)>/string/. To find the second occurrence of the string ABCD between columns 1 and 50, the user would type 2(1,50) /ABCD/. The system would then display the section of the text containing the second such occurrence of ABCD (if any) with the cursor over the A. The symbols <> in the prompting message indicate optional parameters. To find the next occurrence of the string ABCD in the whole source program, the user need only type /ABCD/. The **R**eplace option is like **F**ind, except that each specified occurrence of the string is replaced by a second string given by the user. **R**eplace includes an optional verify operator which allows the user to say yes or no to each replacement. The **M**ove option allows the user to move sections of text, indicated by an interval of line numbers, to a different place in the source program. **C**opy is similar, except that the section of text ends up in both the original position and the new position. **D**elete allows easy removal of several contiguous lines from the text. **I**nsert takes a file from disk and places it anywhere you want in the program being edited. **S**how is a way of moving the window so that a certain line number is at the top of the screen. **A**djust is an easy way to make the line numbers disappear so that the window shows the source program only. **E**scape gets you out of Command Mode and back to the Editor's menu, where you can choose to save the source program to disk, print it, purge it or edit the same or another program.

The Editor performs all line numbering automatically as lines are entered and maintains these numbers in sequence as lines are added or deleted. The user can refer to them for operating on sections of the program; they also appear

```
            Larger system (TXMIRA):
            LI      2,0             MOVE 0 TO REGISTER 2 FOR INDEX
            LI      12,>CO          SET CRU BASE ADDRESS FOR SCREEN
            SBO     >F              SELECT CRU WORD 1
            LDCR    @ZERO,11        MOVE CURSOR TO HOME POSITION
            SBZ     >F              SELECT CRU WORD 0
LOOP        LDCR    @AB(2),7        PUT CHARACTER ON CRU LINE
            SBZ     >8              STROBE CHARACTER TO SCREEN
            SBZ     >A              INCREMENT CURSOR POSITION
            INC     2               ADD 1 TO INDEX REGISTER
            CI      2,2             COMPARE REGISTER 2 TO 2
            JLT     LOOP            LOOP IF MORE CHARACTERS
            ...
ZERO        DATA    0               DATA DEFINITIONS
AB          TEST    'AB'

            TI-99/4A assembler:
            REF     VMBW .          EXTERNAL REFERENCE TO ROUTINE UTILITY
            ...
            LI      0,0             VDP RAM ADDRESS = 0 FOR HOME POSITION
            LI,     1,AB            REGISTER 1 POINTS TO FIRST CHARACTER TO DISPLAY
            LI      2,2             REGISTER 2 = NUMBER OF BYTES TO WRITE
            BLWP    @VMBW           CALL UTILITY ROUTINE TO WRITE STRING
            ...
AB          TEXT    'AB'            DATA DEFINITION
```

Figure 1

on the Assembler output listing, which is handy for debugging.

TI has incorporated most of the features found in editors for larger systems into the 99/4A Editor. In fact, the abilities to edit at the character, line, and group-of-lines levels are not always *all* available in larger editors. The only feature missing from the 99/4A Editor is a variable right margin—a feature which is really not too significant for Assembly Language source programs. [But that would be nice for word processing applications, since this editor already performs 95% of what most people would need for correspondence and document preparation.—Ed.]

The Assembler

The Assembler is a program which converts Assembly Language source programs into object form—the machine-language program that executes on the TI-99/4A. The object program is written to disk. Optionally, a user can print out or write an Assembly Language listing to disk.

The 99/4A Assembler is a lot like the 9900 Assembler, TXMIRA, which runs on larger TI systems. See sample listing in Figure 1. A programmer who is familiar with TXMIRA will be able to write Assembly Language programs for the 99/4A without too much difficulty since the same addressing modes are used and most of the instructions operate in the same way.

One big difference, as might be expected, is in the way a programmer handles input and output to the monitor. The 99/4A Editor/Assembler package includes three groups of built-in subroutines, or *macros*: (1) *Utility Routines* for accessing machine resources, such as screen I/O; (2) *Extended Utilities*, for accessing routines built into the console ROMs and GROMs; and (3) *Basic Support Utilities* for accessing the parameter list in CALL LINK statements from Extended BASIC. These utilities make it unnecessary to use the CRU (Communications Register Unit) lines to the monitor. Under TXMIRA, all peripheral devices are addressed via a fairly complex arrangement of CRU lines. Each device has its own CRU base address and CRU bit assignments, which means that a programmer must have very specific information about each device in order to perform any input or output. On the 99/4A Assembler these difficulties in handling the screen have been eliminated by the Utility Routines. By loading a few registers and invoking the proper utility, a programmer can handle screen I/O in a much simpler way. Figure 1 has the code segments which might be used for writing the character AB to the upper left portion of the screen.

You can see that the Utility Routines really make screen handling easier: You can focus your attention on merely the VDP RAM (the memory associated with the 99/4A monitor) addresses, and not have to worry about the logistics of the move. Furthermore, there is no apparent loss of execution speed in doing it this way.

Another difference between the 99/4A Assembler and those for larger TI computers is that the IDLE instruction is not implemented on the 99/4A. This causes no great difficulty, but it is useful to know. The IDLE instruction just causes the computer to wait for an interrupt; this can be done via another Utility Routine or other means, depending on which device will cause the interrupt.

The optional listing produced by the 99/4A Assembler is quite complete. Statement sequence numbers, source statements, and the hexadecimal code generated are all shown clearly. A symbol table can also be given and, of course, the number of errors is shown. Each error is also flagged in the body of the listing with a descriptive message. One very nice—and all too uncommon—feature is that a display of the number of errors is on the monitor when the Assembler is finished.

Running and Debugging

Once a program has been input, edited, and assembled with no errors, it can be loaded and run by choosing this option from the menu. Another menu option (RUN PROGRAM FILE) allows the user to run programs which were assembled on other Texas Instruments systems or previously assembled on your system.

The Editor/Assembler package has a special debugging utility called DEBUG, which can be very helpful in isolating program errors. For instance, the commands in DEBUG allow you to set *breakpoints* in your program. When the program hits a breakpoint and stops execution, you can then use other commands to examine the contents of memory locations and registers, the Workspace Pointer, the Status Register, or the Program Counter, and if necessary change them to alter the program's execution. DEBUG commands will also allow you to search memory locations for a specific value, or to search memory locations and print those which *don't* have a specific value. DEBUG allows you to begin executing your program at any point you determine; combined with the breakpoints, this allows you to go through a program section by section. All in all, DEBUG provides a good repertoire of useful tools which will make it easier to find out why the program you wrote isn't working the way you thought it would.

PART 2: Fundamentals of Assembly Language Programming on the TI-99/4A

In Part 1 we gave you a preliminary look at TI's Editor/Assembler for the TI-99/4 and TI-99/4A and mentioned briefly the advantages of programming in Assembly Language. Now let's explore the benefits of Assembly Language more fully by comparing some programs written in Assembly Language and BASIC.

Some Assembly Language Explanations

Before examining some programs, it would be useful to mention some general characteristics of the TMS9900 processor, and then some specifics on the structure of the TI-99/4A.

All 9900 programs make use of 16 workspace registers, each containing 16 bits (one word). Assembly Language programs define 16 contiguous words of memory for these workspace registers and set the hardware register called the Workspace Pointer to point to the first of these memory locations. Having these workspace registers resident *in memory* rather than in the CPU is one of the most power-

ful features of the 9900-family processors. In an Assembly Language program, the hexadecimal numbers 0 through F refer to the current workspace registers. (In addition, an Assembly Language option allows you to refer to them as R0 through R15, which makes programs easier to read.)

The structure of the memory of the 99/4A is fairly complex. The following explanations cover concepts necessary to understanding the programs in this article, but they only begin to scratch the surface of the memory structure.

CPU RAM (Random Access Memory) resides in the console and is directly addressable by Assembly Language programs. Workspace registers and other memory locations, as well as the programs themselves, reside in CPU RAM.

VDP (Video Display Processor) RAM, also located in the console, takes care of the video screen. Sprites, colors, character patterns, and the screen image itself all reside in VDP RAM. Unlike CPU RAM, however, VDP RAM is not directly addressable by Assembly Language programs. VDP RAM is accessed through specifically assigned CPU RAM addresses. This is called memory mapping. Locations 0 through >02FF in VDP RAM contain the screen image. (The symbol ">" means hexadecimal notation; >02FF = 767 in decimal notation.) This means that whatever characters reside in this section of VDP RAM are visible on the screen. To change the screen, the programmer would place the desired character code(s) into VDP RAM at the corresponding location(s). VDP RAM location 0 corresponds to the home position (upper left) on the screen; location 48 (or >30) corresponds to the position called row 2 and column 17 in BASIC. Let's say you want to put an * on the screen at row 2, column 17. The ASCII code for * is 42, or >2A, and the desired VDP RAM location is >30. You might be tempted to use a MOVB (Move Byte) instruction to accomplish this, but remember, the VDP RAM cannot be directly addressed from your Assembly Language program. To access VDP RAM, you'll need to use a Utility Routine. VSBW (VDP Single Byte Write) is a *macro instruction* which places the most significant (leftmost) byte of workspace register 1 at the VDP RAM address contained in register 0. Therefore, to place the * at row 2, column 17, you'd write:

```
REF     VSBW      UTILITY REFERENCE
 .
 .
 .
LI      0,>30     R0=VDP RAM ADDRESS
LI      1,>2A00   RI CONTAINS * IN MSB
BLWP    @VSBW     MOVE TO VDP RAM
```

Most of the utilities use similar schemes of loading data into certain registers and calling the utility by name. I'll talk more about some specific ones later.

The Game of Life

Life is a classic computer game. It is based on the idea of a population which goes through life cycles to form new generations; each position on the screen corresponds to a cell in the population. Cells which are alive are filled in (with asterisks in my example); dead cells are blank. The life cycle, or rules of the game, are applied to each generation to obtain the next generation, and then the new generation is displayed on the screen. The rules of the game determine birth, death, or survival of individual cells, and depend on the state of each cell's 8 neighbors (adjoining cells, considered horizontally, vertically, and diagonally) as follows:

1. A live cell with 2 or 3 neighbors survives to the next generation.
2. A live cell with 0 or 1 neighbor dies of loneliness; a live cell with more than 3 neighbors dies of overcrowding.

The rules are applied to a generation as a whole, before the next generation is displayed. Depending on the initial population, you may see a colony which goes on changing forever, one which dies out or becomes static after a few generations, or one which oscillates among a few patterns.

There are a few restrictions on my implementation of *Life* which should be explained. First, I have defined the initial population in the programs, whereas other versions might allow the user to enter the initial population on the screen at the beginning of the game. In order to be sure the colony does not exceed the size of the 99/4A screen, which is 32 × 24, I have forced the border (rows 1 and 24 and columns 1 and 32) always to remain blank. This means that when the colony becomes large it may lose its symmetry as one side of the colony hits the border.

The two programs which follow are in BASIC (Listing 1) and in Assembly Language (Listing 3). Both follow the same strategy: display the initial colony, calculate the next generation by considering the neighbors of each cell in turn, clear the screen, display the new generation, and loop back to calculate the next generation. The Assembly Language version uses one byte to represent each cell; the BASIC version uses one entry in array SCRN for each cell. At the start of each generation, live cells contain the value 1 and dead cells contain 0. During the calculation of the next generation, a cell can have the values 0 through 3 as follows:

0 = cell is dead and remains dead for the next generation
1 = live cell survives to the next generation
2 = dead cell will be born in the next generation
3 = live cell will die in the next generation

It is necessary to have these four possible values during the calculation so that the program can have the information about the current state of each cell while calculating and storing the next state of each cell. Just before the new generation is displayed (or not displayed if dead), the values of the cells are reset to 0 or 1 by means of the array AFTER.

In examining both versions of *Life* which follow (Listings 1 and 3), you might wonder why anyone would use the more esoteric Assembly Language over the easier-to-understand BASIC. The answer is simple: *speed*. On the 99/4A, the BASIC program takes 2 minutes and 26 seconds between generations; the Assembly Language program takes *less than one second*! The BASIC version is no fun at all to watch, whereas the Assembly Language program provides fine entertainment. [The use of the Utility Routine VMBW (VDP Multiple Byte Write) in the Assembly language is partly responsible for this speed. It shows each new generation all at once. And fortunately, the monitor program is smart enough to capitalize on this by showing only the changed portions of the screen, rather than re-drawing the whole screen each time. If fast enough, the human brain's "persistence of vision" allows us to see individual frames of moving images as continuous rather than discrete pictures—thus making realistic animation sequences truly possible.—Ed.]

Using Assembly Language to Move Sprites

The ability to create sprites which move automatically is one of the best features of the 99/4A. Sprites can be used in Extended BASIC and in Assembly Language programs.

VDP RAM has several areas dedicated to sprites. The Sprite Attribute Block, which gives the sprite locations, sprite numbers, and colors, starts at address >300. Each entry in the Sprite Attribute Block occupies four bytes. A terminator byte with value >0D denotes the end of the Sprite Attribute Block. The Sprite Descriptor Block contains the sprite patterns (shapes), with 8 bytes for each possible sprite. Although the Sprite Descriptor Block starts at VDP RAM address 0 by default, we have already seen that VDP RAM locations 0 through >02FF are used for the screen image table, and locations >0300 through >03FF for the sprite Attribute Block. In order to avoid writing over these areas, the Sprite Descriptor Block usually starts at location >0400 for practical purposes. The entries in the Sprite Descriptor Block are defined to correspond to sprite numbers starting at 0 and occupying 8 bytes each; therefore the entry at location >0400 is for sprite number >80. Thus in Assembly Language programs, the lowest sprite number is usually >80. The Sprite Motion Table, which gives the x- and y-velocities of defined sprites, resides at VDP RAM location >0780. Each entry in the Motion Table occupies four bytes, the last two of which are for system use. The Sprite Motion Table is filled only if automatic motion is to be used. An Assembly Language program could move the sprites (non-automatically) by changing the x- and y-locations of the sprites in the Sprite Attribute Block. But the system is able to move the sprites for you via an interrupt processing routine: Each time a VDP interrupt occurs (60 times per second), the interrupt processing routine moves any eligible sprites according to the Sprite Motion Table. In order to make use of this facility, the Assembly Language program must also load the number of moving sprites at CPU RAM address >837A and enable the VDP interrupts.

Assembly Language vs Extended BASIC

You are probably thinking that this sounds like a lot of work to achieve moving sprites, especially compared to the simple CALL SPRITE statement of Extended BASIC. However, there are times when an Extended BASIC program is inadequate. Coincidence checking in Extended BASIC is not as responsive to velocity changes as you might like.

The programs which follow (Listings 2 and 4) illustrate how Assembly Language can be used to overcome these deficiencies. The program simply moves a target from left to right on the screen while shooting an arrow from the top of the screen to the bottom. Both sprites wrap around the screen. Whenever the arrow hits the target, the sprites stop moving, the target changes to an X, and the program delays long enough to make the blow-up visible. Then the program starts over. The Extended BASIC program relies on CALL COINC to detect hits. You'll notice, however, that the program doesn't seem to detect all hits. The Assembly Language program can stop the action by disabling the VDP interrupt while it checks for coincidence by comparing the locations of the arrow and the target from the Sprite Attribute Block. Moreover, the Assembly Language program can check the point of the arrow against the target instead of checking the upper lefthand corners of the sprites.

Because of these differences, the Assembly Language program appears to detect more hits correctly. Of course, this stop-motion processing must slow down the motion, but it is not noticeable to me. (One indication of the speed of Assembly Language program execution is the large number of statements executed in LOOP2 while the hit shape briefly remains on the screen.)

Another shortcoming of the Extended BASIC version is that the hit shape appears quite a bit to the right of its actual position when the hit occurred. That is because the sprites have continued to move while two BASIC statements (lines 190 and 200) are interpreted and executed. The Assembly Language version has already stopped the motion by disabling the VDP interrupt program via LIMI 0; it doesn't start the motion again until after the hit sequence is complete. Thus, only the Assembly Language program actually shows the blow-up in the right place on the screen.

Understanding An Assembler Listing

The Assembly Language listing (Figure 4) was output by the 99/4A Assembler. You'll notice that the Assembler has added a page number and short title at the top of each page and added a cross-reference list and number-of-errors-found-during-assembly message to the end. The cross-reference list shows the location of the symbols used in the program relative to the beginning of the program. The line numbers in the first column were supplied by the Editor when the program was input and passed along by the Assembler. The second column of the listing shows the relative memory location where each statement or data area will reside during program execution. The third column was also supplied by the Assembler and shows the machine language generated by the Assembly Language statement to the right. The machine language (or *object code*) is expressed in hexadecimal notation with one word per line. The Assembly Language source program (or *source code*) itself starts in the fourth column, which contains the labels. The fifth column contains the source program opcodes, and the sixth column contains the operands. The seventh column contains comments, and other comments are sprinkled throughout the program with asterisks in column 1. *Only the fourth through seventh columns comprise the Assembly Language source program; this is the only part entered by the programmer.* The Assembler generates the rest.

The Utility Routines VMBW, VSBW, VWTR, and VMBR are used in the example program. The VDP Multiple Byte Write (VMBW) moves the number of bytes in register 2 (R2) from the CPU RAM address in R1 to the VDP RAM address in R0. VSBW, the VDP Single Byte Write routine, was explained earlier. VDP Write To Register (VWTR) puts the value that is in the rightmost byte of R1 into the VDP register whose number is in the leftmost byte of R1. Among other things, these VDP registers are used to select VDP modes and features. VMBR is the VDP Multiple Byte Read routine, which reads the number of bytes specified in R2 into the CPU RAM location in R1 from the VDP RAM location in R0.

The logic for detecting hits in the Assembly Language program is based on the fact that the point of the arrow is three pixels to the right and seven pixels below the corner of the sprite which is obtained from the Sprite Attribute Block.

Conclusion

Although they are more complex to write, Assembly Language programs are far superior to BASIC programs when it comes to execution speed and for controlling the facilities of the 99/4A computer. In some cases, as in the game of *Life*, the faster speed of Assembly Language turns a boring game into one which is fun to watch. In other cases, as in the program SHOOT, Assembly Language is capable of providing more accurate results. Thus, having the capability to write programs or subroutines in Assembly Language lets you achieve results which are impossible with BASIC and Extended BASIC alone.

Listing 1 *Life*

```
100 CALL CLEAR
110 DIM OFFSETS(8),AFTER(4)
120 FOR I=1 TO 8
130 READ OFFSETS(I)
140 NEXT I
150 DATA -33,-32,-31,-1
160 DATA 1,31,32,33
170 DIM SCRN(768)
180 REM  INITIALIZE
190 FOR I=1 TO 768
200 SCRN(I)=0
210 NEXT I
220 AFTER(0)=0
230 AFTER(1)=1
240 AFTER(2)=1
250 AFTER(3)=0
260 REM  INITIALIZE POPULATION
270 READ NUMSUB
280 FOR I=1 TO NUMSUB
290 READ ROW,COL
300 ISUB=(ROW-1)*32+COL
310 SCRN(ISUB)=1
320 CALL HCHAR(ROW,COL,42)
330 NEXT I
340 DATA 7
350 DATA 11,16,12,15,12,17,13,14,13,18
    ,14,14,14,18
360 REM  CALCULATE NEXT GENERATION
370 ISUB=34
380 FOR ROW=2 TO 23
390 FOR COL=2 TO 31
400 CNT=0
410 FOR K=1 TO 8
420 M=SCRN(ISUB+OFFSETS(K))
430 IF M=0 THEN 460
440 IF M=2 THEN 460
450 CNT=CNT+1
460 NEXT K
470 IF SCRN(ISUB)=1 THEN 500
480 IF CNT=3 THEN 520
490 GOTO 530
500 IF CNT=2 THEN 530
510 IF CNT=3 THEN 530
520 SCRN(ISUB)=SCRN(ISUB)+2
530 ISUB=ISUB+1
540 NEXT COL
550 ISUB=ISUB+2
560 NEXT ROW
570 REM  SHOW NEW GENERATION
580 CALL CLEAR
590 ISUB=34
600 FOR ROW=2 TO 23
610 FOR COL=2 TO 31
620 SCRN(ISUB)=AFTER(SCRN(ISUB))
630 IF SCRN(ISUB)=0 THEN 650
640 CALL HCHAR(ROW,COL,42)
650 ISUB=ISUB+1
660 NEXT COL
670 ISUB=ISUB+2
680 NEXT ROW
690 GOTO 370
700 END
```

Listing 2 *Shoot an Arrow*

```
100 CALL CLEAR
110 REM  DEFINE SPRITES
120 CALL CHAR(142,"FF81BDA5A5BD81FF")
130 CALL CHAR(143,"1818181818181818")
140 CALL CHAR(141,"8142241818244281")
150 CALL SPRITE(#1,142,7,124,1,0,100)
160 CALL SPRITE(#2,143,2,1,124,127,0)
170 REM  TEST FOR HIT
180 CALL COINC(#1,#2,10,HIT)
190 IF HIT=0 THEN 180
200 CALL MOTION(#1,0,0)
210 CALL MOTION(#2,0,0)
220 CALL PATTERN(#1,141)
230 FOR DELAY=1 TO 50
240 NEXT DELAY
250 GOTO 150
260 END
```

Listing 3 *Life*

```
            IDT 'LIFEA'
            DEF LIFEA
            REF VMBW
WS          BSS 32
SCRN        BSS 768
GENSCR      BSS 768
OFSET       DATA -33,-32,-31,-1
            DATA 1,31,32,33
FSTGEN      DATA 7,355,366,368,397,401,429,433
H00         BYTE >00
H01         BYTE >01
H02         BYTE >02
BLNK        BYTE >20
STAR        BYTE >2A
AFTER       BYTE 0,1,1,0
            EVEN
H2000       DATA >2000
LIFEA       LWPI WS                  START OF PROGRAM
*CLEAR      SCREEN ARRAY.
            LI R1,766                LOOP COUNTER AND INDEX
CLEAR       CLR @SCRN(R1)            CLEAR WORD
            DECT R1                  POINT TO WORD
            JLT  INIT                DONE
            JMP CLEAR
```

Listing 3 *Life* **continued**

```
*LOAD  INITIAL GENERATION AND DISPLAY.
INIT    MOV   @FSTGEN,R3     R3=#OF CELLS
        A     R3,R3          DOUBLE IT FOR WORDS
INITLP  MOV @FSTGEN(R3),R4   R4 CONTAINS OFFSET
        MOVB @H01,@SCRN(R4)     SCREEN POSITION =1
        DECT R3
        JNE  INITLP          MORE TO DO
        BL   @SHOWIT         SHOW INITIAL GEN
        LIMI 2               ENABLE VDP INTERRUPT FOR QUIT
*CALCULATE NEXT GENERATION.
CLCGEN LI    R1,33           INDEX(ISUB)
        LI   R5,22           OUTER LOOP CTR(ROW)
CLCLP   LI   R4,30
*COUNT NEIGHBORS.
CLCNBR LI    R5,0            NEIGHBORS COUNTER(CNT)
        LI   R6,0            LOOP CONTROL,INDEX TO OFSET
NBRS    MOV  R1,R7           COPY TO WORK ON
        A    @OFSET(R6),R7      R7->DISP OF NEIGHBOR
        CB   @SCRN(R7),@H00     NBR=0?
        JEQ  NXTNBR             YES
        CB   @SCRN(R7),@H02     NBR=2?
        JEQ  NXTNBR             YES
        INC  R5                 NEIGHBOR ON
NXTNBR  INCT R6
        CI   R6,16              DONE?
        JLT  NBRS               LOOK AT NEXT NEIGHBOR
        CB   @SCRN(R1),@H01     IS CELL ON NOW?
        JEQ  CELLON             YES
        CI   R5,3               3 NEIGHBORS?
        JEQ  CHANGE             YES-BIRTH
        JMP  NOCHG              NO
CELLON  CI   R5,2               2 NEIGHBORS?
        JEQ  NOCHG              YES-SURVIVE
        CI   R5,3               3 NEIGHBORS?
        JEQ  NOCHG              YES-SURVIVE
CHANGE  AB   @H02,@SCRN(R1)     BIRTH OR DEATH
NOCHG   INC  R1                 NEXT CELL
        DEC  R4                 NEXT COL
        JNE  CLCNBR
        INCT R1                 SKIP TWO EDGE CELLS
        DEC  R3                 NEXT ROW
        JNE  CLCLP
*RESET SCRN ELEMENTS TO 0 FOR DEAD, 1 FOR ALIVE.
        LI R5,33                INDEX TO SCRN(ISUB)
        LI R3,22                ROW CTR
LOOP    LI R4,30
LOOP1   MOVB @SCRN(R5),R6        R6=CELL VALUE IN MSB
        SRL  R6,8               SHIFT TO LSB
        MOVB @AFTER(R6),@SCRN(R5)   CHANGE CELL TO 0 OR 1
        INC  R5                 NEXT CELL
        DEC  R4                 NEXT COL
        JNE  LOOP1
        INCT R5
        DEC  R3
        JNE  LOOP
        BL   @SHOWIT            SHOW NEW GENERATION
        JMP  CLCGEN             CALC NEXT GEN
*SUBROUTINE TO DISPLAY GENERATION ON SCREEN.
SHOWIT LI    R5,767             R5 INDEXES BOTH SCRN
*                                 &GENSCR.
BLDSCR CB    @H00,@SCRN(R5)     IS BYTE 0 (DEAD)?
        JEQ  BLK                YES
        MOVB @STAR,@GENSCR(R5)  NO-PUT * IN GENSCR
        JMP  NXTPOS
BLK     MOVB @BLNK,@GENSCR(R5)  PUT BLANK IN GENSCR
NXTPOS  DEC  R5                 POINT TO NEXT CELL
        JLT  OUTSCR             DISPLAY IF DONE
        JMP  BLDSCR             LOOP IF NOT DONE
OUTSCR  CLR  R0                 VDP RAM ADDRESS (HOME)
        LI   R1,GENSCR          GENSCR CONTAINS DISP DATA
        LI   R2,768             768 BYTES TO WRITE
        LIMI 0
        BLWP @VMBW              WRITE SCREEN
        LIMI 2
        B    *R11                RETURN
        END  LIFEA
```

Listing 4 *Shoot an Arrow*

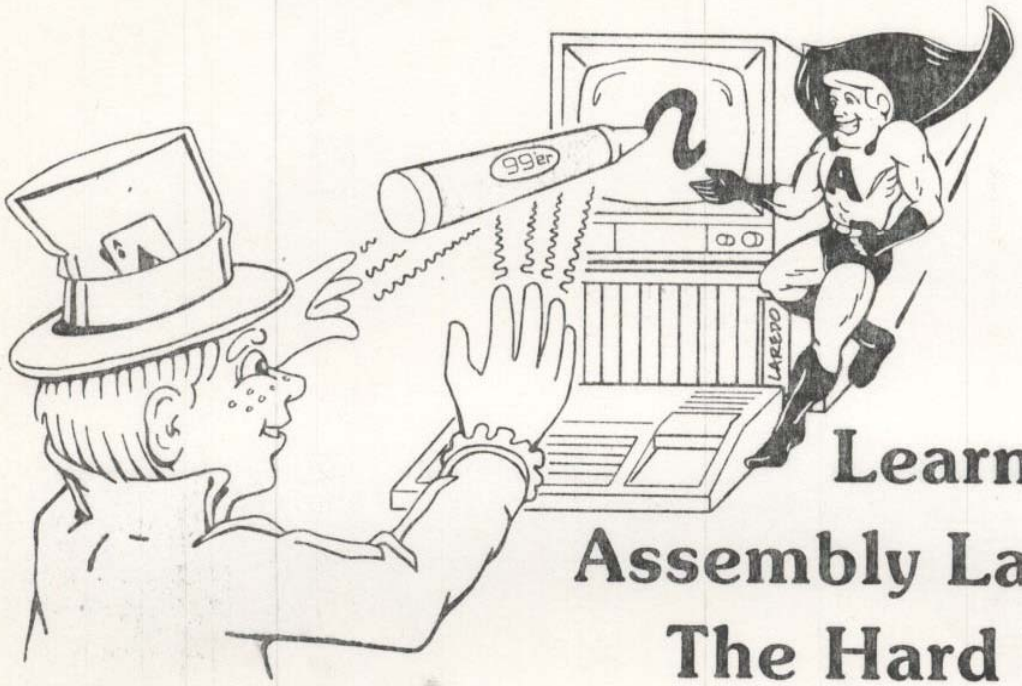
```
99/4 ASSEMBLER
VERSION 1.2                                                    PAGE 0001
  0001                          IDT   'SHOOTA'
  0002                          DEF   SHOOTA
  0003                          REF   VMBW,VSBW,VWTR,VMBR
  0004 0000             WS      BSS   32
  0005 0020    7C       SAL     BYTE  >7C,>01,>80,>06    SPRITE 1 LOCN AND COLOR
       0021    01
       0022    80
       0023    06
  0006 0024    01               BYTE  >01,>7C,>81,>01    SPRITE 2 LOCN AND COLOR
       0025    7C
       0026    81
       0027    01
  0007 0028    D0               BYTE  >D0                TERMINATOR
  0008 0029    FF       SHAPE   BYTE  >FF,>81,>BD,>A5,>A5,>BD,>81,>FF   TARGET
       002A    81
       002B    BD
       002C    A5
       002D    A5
       002E    BD
       002F    81
       0030    FF
  0009 0031    18               BYTE  >18,>18,>18,>18,>18,>18,>3C,>18   ARROW
       0032    18
       0033    18
       0034    18
       0035    18
       0036    18
       0037    3C
       0038    18
  0010 0039    81       HITSHP  BYTE  >81,>42,>24,>18,>18,>24,>42,>81   HIT SHAPE
       003A    42
       003B    24
       003C    18
       003D    18
       003E    24
       003F    42
       0040    81
  0011 0041    00       SPEED   BYTE  >00,>64,>00,>00    SPRITE 1 VELOSITY
       0042    64
       0043    00
       0044    00
  0012 0045    7F               BYTE  >7F,>00,>00,>00    SPRITE 2 VELOSITY
       0046    00
       0047    00
       0048    00
  0013 0049    00       H00     BYTE  >00
  0014 004A    02       H02     BYTE  >02
  0015 004B             Y1      BSS   1
  0016 004C             X1      BSS   1
  0017 004D             DUMMY   BSS   2
  0018 004F             Y2      BSS   1
  0019 0050             X2      BSS   1
  0020 0051    03       H03     BYTE  >03
  0021 0052    07       H07     BYTE  >07
  0022                          EVEN
  0023 0054 0020        H0020   DATA  >0020
  0024 0056 02E0        SHOOTA  LWPI  WS
       0058 0000'
  0025                          *FILL SCREEN WITH BLANKS.
```

Listing 4 *Shoot an Arrow* **continued**

```
 99/4 ASSEMBLER
VERSION 1.2                                                    PAGE 0002
  0026 005A 04C0          CLR   @                VDP RAM SCREEN HOME
  0027 005C 0201          LI    1,>2000          BLANK IN MSB OF R1
       005E 2000
  0028 0060 0420  BLNKIT  BLWP  @VSBW            WRITE BLANK
       0062 0000
  0029 0064 0580          INC   @
  0030 0066 0280          CI    @,768            DONE?
       0068 0300
  0031 006A 11FA          JLT   BLNKIT           NOT YET
  0032                  *SET UP VDP REGISTER 1
  0033 006C 0200          LI    @,>01E0          NORMAL SIZED SPRITES
       006E 01E0
  0034 0070 0420          BLWP  @VWTR
       0072 0000
  0035                  *SET UP SPRITE ATTIBUTE BLOCK.
  0036 0074 0201  DEFSPR  LI    1,SAL            R1-MY ATTRIBUTE LIST
       0076 0020'
  0037 0078 0200          LI    @,>0300          R@->ADDRESS OF VDP SAB
       007A 0300
  0038 007C 0202          LI    2,9              9 BYTES TO WRITE
       007E 0009
  0039 0080 0420          BLWP  @VMBW            WRITE TO VDP RAM
       0082 0000
  0040                  *LOAD SPRITE DEFINITIONS
  0041 0084 0201          LI    1,SHAPE          R1->MY SPRITE SHAPES
       0086 0029'
  0042 0088 0200          LI    @,>0400          ADDRESS OF FIRST SPRITE
       008A 0400
  0043 008C 0202          LI    2,16             16 BYTES TO MOVE
       008E 0010
  0044 0090 0420          BLWP  @VMBW            WRITE TO VDP RAM
       0092 0082'
  0045                  *SET UP SPRITE MOTION TABLE.
  0046 0094 0200          LI    @,>0780          R@->MOTION TABLE IN VDP RAM
       0096 0780
  0047 0098 0201          LI    1,SPEED          R1->MY SPEED DATA
       009A 0041'
  0048 009C 0202          LI    2,8              8 BYTES TO MOVE
       009E 0008
  0049 00A0 0420          BLWP  @VMBW            WRITE
       00A2 0092'
  0050                  *SET NUMBER OF MOVING SPRITES.
  0051 00A4 D820          MOVB  @H02,@>837A      2 MOVING SPRITES
       00A6 004A'
       00A8 837A
  0052                  *MAKE SPRITES MOVE BY INTERRUPT FROM 9901 I/O BOARD.
  0053 00AA 0300  MOVEIT  LIMI  2                ENABLE INTERRUPT
       00AC 0002
  0054                  *CHECK FOR COINCIDENCE.
  0055 00AE 0300          LIMI  @                DISABLE VDP INTERRUPT
       00B0 0000
  0056                  *GET SPRITE POSITIONS.
  0057 00B2 0200          LI    @,>0300          R@->Y OF SPRITE IN VDP RAM
       00B4 0300
  0058 00B6 0201          LI    1,Y1             BUFFER FOR READ
       00B8 004B'
  0059 00BA 0202          LI    2,6              6 BYTES TO READ
       00BC 0006
  0060 00BE 0420          BLWP  @VMBR            READ FROM VDP RAM
```

```
 99/4 ASSEMBLER                                                  PAGE 0003
VERSION 1.2
          00C0 0000
   0061                    *CHECK COLUMNS FOR X1<=X2+3<=X1+7
   0062 00C2 8820          AB    @H03,@X2          X2=X2+3
          00C4 0051'
          00C6 0050'
   0063 00C8 7820          SB    @X1,@X2           X2+X2-X1
          00CA 004C'
          00CC 0050'
   0064 00CE 11ED          JLT   MOVEIT            NO HIT IF RESULT >0
   0065 00D0 9820          CB    @X2,@H07          COMPARE TO 7
          00D2 0050'
          00D4 0052'
   0066 00D6 15E9          JGT   MOVEIT            NO HIT IF RESULT >7
   0067                    *CHECKS ROWS FOR Y1<=Y2+7<=Y1+7.
   0068 00D8 8820          AB    @H07,@Y2          Y2=Y2+7
          00DA 0052'
          00DC 004F'
   0069 00DE 7820          SB    @Y1,@Y2           Y2=Y2-1
          00E0 004B'
          00E2 004F'
   0070 00E4 11E2          JLT   MOVEIT            NO HIT IF RESULT <0
   0071 00E6 9820          CB    @Y2,@H07
          00E8 004F'
          00EA 0052'
   0072 00EC 15DE          JGT   MOVEIT            NO HIT IF RESULT >7
   0073                    *HIT
   0074                    *CHANGE SPRITE DEFINITIONS.
   0075 00EE 0201          LI    1,HITSHP          R1->HIT SHAPE
          00F0 0039'
   0076 00F2 0200          LI    0,>400            R0->VDP RAM
          00F4 0400
   0077 00F6 0202          LI    2,8               8 BYTES TO LOAD
          00F8 0008
   0078 00FA 0420          BLWP  @VMBW             WRITE TO VDP RAM
          00FC 00A2'
   0079                    *WAIT TO LET BLOW UP BE SEEN.
   0080 00FE 0203          LI    3,10              OUTER LOOP CTR
          0100 000A
   0081 0102 0202  LOOP2A  LI    2,12000           LOOP CUONTER
          0104 2EE0
   0082 0106 0602  LOOP2   DEC   2                 DECREMENT
   0083 0108 16FE          JNE   LOOP2             WAIT MORE
   0084 010A 0603          DEC   3                 DECREMENT OUTER CTR
   0085 010C 16FA          JNE   LOOP2A            WAIT MORE
   0086 010E 10B2          JMP   DEFSPR            START OVER
   0087                    END   SHOOTA
L
 99/4 ASSEMBLER                                                  PAGE 0004
VERSION 1.2
   ' BLNKIT    0060     ' DEFSPR    0074     ' DUMMY     004D     ' H00       0049
   ' H0020     0054     ' H02       004A     ' H03       0051     ' H07       0052
   ' HITSHP    0039     ' LOOP2     0106     ' LOOP2A    0102     ' MOVEIT    00AA
     R0        0000       R1        0001       R10       000A       R11       000B
     R12       000C       R13       000D       R14       000E       R15       000F
     R2        0002       R3        0003       R4        0004       R5        0005
     R6        0006       R7        0007       R8        0008       R9        0009
   ' SAL       0020     ' SHAPE     0029     D SHOOTA    0056     ' SPEED     0041
   E VMBR      00C0     E VMBW      00FC     E VSBW      0062     E VWTR      0072
   ' WS        0000     ' X1        004C     ' X2        0050     ' Y1        004B
   ' Y2        004F
    0000 ERRORS
```

MAGIC CRAYON



Learning Assembly Language The Hard Way

Like many other 99'ers, I was anxious to receive the long-awaited Editor/Assembler package. I remember the excitement of unwrapping the 470 page manual when it arrived—and the sinking feeling when I read, "This manual assumes that you already know a programming language, preferably an assembly language."

My anxiety grew as I thumbed through it—there were no pictures, cartoons, or fill-in-the-blank examples. It did say, "There are many fine books available which teach the basics of assembly language." So I called the local computer stores. The only books they were aware of, however, also assumed familiarity with basics.

I guess I had some fuzzy ideas about assembly language in the back of my mind: It was qualitatively different from higher level languages, requiring an in-depth knowledge of digital electronics and a capacity for the most detailed sort of logical-mathematical thought. In short—nothing seemed more difficult. . .

And my experience thus far seemed to confirm my worst fear. Learning assembly language presumed a prior knowledge of assembly language; it was not merely difficult—it was *impossible.* After running *Tombstone City* a few times and typing in Pat Swift's *Life* program (See "Fundamentals of Assembly Language Programming, Part 1"), I put the Editor/Assembler on a shelf thinking maybe I'd learn about it gradually over the next year or two.

It would still be there gathering dust were it not for a back injury that kept me flat on the floor, unable to do anything *except* read the manual. I was surprised to discover that writing an assembly language program is similar to, and in some respects simpler than, writing a program in BASIC. A new programming context or conceptual model is required. But to get started, I found that this picture could be primitive, containing many over-simplifications and approximations.

The picture I developed enabled me to successfully formulate and execute a simple programming objective. The program and associated underlying concepts are presented here to facilitate the learning process for others who, like me, find it hard to overcome preconceived notions about how difficult assembly language is. The program should not be taken as a model of exemplary programming technique; at this point my conception of "good programming" is programming that works . . . period. You will undoubtedly be able to find ways to improve this one—to make it work faster and utilize memory more efficiently—and in so doing, further develop the concepts presented.

In TMS9900 Assembly Language, four video display modes are available: Graphics (or Pattern) Mode, Text Mode, Bit-Map Mode (99/4A only), and Multicolor Mode. In Multicolor Mode, the screen is divided into a grid 64 × 48, with each box measuring 4 pixels on a side. Each box can have a color assigned to it.

The program allows use of a joystick to move a flashing cursor on the screen. Whenever the fire button is depressed, the cursor leaves a trail of small, colored boxes. The following single key commands are available:

C—*Change Color.* Displays a color palette and pointer. Move the pointer to the desired color with the joystick. Press the fire button to make that the color of the boxes, or press the C key to make it the color of the screen background.

S—*Save Screen.* Saves the current contents of the screen as DSK1.SCREEN.

R—*Recall Screen*. Loads the contents of DSK1.SCREEN for subsequent modification.

E—*Erase Screen*. Erases the screen contents.

T—*Terminate*. Returns to the Master Title Screen.

In order to understand how the program works, it will be helpful to differentiate two systems. You probably know that the Central Processing Unit (CPU) in the Home Computer is the TMS9900. It has three built-in 16-bit "hardware" registers (the Program Counter, Workspace Pointer, and Status Register) and makes use of sixteen workspace registers located in read-write memory. Because these 16-bit workspace registers are not located on the chip, they are called "software" registers. The CPU can directly address the read-write memory (RAM) in the Memory Expansion Unit and CPU scratch pad, as well as ROM in the console, Command Cartridges, and various peripherals. However, it cannot directly address the 16K of RAM built into the console.

The 16K RAM block is addressed by another microprocessor—The TMS9918 (or 9918A if you have a 99/4A). This Video Display Procesor (VDP) has eight 8-bit hardware registers and four 8-bit software registers. The software registers are located in read-write memory locations which can also be addressed by the CPU. The fact that these four bytes can be addressed by both the CPU and VDP makes it possible for the CPU and VDP systems to transfer data back and forth. The CPU addresses of the registers—8800, 8802, 8C00, 8C02—are assigned respectively to the symbols VDPRD (VDP Read Data Address), VDPSTA (VDP Read Status Register), VDPWD (VDP Write Data Address), VDPWA (VDP Write Address).

We don't have to be concerned with the details of moving data to and from VDP RAM and to VDP registers, however, thanks to some of the built-in programs called *utilities*. The five utilities of use are identified by the symbols VSBW, VMBW, VSBR, VMBR, and VWTR. The respective functions of these programs are VDP RAM: Single Byte Write, Multiple Byte Write, Single Byte Read, Multiple Byte Read, and Write to Register. User workspace registers are used to pass parameters—e.g., the number of bytes to read or write—to the utility.

The standard utilization of VDP RAM in the Editor/Assembler is shown on Table 1. The blocks involved in the multicolor mode are the Screen Image and Pattern Descriptor Tables. Before entering multicolor mode, the Screen Image Table is initialized. The 768 bytes of the table are divided into six 128-byte sets. Each set is further subdivided into four 32-byte groups. To initialize the table, the numbers 1-31 are written in order into each of the four 32-byte groups in the first set: 0, 1, 2,. . . 31 four times. Then the numbers 32-61 are written four times into the next 128-byte set. This process is continued until the numbers 160-191 are written four times in the sixth 128-byte set. In my program, I didn't want this process to be visible on the screen, so I first put the display in Text Mode and made the foreground and background colors gray.

Once the Screen Image Table is initialized, color boxes are placed on the screen by means of the Pattern Descriptor Table. Each 4×4 pixel box on the screen corresponds to half a byte in the Pattern Descriptor Table. To place a colored box on the screen, the appropriate color code is written in the nybble (4 bits) in the Pattern Descriptor Table which corresponds to the desired screen position.

Table 1 VDP RAM MEMORY —Editor/Assembler—

Address of First Byte Decimal	Hex	Length of Block, Bytes	Contents
0	>0000	768	Screen Image Table
768	>0300	128	Sprite Attribute List
896	>0380	128	Color Table
1024	>0400	896	Sprite Descriptor Table
1920	>0780	128	Sprite Motion Table
2048	>0800	2048	Pattern Descriptor Table and Peripheral Access Blocks
4096	>1000	10199	More Peripheral Access Blocks and Buffers
14295	>37D7	2089	Reserved for Diskette Device Service Routines
16383	>3FFF	___	Last Address
Total 16384 Bytes			

The first eight bytes of the Pattern Descriptor Table correspond to the boxes in a column beginning in the upper left corner of the screen. The first four bits in byte #1 contain the color of the box in the extreme upper left corner, and the last four bits the color of the box immediately to the right of the first box. Byte #2 contains the colors of the two boxes immediately under the first two, and so on for the first eight bytes.

The ninth byte in the table contains the colors for the pair of boxes in a new column beginning again at the top of the screen. Subsequent bytes follow this pattern corresponding to 32 columns of box pairs with eight pairs in each column. This group of 256 bytes thus takes care of the top sixth of the screen.

The 257th byte corresponds to the beginning of a new column of box pairs starting again on the left side of the screen. The six 256-byte groups thus correspond to the 3,072 possible boxes in multicolor mode. [Since the color of each box is indicated in a name table in memory, and the names are mapped onto the screen according to their position in the table, this multicolor mode is a *true* memory-mapped configuration. It does, however, trade off lower resolution for color memory-mapping capability, but the high-resolution sprites are still available. For an explanation of sprites and an introduction to the high-resolution bit-map mode, see "3-D Animation".—Ed.]

In the program, a double-size sprite provides a reference point for determining where boxes will appear. The dot row and dot column of the sprite can be determined at any time by referring to the Sprite Attribute List in VDP RAM. Then, since boxes are supposed to appear in the center of the sprite, the screen location can be calculated by adding 8 to the dot row and dot column, which represent the sprite's upper left corner. But in order to find the corresponding location in the Pattern Descriptor Table, a few more calculations must be performed.

If we let R and C be the dot row and dot column desired for the box location, the number of complete 256-byte groups above that location is the integer quotient of R/32. Multiplying that number by 256 thus gives the first component of the offset in the Pattern Descriptor Table.

Similarly, the integer quotient of C/8 gives the number of complete 8-byte columns to the left of the location. So

multiplied by 8 and added to the offset. The mnemonic BSS stands for Block Starting with Symbol.

that number is multiplied by 8 and added to the offset. Dividing the remainder of R/32 by 4 gives the number of bytes above the location in the 8-byte column the location is in. Adding that to the offset gives the offset for the byte in the Pattern Descriptor Table.

But we still have to know if the desired location is the most or least significant nybble of the byte, and to determine that we can divide the remainder of C/8 by 4. If the integer quotient is 0, it's the left nybble; if 1, it's the right nybble. The appropriate color code then need only be placed in the correct nybble (leaving the other one unchanged), and the box appears just where it should.

Let's consider an example: Suppose the upper left corner of the sprite were at dot row 83 and dot column 147. The center of the sprite would then be at 91 and 155. The number of complete groups (32 columns with 8 bytes in each) above that location is 2, i.e., INT(91/32). So the initial component of the offset is 2 * 256 or 512 bytes. The number of 8-byte columns to the left of the location is INT(155/8) or 19. That makes the offset 531. Above the location, in its 8-byte column, there are 6 bytes—i.e., INT((remainder 91/32)/4)—giving an offset of 537. The remainder of 155/8 is 3, and INT(3/4) is 0, so the nybble of interest is the most significant (left) one of the 539th byte of the Pattern Descriptor Table.

Now let's take a brief look at the source listing. The first section consists of a number of assembler directives. The DEF directive makes the symbol MARKER available to other programs, and the REF directives make several utilities available for use of MARKER. Then there is a variety of other assembler directives. The simplest type is EQUate, which assigns a constant to a symbol at assembly time. USRWS, for >20BA (8378), and that value replaces the symbol wherever it appears in an operand; the label may subsequently be substituted for the number.

The mnemonic BSS stands for Block Starting with Symbol. This directive causes the assembler to advance its location counter without writing anything into the object program. It leaves an empty area (of the number of bytes specified in the operand) which can then be used as a storage space for data later on. The label is set equal to the memory location of the first byte in the block at the time the object program is loaded. (Since this program is relocatable, the place where the loader program decides to start loading it may change, depending on what other programs have already been loaded.)

The DATA, BYTE, and TEXT directives are similar to BSS except that the contents of the buffer are explicitly defined in the operand field. The label is assigned the address of the first byte at the time the object program is loaded. All of these buffer areas are contiguous. For example, look at the instructions immediately after the label MARKER. The pattern codes for two double-size sprites, the cursor and arrow, are loaded into the Sprite Descriptor Table in VDP RAM. Since the pattern data for ARROW is contiguous with that of CURSOR in both CPU and VDP RAM, all 64 bytes can be loaded in one shot.

You should have little trouble figuring out the rest of the program by reading the comments provided and referring to the manual. But don't stop after you understand how it works—try to make some changes. To start with, try changing the shape and colors of the sprite cursor, the arrangement of the color palette on the screen, etc. Then try to make the program more efficient in speed and utilization of memory.

Be prepared to run into problems; it's through encountering and solving them that you'll learn most rapidly. When I decided to stop reading and start trying to write a program, I had visions of seeing a curl of white smoke rise from the computer's cooling vents, but that didn't happen to me and probably won't happen to you either. So don't be afraid to experiment.

Listing 1 *Magic Crayon*

```
        DEF   MARKER
        REF   VSBW,VMBW,VMBR,VSBR
        REF   VWTR,KSCAN,DSRLNK
*
* DEFINITION OF LABELS
*
SCREEN  BSS   >300
PALET   BSS   >600
PATRN   BSS   >680
ROW     BSS   1
COL     BSS   1
CURSOR  DATA  >8040,>2010,>0804,>0000
        DATA  >0000,>0408,>1020,>4080
        DATA  >0102,>0408,>1020,>0000
        DATA  >0000,>2010,>0804,>0201
ARROW   DATA  >0102,>0408,>0000,>0000
        DATA  >0000,>0000,>0000,>0000
        DATA  >0080,>4020,>0000,>0000
        DATA  >0000,>0000,>0000,>0000
ATTRIB  DATA  >5878,>800F,>D000
ARRATT  DATA  >6578,>8401
PDATA   DATA  >0600,>1000,>0000,>0600
        DATA  >000B
        TEXT  'DSK1.SCREEN'
ZERO    DATA  >0000
D32     DATA  >0020
D8      DATA  >0008
GRAY    DATA  >EEEE
MAX     DATA  >05FF
COLMAX  DATA  >0100
LOAD    BYTE  >05
BLACK   BYTE  >11
ONE     BYTE  >01
TWO     BYTE  >02
FCOLOR  BYTE  >10
BCOLOR  BYTE  >0E
H18     BYTE  >12
H14     BYTE  >0E
H11     BYTE  >0B
H07     BYTE  >07
H06     BYTE  >06
H05     BYTE  >05
H02     BYTE  >02
NOKEY   BYTE  >FF
PAB     EQU   >0FS0
USRWS   EQU   >20BA
PNTR    EQU   >8356
UNIT    EQU   >8374
FIRE    EQU   >8375
JOYSTY  EQU   >8376
JOYSTX  EQU   >8377
SPRITE  EQU   >837A
STATUS  EQU   >837C
GPLWS   EQU   >83E0
*
* DEFINE SPRITE PATTERNS FOR CHRS 128 AND 132
*
MARKER  LWPI  USRWS             LOAD WORKSPACE POINTER / START
        LI    R0,>400           VDP ADDRESS CH 128 SPRITE DESCRIPTOR TABLE
        LI    R1,CURSOR         CPU ADDRESS OF CHAR PATTERN
        LI    R2,64             64 BYTES TO MOVE (2 PATERNS)
        BLWP  @VMBW             LOAD DATA TO VDP RAM
*
* SET FOREGROUND AND BACKGROUND TO GRAY
*
        LI    R0,>01F0          PLACE IN TEXT MODE
        BLWP  @VWTR             WRITE TO VDP R1
        LI    R0,>07EE          SET FORE AND BACKGROUND TO GRAY
        BLWP  @VWTR             WRITE TO VDP R7
*
* INITIALIZE SCREEN IMAGE TABLE FOR MULTICOLOR MODE
*
        LI    R0,SCREEN         INITIALIZE POINTER
        LI    R1,6              INITIALIZE GROUP COUNTER
        CLR   R2                INITIALIZE VALUE
LOOP0   LI    R3,4              INITIALIZE REPETITIONS COUNTER
LOOP1   LI    R4,>20            INITIALIZE VALUE COUNTER
        MOVB  R2,R5             START REPETITION
LOOP2   MOVB  R5,*R0+           STORE VALUE IN ARRAY SCREEN
        AI    R5,>0100          CHANGE TO NEXT VALUE
        DEC   R4                COUNT DOWN FOR NEXT VALUE
        JNE   LOOP2             DO NEXT VALUE
        DEC   R3                DEC REPETITION COUNTER
```

Listing 1 *Magic Crayon* **continued**

```
          JNE   LOOP1                    DO NEXT REPETITION
          AI    R2,>2000                 NEXT STARTING VALUE
          DEC   R1                       DEC GROUP COUNTER
          JNE   LOOP0                    DO NEXT GROUP
          LI    R0,>00                   VDP ADDRESS FOR SCREEN IMAGE
          LI    R1,SCREEN                CPU ADDRESS OF DATA BUFFER
          LI    R2,>300                  768 BYTES TO WRITE
          BLWP  @VMBW                    INITIALIZE VDP SCREEN IMAGE
*
* INITIALIZE COLOR PALETTE SCREEN
*
          LI    R0,>100                  INITIALIZE WORD COUNTER
          LI    R1,PALET                 INITIALIZE POINTER FOR PALET ARRAY
LOOP3     MOV   @GRAY,*R1+               STORE GRAY COLOR >EEEE
          DEC   R0                       DEC WORD COUNTER
          JNE   LOOP3                    WRITE NEXT WORD
          CLR   R0                       INITIALIZE COLOR VALUE
          LI    R3,16                    INITIALIZE COLOR COUNTER
LOOP4     LI    R4,2                     INITIALIZE COLUMN COUNTER
LOOP5     MOVB  @GRAY,*R1+               STORE GRAY BYTE
          MOVB  @GRAY,*R1+               STORE ANOTHER GRAY BYTE
          MOVB  @BLACK,*R1+              STORE BLACK BYTE
          LI    R5,4                     LOAD COUNTER FOR COLOR BYTES
LOOP6     MOVB  R0,*R1+                  STORE A COLOR BYTE
          DEC   R5                       DEC COLOR BYTE COUNTER
          JNE   LOOP6                    STORE ANOTHER COLOR BYTE
          MOVB  @BLACK,*R1+              STORE A BLACK BYTE
          DEC   R4                       DEC COLUMN COUNTER
          JNE   LOOP5                    DO SECOND COLUMN
          SWPB  R0                       SHIFT TO LEAST SIG BYTE
          AI    R0,>11                   ADD 1 FOR NEXT COLOR NUMBER
          SWPB  R0                       SHIFT BACK TO MOST SIG BYTE
          DEC   R3                       COUNT DOWN COLOR COUNTER
          JNE   LOOP4                    DO NEXT TWO COLUMNS
          LI    R0,>300                  SET BYTE COUNTER FOR REMAINING SCREEN
LOOP7     MOVB  @GRAY,*R1+               STORE A GRAY BYTE
          DEC   R0                       COUNT DOWN
          JNE   LOOP7                    REPEAT UNTIL DONE
*
* INITIALIZE PATTERN TABLE - TRANSPARENT
*
CLEAR     LI    R0,>300                  INITIALIZE WORD COUNTER
          LI    R1,PATRN                 INITIALIZE POINTER FOR PATTERN ARRAY
LOOP8     MOV   @ZERO,*R1+               STORE COLOR = TRANSPARENT
          DEC   R0                       COUNT DOWN FOR NEXT WORD
          JNE   LOOP8                    WRITE NEXT WORD IN ARRAY
*
* LOAD PATTERN TABLE
*
          LI    R0,>800                  VDP PATTERN TABLE ADDRESS
          LI    R1,PATRN                 CPU BUFFER ADDRESS
          LI    R2,>600                  1536 BYTES TO WRITE
          BLWP  @VMBW                    WRITE TO VDP RAM
*
* SELECT DOUBLE SIZE AND MULTICOLOR MODE
*
          LI    R0,>01EA                 TO WRITE 11101010 TO VDP R1
          BLWP  @VWTR                    WRITE TO VDP R1
          SWPB  R0                       MOVE >EA TO MOST SIG BYTE
          MOVB  R0,@>83D4                STORE COPY (>EA) IN CPU RAM
*
* DEFINE ATTRIBUTES FOR SPRITE #0
*
          LI    R0,>300                  VDP SPRITE ATTRIBUTE LIST
          LI    R1,ATTRIB                LOCATION OF ATTRIBUTE LIST FOR SPRITE 0
          LI    R2,6                     6 BYTES TO MOVE
          BLWP  @VMBW                    WRITE DATA TO VDP RAM.
*
* DEFINE # OF ACTIVE SPRITES
*
          MOVB @ONE,@SPRITE              STORE NO. OF ACTIVE SPRITES IN CPU RAM
*
* INITIALIZE CURSOR COLOR AND COLOR CHANGE COUNTER
*
          LI    R3,>0F01                 SPRITE COLORS - WHITE/BLACK IN R3
          CLR   R4                       INITIALIZE COUNTER - COLOR CHANGE
*
* ---------- START MAIN LOOP ----------------
*
* CHECK JOYST FOR MOTION, FIRE BUTTON AND KEYS
*
CHECK     LIMI 2                         ENABLE INTERRUPTS
          LIMI 0                         DISABLE INTERRUPTS
```

Listing 1 *Magic Crayon* continued

```
         LI    R0,1                    INDICATE REPETIONS OF CHECKS
         BL    @CHECKS                 BRANCH TO SUBROUTINE CHECKS
         MOVB  @ONE,@UNIT              SELECT REMOTE UNIT TO SCAN
         BLWP  @KSCAN                  SCAN LEFT KEYBOARD
         CB    @FIRE,@H05              WAS 'E' PRESSED?
         JEQ   CLEAR                   IF YES GO TO CLEAR SCREEN
         CB    @FIRE,@H02              WAS 'S' PRESSED?
         JNE   NEXT1                   IF NOT, GO ON
         B     @SAVE                   IF SO, BRANCH TO SAVE ROUTINE
NEXT1    CB    @FIRE,@H06              WAS 'R' PRESSED?
         JNE   NEXT2                   IF NOT, GO ON
         B     @RECALL                 IF SO, BRANCH TO RECALL ROUTINE
NEXT2    CB    @FIRE,@H11              WAS 'T' PRESSED?
         JNE   NEXT3                   IF NOT, GO ON
         LIMI  2                       ENABLE INTERRUPTS
         LWPI  GPLWS                   LOAD GPL WORK SPACE
         BLWP  @0000                   RETURN TO MASTER TITLE SCREEN
NEXT3    CB    @FIRE,@H14              WAS 'C' PRESSED?
         JNE   NEXT4                   IF NO, GO ON
         B     @SELECT                 IF YES, GO TO COLOR SELECT ROUTINE
NEXT4    CB    @FIRE,@H18              WAS FIRE BUTTON PRESSED?
         JNE   SKIP                    IF NO, SKIP DRAW ROUTINE
*
* ROUTINE TO PLACE BLOCK ON SCREEN
*
DRAW     LI    R0,>300                 VDP SPRITE ATTRIBUTE ADDRESS
         LI    R1,ROW                  CPU BUFFER TO RECEIVE DATA
         LI    R2,2                    FETCH 2 BYTES
         BLWP  @VMBR                   FETCH DOT ROW AND DOT COLUMN
         CLR   R7                      INITIALIZE R7 AND R8
         CLR   R8                      --FOR USE IN DIVIDE OPERATION
         CLR   R2                      INITIALIZE OFFSET FOR PATRN ARRAY
         MOVB  @ROW,R8                 PUT DOT ROW IN R8
         SWPB  R8                      MAKE IT LEAST SIG BYTE
         AI    R8,9                    ADD ROW OFFSET FOR COLOR BLOCK +1
         C     R8,@COLMAX              IS THE DOT ROW > 255?
         JLT   NOCORR                  IF NOT, DO NOT APPLY CORRECTION
         S     @COLMAX,R8              IF SO, SUBTRACT 255
NOCORR   DIV   @D32,R7                DIVIDE DOT ROW OF BLOCK BY 32
         SLA   R7,8                    CALCULATE BYTES IN PRECEEDING GROUPS
         A     R7,R2                   ADD # OF BYTES IN PREVIOUS 32X8 BYTE GROUPS
         SRL   R8,2                    DIVIDE REMAINDER BY 4
         A     R8,R2                   ADD # BYTES ABOVE IN CURRENT 8 BYTE SET
         CLR   R7                      INITIALIZE R7 AND R8
         CLR   R8                      --FOR USE IN DIVIDE OPERATION
         MOVB  @COL,R8                 PUT DOT COLUMN IN R8
         SWPB  R8                      MAKE IT LEAST SIG BYTE
         AI    R8,8                    ADD COLUMN OFFSET FOR COLOR BLOCK
         C     R8,@COLMAX              IS THE DOT COLUMN > 255?
         JLT   NOCORC                  IF NOT, DO NOT APPLY CORRECTION
         S     @COLMAX,R8              IF SO, SUBTRACT 256
NOCORC   DIV   @D8,R7                 DIVIDE BY 8
         SLA   R7,3                    CALCULATE BYTES IN PRECEEDING 8 BYTE SET'S
         A     R7,R2                   ADD # BYTES IN PREVIOUS 8 BYTE SETS, THIS GROUP
         MOV   R2,R2                   CHECK IF INSIDE PATTERN ARRAY  N
         JLT   SKIP                    IF NOT SKIP SCREEN PLACEMENT
         C     R2,@MAX                 CHECK IF INSIDE PATTERN ARRAY EEN
         JGT   SKIP                    IF NOT SKIP SCREEN PLACEMENT
         LI    R0,>14                  REPEAT SUBROUTINE CHECKS 20 TIMES
         BL    @CHECKS                 BRANCH TO SUBROUTINE CHECKS
         CLR   R1                      INITIALIZE R1 FOR BLOCK COLOR
         MOVB  @FCOLOR,R1              STORE COLOR IN R1
         SWPB  R1                      MAKE IT LEAST SIG BYTE
         CLR   R0                      INITIALIZE R0 FOR CURRENT ARRAY ELEMENT
         MOVB  @PATRN(2),R0            COPY ARRAY ELEMENT AT OFFSET INTO R0
         SRL   R8,2                    CALCULATE WHETHER BLOCK IS LEFT OR RIGHT
         JEQ   MARK1                   IF 0 LEAVE BLOCK AS LEFT NYBBLE
         SRL   R1,4                    IF 1 MAKE BLOCK RIGHT NYBBLE
         SWPB  R0                      MAKE CURRENT ELEMENT LEAST SIG BYTE
         SRL   R0,4                    GET RID OF LEAST SIG NYBBLE
         SLA   R0,4                    PUT REMAINING NYBBLE BACK
         JMP   MARK2                   SKIP TO LABEL
MARK1    SLA   R0,4                    GET RID OF MOST SIG NYBBLE
         SRL   R0,4                    PUT BACK REMAINING NYBBLE
         SWPB  R0                      MAKE IT LEAST SIG BYTE
MARK2    A     R1,R0                   ADD NEW COLOR TO ADJACENT VALUE
         SWPB  R0                      MAKE IT MOST SIG BYTE
         MOVB  R0,@PATRN(2)            MOVE IT TO ARRAY AT OFFSET
         LI    R0,>0800                VDP PATTERN TABLE ADDRESS
         LI    R1,PATRN                CPU BUFFER
         LI    R2,>600                 1536 BYTES TO MOVE
         BLWP  @VMBW                   WRITE TO REDRAW SCREEN
```

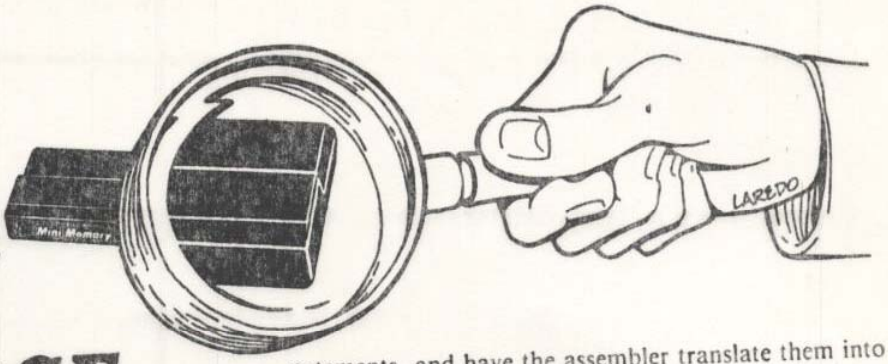
Listing 1 ***Magic Crayon*** **continued**

```
SKIP    CLR   R5                      CLEAR R5 AND R6 TO RECEIVE JOYST VALUES
        CLR   R6
        MOVB  @JOYSTY,R5              PUT Y RETURN IN R5
        NEG   R5                      MULTIPLY BY -1
        SLA   R5,2                    MULTIPLY BY 4
        MOVB  @JOYSTX,R6              PUT X RETURN IN R6
        SLA   R6,2                    MULTIPLY TIMES 4
        SWPB  R6                      MAKE XVEL LEAST SIG BYTE
        MOVB  R5,R6                   MOVE YVEL TO R6 AS MOST SIG BYTE
        LI    R1,USRWS+12             CPU ADDRESS OF VELOCITY BYTES (R6)
        LI    R0,>0780                VDP ADDRESS OF MOTION TABLE
        LI    R2,2                    2 BYTES TO MOVE
        BLWP  @VMBW                   WRITE DATA TO VDP RAM
        B     @CHECK                   START LOOP OVER AGAIN
*
* ----- END OF MAIN PROGRAM LOOP ----------------------
*
* COLOR SELECT ROUTINE
*
SELECT  LI    R0,>07EE                CHANGE BACKGROUND TO GRAY
        BLWP  @VWTR                   WRITE TO VDP R7
        LI    R0,>800                 VDP BUFFER FOR PATTERN TABLE
        LI    R1,PALET                CPU BUFFER FOR PALETTE
        LI    R2,>600                 1536 BYTES TO MOVE
        BLWP  @VMBW                   DISPLAY PALETTE
        LI    R0,>300                 VDP BUFFER FOR ATTRIBUTE LIST
        LI    R1,ARRATT               ARROW ATTRIBUTES
        LI    R2,4                    4 BYTES TO MOVE
        BLWP  @VMBW                   WRITE DATA
        BL    @DEBNC                  BRANCH TO 'DEBOUNCE' SUBROUTINE
LOOP9   LIMI  2                       ENABLE VDP INTERRUPT
        LIMI  0                       DISABLE INTERRUPT
        MOVB  @ONE,@UNIT              IDENTIFY REMOTE UNIT TO SCAN
        BLWP  @KSCAN                  SCAN LEFT KBD AND REMOTE UNIT #1
        CB    @FIRE,@H18              CHECK FIRE BUTTON
        JEQ   CMARK                   IF PRESSED, CHANGE MARK COLOR
        CB    @FIRE,@H14              CHECK 'C' KEY
        JEQ   CSGRN                   IF PRESSED, CHANGE SCREEN COLOR
        CLR   R6                      INITIALIZE R6
        MOVB  @JOYSTX,R6              PUT JOYST X IN R6
        SLA   R6,2                    MPY BY 4
        SWPB  R6                      MAKE LEAST SIG BYTE
        LI    R1,USRWS+12             LOAD CPU ADDRESS (R6)
        LI    R0,>0780                LOAD ADDRESS OF MOTION TABLE
        LI    R2,2                    MOVE 2 BYTES
        BLWP  @VMBW                   LOAD DATA TO VDP RAM
        JMP   LOOP9                   GOTO LOOP9
CSCRN   BL    @DOTCOL                 DETERMINE COLOR FROM DOT COLUMN OF ARROW
        SWPB  R1                      MAKE IT MOST SIG BYTE
        MOVB  R1,@BCOLOR              MOVE IT TO BCOLOR
        JMP   BACK                    JUMP TO BACK
CMARK   BL    @DOTCOL                 DETERMINE COLOR FROM DOT COLUMN OF ARROW
        SLA   R1,12                   PUT IN PROPER POSITION FOR @FCOLOR
        MOVB  R1,@FCOLOR              MOVE IT TO FCOLOR
BACK    BL    @DEBNC                  DEBOUNCE
        CLR   R0                      PREPARE TO RETURN SCREEN COLOR
        MOVB  @BCOLOR,R0              PUT BACKGROUND COLOR IN R0
        SWPB  R0                      MAKE IT LEAST SIG BYTE
        MOVB  @H07,R0                 INDICATE WRITE TO VDP R7
        BLWP  @VWTR                   WRITE IT TO R7
        LI    R0,>800                 VDP PATTERN TABLE ADDRESS
        LI    R1,PATRN                PATTERN BUFFER IN CPU RAM
        LI    R2,>600                 1536 BYTES TO WRITE
        BLWP  @VMBW                   LOAD PATTERN SCREEN
        LI    R0,>300                 VDP SPRITE ATTRIBUTE TABLE ADDRESS
        LI    R1,ATTRIB               ADDRESS OF CURSOR ATTRIBUTES
        LI    R2,4                    4 BYTES TO MOVE
        BLWP  @VMBW                   LOAD DATA TO GET CURSOR SPRITE
        B     @SKIP                   BRANCH TO LABLE SKIP
*
* DSR ROUTINE TO SAVE 'SCREEN' -- PATTERN TABLE
*
SAVE    LI    R0,>1000                PREPARE TO MOVE PATRN TO VDP BUFFER
        LI    R1,PATRN                CPU BUFFER ADDRESS
        LI    R2,>600                 1536 BYTES TO MOVE
        BLWP  @VMBW                   WRITE DATA
        LI    R0,PAB                  VDP PERIPHERAL ACCESS BLOCK ADDRESS
        LI    R1,PDATA                CPU BUFFER TO BE WRITTEN TO VDP
        LI    R2,21                   21 BYTES TO WRITE
        BLWP  @VMBW                   WRITE PAB
        LI    R6,PAB+9                SET POINTER TO NAME LENGTH
        MOV   R6,@PNTR                STORE IN >8356 >8357
        BLWP  @DSRLNK                 EXECUTE SAVE OR LOAD
```

Listing 1 *Magic Crayon* continued

```
        DATA 8
        B    @CHECK              IF SO, BRANCH BACK TO BEGINNING
*
* DSR ROUTINE TO RECALL 'SCREEN' -- PATTERN TABLE
*
RECALL LI    R0,PAB              VDP PERIPHERAL ACCESS BLOCK ADDRESS
       LI    R1,PDATA            CPU BUFFER TO WRITE
       LI    R2,21               21 BYTES TO WRITE
       BLWP  @VMBW               WRITE PAB
       LI    R0,PAB              SUBSTITUTE 'LOAD' I/O OP CODE
       MOVB  @LOAD,R1            MOVE OP CODE TO R1
       BLWP  @VSBW               WRITE BYTE TO PAB
       LI    R6,PAB+9            SET POINTER TO NAME LENGTH
       MOV   R6,@PNTR            STORE IN >8356 >8357
       BLWP  @DSRLNK             COPY DATA TO VDP BUFFER
       DATA 8
       LI    R0,>1000            PREPARE TO COPY FROM VDP TO PATRN
       LI    R1,PATRN            CPU BUFFER ADDRESS
       LI    R2,>600             1536 BYTES TO COPY
       BLWP  @VMBR               COPY BUFFER
       LI    R0,>0800            NOW COPY TO PATTERN TABLE
       LI    R1,PATRN            ADDRESS OF CPU BUFFER
       LI    R2,>600             1536 BYTES TO COPY
       BLWP  @VMBW               COPY TO TABLE
       B     @CHECK              BACK TO THE BEGINNING
*
* SUBROUTINE TO PERIODICALLY CHANGE SPRITE COLORS
*
CHECKS AI    R4,>100             ADD 256 TO R4
       JEQ   CHANGE              WHEN R4 REACHES 0, CHANGE COLOR
       DEC   R0                  DEC COUNTER
       JNE   CHECKS              IF NOT 0 ADD ANOTHER 256
       JMP   RETURN              BACK TO MAIN PROGRAM
CHANGE SWPB  R3                  SWITCH COLOR BYTES IN R3
       MOV   R3,R1                PUT R3 IN R1
       LI    R0,>383             ADDRESS OF SPRITE #0 COLOR IN VDP RAM
       BLWP  @VSBW               WRITE MOST SIG BYTE OF R1
RETURN RT                        BACK TO MAIN PROGRAM
*
* DEBOUNCE SUBROUTINE
*
DEBNC  MOVB  @ONE,@UNIT          KEY UNIT TO CHECK
       BLWP  @KSCAN              SCAN KEYBOARD
       CB    @FIRE,@NOKEY        IS NO KEY PRESSED?
       JNE   DEBNC               IF A KEY IS PRESSED, CHECK AGAIN.
       RT                        GO BACK TO MAIN PROGRAM
*
* SUBROUTINE TO DETERMINE COLOR FOR ARROW
*
DOTCOL CLR   R1                  INITIALIZE R1 TO RECEIVE DOT COLUMN
       LI    R0,>301             VDP ADDRESS OF DOT COLUMN
       BLWP  @VSBR               READ BYTE FROM ATTRIBUTE TABLE
       SWPB  R1                  MAKE IT LEAST SIG BYTE
       AI    R1,>07              ADD OFFSET FOR POINT OF ARROW
       SRL   R1,4                DIVIDE BY 16
       RT                        RETURN
*
* 'END START'
*
AUTO   END   MARKER              AUTOSTART
```

MINI MEMORY CARTRIDGE



There's More There Than Meets the Eye

You know, looks can be deceiving. Who'd suspect that a bespectacled, mild-mannered reporter for the *Daily Planet* could leap over tall buildings with a single bound? In the same way, there's more to TI's Mini Memory Command Cartridge than meets the eye. What appears to be a normal, garden-variety Command Cartridge, however, really converts your TI Home Computer from a good BASIC machine to a trim and efficient assembly language instrument.

Even the name is a clever disguise: "Mini" Memory, indeed! If you believe that there's just a tiny bit of memory in there, you probably believe that the Trojan Horse was nothing more than an overgrown hobby-horse! This cartridge actually has 14K bytes of memory: 4K of RAM, 4K of ROM and 6K of GROM.

RAM (read/write) memory is used by your computer to store your programs. And you know that any program you write disappears from the computer's memory when you shut the computer off. But *Mini Memory* has a surprise for you: When you shut the computer off and unplug the cartridge, your programs don't disappear from the cartridge's RAM. A battery inside the cartridge feeds a trickle of current to the CMOS devices—which are real power misers—and keeps them alive. And now you can carry your programs around with you, plug them in, and *instantly* load them—no cassettes, no diskettes, no messy cables, no long waits.

But there's more yet. Besides battery-backed RAM, this cartridge also has 4K bytes of ROM (Read-Only Memory) and 6K bytes of GROM (Graphics Read-Only Memory). The ROM and GROM give you seven additional TI BASIC subprograms, as well as access to many system routines from assembly language programs. The ROM also contains a powerful program debugger, EASY BUG, which can help you exterminate those pesky "logic vermin" which infest programs.

At this point, you may be saying to yourself, "What good does all this Assembly Language access and debugging stuff do for me, anyway, without an assembler?" Glad you asked. The Mini Memory Command Cartridge comes with an assembler cassette. You can load this assembler into memory, enter assembly language statements, and have the assembler translate them into TMS9900 object code.

Let's explore this cornucopia one item at a time.

FILE STORAGE

Probably most persons will use the Mini Memory cartridge most often for temporary storage of programs and data. You can think of the Mini Memory cartridge as a very fast-access storage device. [See "Getting Down to Business" for a tutorial on random access files.—Ed.]

When you have the *Mini Memory* Command cartridge plugged in, the 4K-byte RAM has the file name MINIMEM for TI BASIC program and data storage. The RAM occupies physical addresses 28672 through 32767 (hexadecimal 7000 through hexadecimal 7FFF). You can save programs in this file and load programs from it. (For example, to save a TI BASIC program, just enter the command SAVE MINIMEM.) You can also store data in this file using the file specification available for any TI BASIC file. For example, the following statements open the Mini Memory file and store data values in the file.

```
OPEN #3 :"MINIMEM",RELATIVE,FIXED,
UPDATE,INTERNAL
PRINT #3: A,B,C,D
```

With the Mini Memory cartridge you can also access a second new file. EXPMEM2 is the name of a 24K-byte memory file located in the 32K Memory Expansion unit. EXPMEM2 is available, however, only if you have the Memory Expansion unit connected to your computer and turned on.

ADDITIONAL TI BASIC SUBPROGRAMS

Seven additional TI BASIC subprograms are yours with the Mini Memory cartridge. These subprograms are PEEK, PEEKV, POKEV, CHARPAT, INIT, LOAD, and LINK.

The PEEK subprogram reads bytes of CPU RAM data and copies the data directly into TI BASIC variables. For example, the statement:

```
CALL PEEK (8192,A,B,C,(8))
```

reads three bytes of data starting at address 8192, and assigns the values read to the variables A, B, and C(8).

The PEEKV subprogram reads bytes from VDP RAM. It works exactly like PEEK, except PEEKV accesses VDP RAM instead of CPU RAM.

The POKEV subprogram stores data values into VDP RAM. For example,

```
CALL POKEV(784,30,30,30)
```

writes the value 30 to VDP RAM locations 784, 785, and 786.

The CHARPAT subprogram reads a 16-character pattern identifier that specifies the pattern of a character code. For example,

```
CALL CHARPAT(68,D$)
```

places the pattern defining character code 68 in the string variable D$.

The three TI BASIC subprograms INIT, LOAD, and LINK interface Assembly Language programs and TI BASIC programs.

The INIT subprogram initializes the CPU memory for Assembly Language programs. The LOAD subprogram loads Assembly Language object files into CPU memory and it loads data into the CPU memory.

There are two forms of the LOAD subprogram. One form is used to load an object file from a storage device into memory, and the second form is used to load data directly into CPU memory. For example, the statement

```
CALL LOAD ("DSK1.DEMO")
```

loads the file DEMO from the diskette in Disk Drive 1.

The second form of the LOAD subprogram is a POKE function for CPU RAM. For example, the statement

```
CALL LOAD (8197,85,40)
```

loads the value 85 into memory location 8197 and the value 40 into memory location 8198.

The LINK subprogram passes control and, optionally, a list of parameters from a TI BASIC program to an Assembly Language program. For example, the statement

```
CALL LINK ("PROG1",A,E(9))
```

passes control from a TI BASIC program to an Assembly Language program named PROG1 and passes the variables A and E(9) to the program.

ACCESS TO SYSTEM ROUTINES

The utility routines resident in the Mini Memory Command Cartridge can be called from an Assembly Language program to access machine resources and interface with the TI BASIC interpreter. It's fair to warn you that the use of these routines requires a knowledge of the routines themselves and the organization of data used by the routines. You can get additional information about these routines from the Editor/Assembler owner's manual (available separately).

Two types of access programs are resident in the Mini Memory Command Cartridge. One program contains a collection of system utilities with which to link to ROM/GROM routines, perform a keyboard scan, access the VDP, etc. The individual utility programs are classified as either *Standard Utility* programs or *Extended Utility* programs.

A second program contains TI BASIC interface utilities with which an Assembly Language program can access variables passed through a CALL LINK statement in a TI BASIC program. This program also contains an error-handling utility to return exceptions to a TI BASIC program.

STANDARD UTILITY PROGRAMS

The following standard system utilities become accessible with the *Mini Memory* Command Cartridge:

—VDP Single Byte Write—Write a single-byte value to a specified VDP RAM address.

—VDP Multiple Byte Write—Write multiple bytes from CPU RAM to VDP RAM.

—VDP Single Byte Read—Read a single byte from a specified VDP RAM address.

—VDP Multiple Byte Read—Read multiple bytes from VDP RAM into CPU RAM.

—VDP Write to Register—Write single-byte value to any of the VDP RAM registers.

—Keyboard Scan—Scan the keyboard and return a key-code and status. This routine can also read the position of the Wired Remote Controller.

EXTENDED UTILITY PROGRAMS

Extended utilities are provided to access routines in the console GROMs and ROMs. These utilities are GPLLNK (link to GPL routines in GROM), XMLLNK (link to routines in ROM), and DSRLNK (link to Device Service Routines).

GPLLNK Routines

The GPLLNK routines are as follows:

—Load Standard Character Set—Load the standard character set into VDP RAM

—Load Small Character Set—Load the small character set (for the 40-column Text Mode) into VDP RAM.

—Execute Power-Up Routine—Initialize the system as if the computer had just been turned on.

—Accept Tone—Issue an accepting tone for input.

—Bad Response Tone—Issue a bad-response tone warning.

—Bit Reversal Routine—Provide a mirror image of a byte of information.

—Cassette Device Service Routine—Access a cassette tape recorder/player as a storage device.

—Load Lower Case Character Set—Load the lower-case character set into VDP RAM.

The following floating point routines are also available through GPLLNK:

—Convert a floating-point number to an ASCII string.

—Compute the greatest integer contained in a value.

—Raise a number to a specified power.

—Compute the square root of a number.

—Compute the inverse natural logarithm of a value.

—Compute the natural log of a number.

—Compute the cosine of a number.

—Compute the sine of a number.

—Compute the tangent of a number.

—Compute the arctangent of a number.

XMLLNK Routines

Routines in the console ROM can be accessed through the XMLLNK routine. The following routines can be called from an Assembly Language program using XMLLNK:

—Floating-point addition.

—Floating-point subtraction.

—Floating-point mutiplication.

—Floating-point division.

—Floating-point compare.

—Floating-point stack addition.

—Floating-point stack subtraction.

—Floating-point stack multiplication.

—Floating-point stack division.

—Floating-point stack compare.

—Convert a string to a number.
—Convert a floating-point format number to an integer.
—Push a value onto the value stack.
—Pop a value from the value stack.
—Convert an integer number to floating-point format.

DSRLNK Routines

DSRLNK links an Assembly Language program to a Device Service Routine (DSR) or a subprogram in ROM. As with GPLLNK and XMLLNK, TI cautions you to make sure you know what you are doing before using DSRLNK. [A DSR is a machine language program that TI has burned into ROMs found in each of its peripherals. Since each peripheral contains its own custom "operating system," the TI-99/4A did not have to be designed to anticipate future peripheral requirements.—Ed.]

TI BASIC INTERFACE UTILITIES

TI BASIC interface utilities allow an Assembly Language program to read or assign values to variables passed in a parameter list from a CALL LINK statement in a TI BASIC program. These utility routines include argument-passing utilities and an error-reporting utility.

The following are the TI BASIC interface utilities:

—Assign a numeric value to a numeric variable.
—Assign a string to a string variable.
—Retrieve the value of a numeric parameter.
—Retrieve the value of a string parameter.
—Report an error. (The Assembly Language program can report any existing TI BASIC error or warning message upon returning to TI BASIC.)

EASY BUG DEBUGGER

Also inside the *Mini Memory* cartridge's ROM is EASY BUG. EASY BUG is a versatile program development tool with which you can (1) debug your Assembly Language programs, (2) access the input/output ports of the computer, (3) load programs, and (4) store programs. And it really is easy to use. With EASY BUG, you can inspect and (optionally) modify the contents of CPU and VDP memory, display the contents of ROM, run Assembly Language programs from EASY BUG, directly access the peripheral devices which are connected to the computer via the 9900 microprocessor's serial I/O port (the CRU), and save or load programs on cassette.

LINE-BY-LINE SYMBOLIC ASSEMBLER

A line-by-line symbolic assembler on a cassette tape is supplied with the Mini Memory cartridge. It assembles Assembly Language statements and stores the object code directly into the 99/4A's CPU RAM. You can make both forward and backward references to one- or two-character labels with the Assembler. Each source statement you enter is immediately assembled into object code and stored into memory. Because some source code is retained in a nine-page text buffer, you can scroll the screen to review previously entered lines of source code by pressing the up- and down-arrow keys. The source program cannot be saved, however.

The Line-by-Line Assembler occupies about 2K bytes. When it is loaded into the Mini Memory cartridge's 4K byte RAM, you still have about 2K bytes of memory for your Assembly Language program.

Assembler Directives

The Assembler recognizes seven directives:

—The AORG (Absolute Origin) directive establishes the location counter value to set the starting address of assembled code.
—The BSS (Block Starting with Symbol) directive reserves a block of initialized memory.
—The DATA (Data Initialization) directive initializes a word or words of memory to a specific value.
—The END (End Program) directive terminates the assembler and causes a display of the number of unresolved references, if any.
—The EQU (Equate) directive defines a value for a symbolic constant.
—The SYM (Symbol Table Display) causes a display of all symbols and their values in the program.
—The TEXT (String Definition) directive causes a string of characters to be translated into their ASCII code and stored as a part of a program.

[Rather than being strictly a part of the internal logic of your program, assembler directives are commands which direct the Assembler to perform certain operations at assembly time.—Ed.]

DEMONSTRATION PROGRAM

Along with the Line-by-Line Assembler on the cassette is an Assembly Language demonstration program called LINES which draws a colorful line design on the screen. The LINES program can be run only on the TI-99/4A Home Computer, however, because it requires the enhanced graphic processor contained on the TI-99/4A.

OPERATION

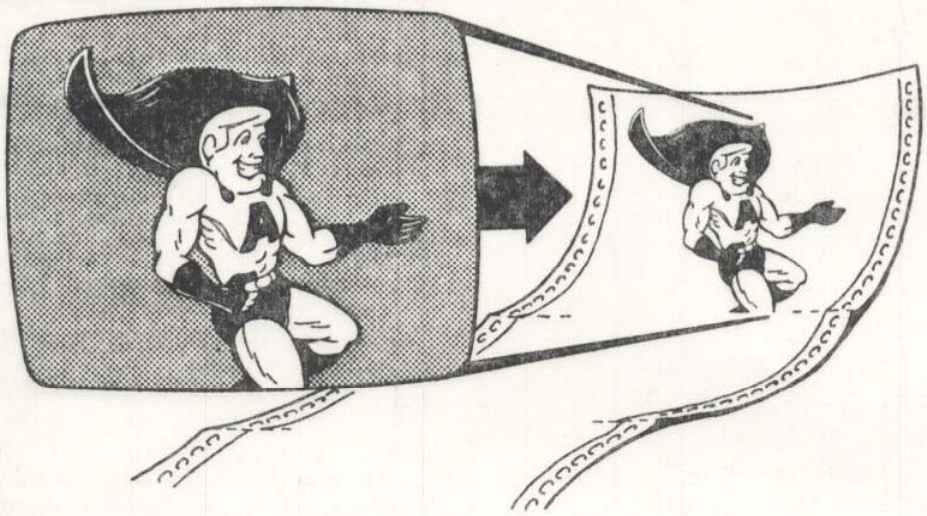
TI has a knack for creating complex and versatile programs that are still simple to operate; they've definitely done it again with the Mini Memory Command Cartridge. When you plug in the cartridge, turn on the computer, and pass the opening credits on the Master Title Screen, you are presented with a simple, three-choice selection screen. You can choose TI BASIC, EASY BUG, or MINI MEMORY.

If you select MINI MEMORY, you are presented with a second three-choice selection screen. You can choose to load an object program into memory and run it, run a previously loaded program already in memory, or reinitialize the cartridge to prepare it for loading new programs or storing data. Pick a number, pluck a key, and you're off and running. It's as easy as eating oatmeal cookies!

CONCLUSION

This has got to be one of the best deals around. 4K bytes of RAM with battery backup assure that all the good stuff stored in the RAM is not lost when you turn off the console or even when you remove the cartridge. 10K bytes of ROM and GROM give you seven additional TI BASIC subprograms (including PEEK and POKE), access to system routines from Assembly Language, and routines to allow you to interface Assembly Language programs to TI BASIC. You've got a user-friendly program debugger, a symbolic line-by-line assembler, and a captivating graphics demonstration program. All of this, plus 84 pages of documentation, for $99.95 (suggested retail price). With all this to offer, it's really not too hard to see why there's definitely more to the Mini Memory Command Cartridge than meets the T-eye .

A Screen Printing Utility



PART 1: Design Considerations

One of the best features of the TI-99/4A computer is its graphics capability. The programmer can create a huge variety of screens by using the simple character-definition commands of TI BASIC. Wouldn't it be nice to dump those screens to your non-thermal printer? This two-part article presents a method for doing this on the TI-99/4 impact printer. Part I discusses the theory behind the screen dump. Part II will provide the Assembly Language subroutine itself.

I should mention that the 99/4A has an improved video processor (TMS9918A) which allows you to define up to 768 unique characters on the screen. However, this bit-map mode requires an extra 12K of memory to hold the larger tables needed. We'll limit ourselves to the Graphics I, or standard mode, in this discussion.

Approach—in English

The video screen contains 768 character positions, arranged in 24 rows of 32 characters. Each character is composed of an 8 × 8 dot matrix, giving you a screen of 192 × 256 dots. The screen dump program will reproduce the screen dot-for-dot on the printer.

With bit-image mode selected, the TI-99/4A prints characters which are one dot wide and 8 dots high. Since the screen characters are also 8 dots high, each screen character can be represented by 8 TI-99/4A bit-image characters, for a total of 64 possible dots per screen character.

Accessing the Screen Image

The contents of the screen are stored in VDP RAM. Since we are not concerned with color here, only two of the screen tables in VDP RAM are of interest. The first is the Screen Image Table, which starts at default address >0000 and contains 768 bytes. Each byte corresponds to the character position on the screen and contains the character number occupying that screen position. VDP RAM addresses >0020 through >003F correspond to the second screen row, and so on. Since each character number is contained in one byte, you can see that the character numbers must be between >00 and >FF, or decimal 0 through 255.

The second table we'll need is the Pattern Descriptor Table, which starts at VDP RAM address >0800 by default. This table contains the dot patterns for each of the 256 characters which can be in use. The BASIC subprogram CHAR, which is used to define dot patterns for characters, stores patterns in this table. Since a character pattern takes 8 bytes to define, and there can be up to 256 different characters, the Pattern Descriptor Table occupies 2084 bytes of VDP RAM.

Figure 1 shows the relationship between these two tables. For a given screen ROW and COLUMN, the VDP RAM address of the corresponding character number is given by (ROW − 1) * 32 + COLUMN − 1. Once you have obtained this character number, you can use it to index to the correct spot in the pattern Descriptor Table. The offset in this table is just 1024 + (N − 32)*8 in decimal, since each pattern description is 8 bytes long. Figure 1 shows an example of finding the pattern for the home position (ROW 1, COLUMN 1) on the screen. The character number resides in the Screen Image Table at address 0. If the home character on the screen is "A", then VDP RAM address 0 contains the value 65 or >41. From the offset in the Pattern Descriptor Table, we get VDP RAM address >800 + >200 = >0A00. The eight bytes starting at >0A00 in VDP RAM contain the pattern for the character "A". You can see that for our purposes, the contents of the Screen Image Table are just intermediate, though necessary, data. The character pattern is what we're really after.

The 8-byte character pattern represents the dot pattern which appears on the screen in what I'll call *row-wise* form. The top portion of Figure 2 illustrates this for the character "A". The first byte of the pattern represents the first row of the dots which comprise the character. The hexadecimal notation is just a shorthand way to group four bits at a time, with bits of value 1 standing for dots which are turned on in the character.

Translating the Characters to TI-99/4A Format

The TI-99/4 printer constructs its bit-image output in a different way. It uses what I'll call *column-wise* form. It still takes 8 bytes to produce the same character, but each byte of data passed to the printer represents a column (rather than a row) of dots in the finished character. The bottom of Figure 2 illustrates this. If we think of the character's dot pattern as an 8×8 matrix, then the translation from TI internal format to TI-99/4A printer bit-image format is equivalent to transposing the matrix. We can't really treat each character pattern as a 64-bit matrix because 9900 Assembly Language does not have a BIT data type, but we can base the logic of the program on this idea.

Figure 1.

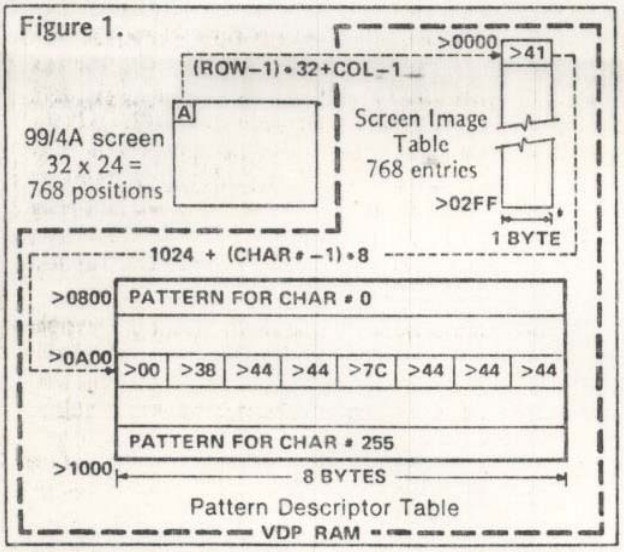


Figure 2. TI-994A character pattern from Pattern Descriptor Table

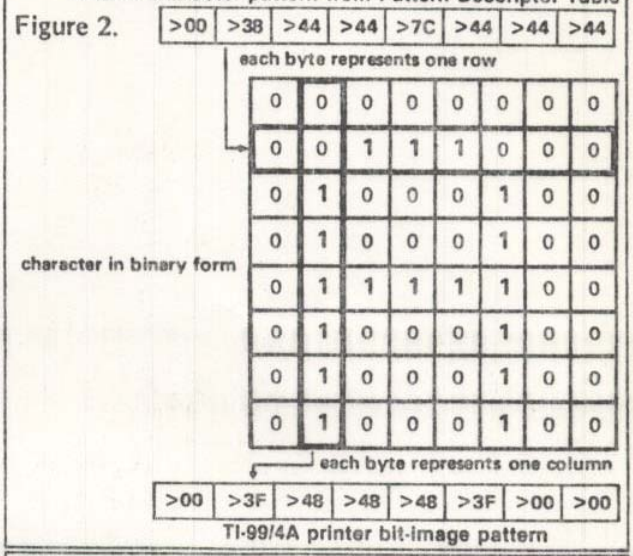
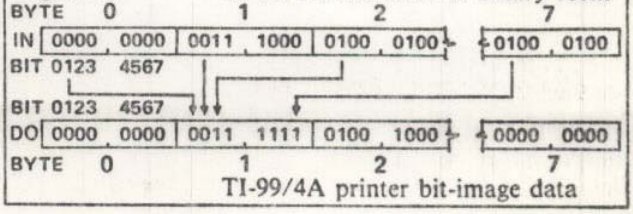


Figure 3. TI-99/4A character in binary form



Program Outline

The screen dump program reads the Screen Image Table one byte at a time starting at the top (VDP RAM address 0). The value of each byte is used to calculate the position of the character pattern, and the 8-byte pattern is obtained from the Pattern Descriptor Table. These 8 bytes will be manipulated to produce 8 bytes of information encoded for the TI-99/4 printer. Figure 3 shows how the bits of the TI-99/4A character pattern are rearranged to form bit-image data for the printer. Notice that the data at byte M, bit N is moved to byte N bit M—or transposed. The program will also have to send certain control characters for bit-image mode to the printer.

PART 2: Screen Dump

The Assembly Language subroutine for dumping 99/4 screens to the TI-99/4 impact printer is designed to be called from console BASIC, and can be entered into your system using either the Editor/Assembler or the Line-by-Line Assembler in the Mini Memory Command Cartridge.

VDP RAM Under Console BASIC

When the TI-99/4A is under control of the BASIC interpreter, VDP RAM contains two areas of interest here. VDP RAM addresses >0000 – >02FF (0 – 767 in decimal) contain the character numbers associated with each screen position. The character patterns for character numbers 32 – 159 start at VDP RAM address >0400 (1024). In the Pattern Descriptor Table address the 8-byte character pattern corresponding to a character number N is 1024 + (N – 32) * 8 in decimal.

The dump subroutine (called DUMP) uses these facts. Starting with VDP RAM address 0, DUMP gets the screen character number and uses it to calculate the VDP RAM address of the associated character pattern. It then reads the 8-byte character pattern, transposes the matrix, and writes the resulting 8 bytes to the printer. DUMP performs this process on each successive byte of screen RAM, up to and including VDP RAM address >02FF (767).

DSRLNK and Printer Output

The actual output to the printer is done by means of a built-in Extended Utility Routine called DSRLNK.

Before calling DSRLNK, the Assembly Language subroutine must set up a Peripheral Access Block (PAB) in VDP RAM. Here is the format of the PAB we'll use for the printer:

BYTE#	CONTENTS
0	I/O opcode: >00 = open >01 = close >03 = write
1	Flag/status byte. >12 is the code for sequential file, output operation, DISPLAY type data and variable length records.
2, 3	Data buffer address in VDP RAM. We'll use >1E00.
4	Logical record length.
5	Number of characters to write.
6, 7, 8	Not used here.
9	Length of file descriptor which follows.
10–35	File descriptor. We'll use RS232.PA=O. DA=8.BA=9600.CR

We'll put the PAB in VDP RAM starting at address >1D00 (hereafter called V1D00), and we'll put the data area containing the actual data for output to the printer at V1E00. These addresses could have been elsewhere in VDP RAM, as long as the locations chosen were not used by something else.

To perform a printer operation, the program must do the following:

1. Build the PAB in VDP RAM.
2. Put the address of the length of the file descriptor (byte 9 of the PAB) into CPU RAM address >8356.
3. Call DSRLNK.

You'll notice that the call to DSRLNK must be followed by a word (two bytes) containing the value 8, which means that you want to link to a Device Service Routine (DSR).

RS232 Considerations

Since the DUMP subroutine uses the RS232 interface to communicate with the printer, some additional code is needed to save and restore the address of the GROM. This is because the GROM address is changed when the RS232 DSR is used. At the beginning of the DUMP subroutine, the GROM address is obtained one byte at a time from the GROM Read Address at location >9802. The GROM address increments itself when the first byte is read (actually moved) from the GROM Read Address. This makes the second byte of the GROM address one too big, so it must be decremented by DUMP. Just before returning to BASIC, the DUMP subroutine restores the GROM address by moving it to the GROM Write Address at location >9C02, again one byte at a time.

Linkage to Console BASIC

A console BASIC program invokes the DUMP subroutine by the statement CALL LINK("DUMP"). DUMP returns to the BASIC program by branching to the contents of register 11 (R11). Just before returning to BASIC, the DUMP subroutine clears the error byte at @>837C (sets it to 0). Failure to clear this byte can result in an undeserved INCORRECT STATEMENT error when you return to BASIC.

Transposing the 8x8 Character Matrix

Once a screen character's 8-byte pattern has been read into CPU RAM (at label IN), the DUMP subroutine uses the following technique to build the 8 bytes of output at label DO.

The first byte of DO is composed of the first bit of each of the 8 bytes starting at IN, the second byte of DO is composed of each second bit of the bytes at IN, and so on. Figure 2 of Part One shows the bit movements for the pattern character of an "A".

DO is built from left to right, and R4 is used to hold each byte of DO as it is built. R4 is cleared before each byte is built, so DUMP has to turn on any bits necessary.

To tell if a certain bit of IN is on, DUMP compares the value of the byte containing the bit in question to a power of 2. To see how this works, consider the byte containing >82 (130 in decimal, 1000 0010 in binary). The leftmost bit of the byte is on; in fact, the leftmost bit would be on in any byte containing >80 (128) through >FF (255). In other words, we could test for the leftmost bit's being on by comparing the value of the byte to decimal 128 (2 to the 7th power); if the value is less than 128, we wouldn't have to turn on the corresponding output bit.

This technique can be used to test any bit of a byte for our purposes, using the appropriate power of 2. The second-to-leftmost bit can be tested against 64, its neighbor to the right against 32, and so on down to 1 for the rightmost bit. This works because we'll be considering the bits from left to right in each byte. After each bit is tested, it must be turned off (in CPU RAM, not on the screen) so that it doesn't interfere with the test of the following bit. To see this, consider the byte containing >82 (130) again. If we want to determine if the second-to-leftmost bit is on, we would compare the byte to 64. You can see that it would pass the test, even though the bit in question is not on! However, if we had reset the leftmost bit to 0 after testing it previously, the byte would now contain >02 instead of >82, and the test would fail, as it should.

Once we have decided that an input bit is on, we must set on the corresponding bit in R4. This is done by adding the appropriate power of 2 to R4. To turn on the leftmost bit, add 128; to turn on the rightmost bit, add 1. Remember that the byte being built is in the right half (LSB, or least significant byte) of R4.

DUMP uses R5 to contain the power of 2 for testing whether the input bit is on, and R6 to contain the power of 2 for setting the bit on for output. The LSB of R7 contains the input byte being tested, and the most significant byte of R7 is filled with zeros. This allows DUMP to use the simpler and more plentiful register instructions, and to completely avoid having the leftmost bit of a byte interpreted as a sign bit.

Printer Consideration

DUMP writes one full screen line to the printer at a time. Before each line, the program must write a 4-byte control sequence to put the printer in graphics mode and tell it how many graphics characters are coming next. This sequence is >1B, >4B, >FF, and >00. The last two bytes mean 255 characters will be written, with the order

of the bytes being reversed for evaluation (>00FF, or 255).

The program issues a carriage return and line feed only after each of these writes, that is, at the end of each screen line. DUMP uses the CZC (Compare Zeros Corresponding) instruction to accomplish this. R9 contains the position in VDP RAM of the next screen character number. Positions 0 – 31 (>00 – >1F) of VDP RAM correspond to the characters on line 1 of the screen; positions 32 – 63 (>20 – >3F) correspond to characters on line 2, etc. The CZC instruction occurs right after R9 is incremented and before the corresponding screen character is decoded. Therefore, the carriage return and line feed should be written whenever R9 is an even multiple of 32. The CZC instruction uses a mask of >1F (0000 0000 0001 1111 binary). If R9 is a multiple of 32, then its last five bits will all be zero. Notice that the mask has only the last five bits turned on. Under these circumstances, the CZC instruction sets the equal status bit on if and only if the last 5 bits of R9 are all zero, that is, if and only if R9 contains an even multiple of 32. The JNE instruction which follows the CZC instruction causes the program to skip outputting the carriage return and line feed when R9 does not contain a multiple of 32.

Left to its own devices, the printer will respond to a line feed by spacing down 1/8" or 1/6". This would leave blank stripes in the screen dump. The sequence ESCAPE A 8 is written by DUMP to tell the printer to space down only 8/72" on each line feed. This produces a continuous image.

Mini Memory Considerations

To enter the DUMP subroutine via the Line-by-Line Assembler, do the following:

1. Select MINI MEMORY and then RUN from the first two menus.

2. Enter NEW as the program name.

3. When the Line-by-Line Assembler screen appears, type a space, then AORG, another space, >7D14, and then press [ENTER.] (From now on the spaces will be assumed.) This sequence lets you start the program at >7D14 instead of the traditional >7D00.

4. Enter the program as shown in Listing #1. Enter only the label (if any), opcode, and operands. Don't enter END yet.

5. Put the entry point for DUMP into the DEF/REF table by entering the following lines:

```
AORG >7FE8(CR)
TEXT 'DUMP   '(CR)
DATA >7D14(CR)
```

6. Set the last used address in Mini Memory by entering:

```
AORG >701C(CR)
DATA >7F02(CR)
```

7. Indicate that you are finished by entering:

```
END(CR).
```

The system should show that you have no unresolved references. Press enter twice, and then QUIT the Line-by-Line Assembler.

8. Enter EASY BUG from the master menu.

9. Press any key to bypass the instruction screen.

10. Enter S7000 when the system prompts with ? and then 7FFF when the system prompts TO? This tells the system to save the contents of the Mini Memory to cassette tape. Just follow the instructions presented by the computer after this, and then QUIT EASY BUG when you have saved and checked your tape.

You are now ready to use the DUMP subroutine. The sample BASIC program in Listing #2 just draws a screen and then waits for you to press the P key, at which point DUMP is called to print out the screen. You can incorporate DUMP into your own programs in any way you choose. Happy dumping!

Listing 1 *Dump*

```
          AORG  >7D14
          MOVB  @>9802,@S1    GET MSB OF GROM ADDR INTO S1
          SWPB  @S1
          MOVB  @>9802,@S1    GET LSB OF GROM ADDR
          SWPB  @S1
          DEC   @S1             CORRECT FOR AUTO-INCREMENT
          LI    0,>1D00
          LI    1,PD
          LI    2,36
          BLWP  @>6028        WRITE PAB TO VDP RAM
          LI    6,>1D09
          MOV   6,@>8356      POINT TO DEVICE NAME LENGTH
          BLWP  @>6038        DSRLNK TO OPEN PRINTER
          DATA  8
          LI    10,>0400
          MOV   10,@>7DEA
          LI    0,>1D00
          LI    1,>0300
          BLWP  @>6024        PUT WRITE OP CODE IN PAB
          LI    0,>1D05
          LI    1,>0400
          BLWP  @>6024        PUT LENGTH OF 4 IN PAB
          LI    0,>1E00
          LI    1,E2          PUT CODE FOR CARRIAGE RTN &
          LI    2,4           8/72" VERTICAL LINE SPACING
          BLWP  @>6028        IN DATA BUFFER.
          MOV   6,@>8356      POINT TO DEVICE  NAME LENGTH
          BLWP  @>6038        DSRLNK-CHANGE VERT SPACING
          DATA  8
          LI    10,50         DELAY
          DEC   10
          JNE   $-2
          CLR   9             R9->NEXT SCREEN POSITION
L0        MOV   9,0
          BLWP  @>602C        PUT BYTE OF SCREEN RAM IN R1
          SRL   1,8           SHIFT TO LSB OF R1
          AI    1,-128        ADJUST FOR BASIC
          SLA   1,3           *8
          AI    1,1024        PTRN ADDR=1024+(CHAR#-32)*8
          MOV   1,@
          LI    1,IN
          LI    2,8
          BLWP  @>6030        PUT PATTERN INTO IN
          LI    5,128         R5 = BIT#
          CLR   8             R8 = OFFSET FOR DO
L3        LI    6,128         R6 =  BYTE#
          CLR   3             R3 = OFFSET FOR IN
          CLR   4             R4 IS FOR BUILDING NEXT CHAR
L2        CLR   7
          MOVB  @IN(3),7      R7 HOLDS BYTE BEING DECODED
          SWPB  7             PUT BYTE IN LSB OF R7
          C     7,5           IS BIT ON?
          JLT   L1            NO
          A     6,4           YES,TURN OUTPUT BIT ON
          S     5,7           TURN OFF INPUT BIT
          SWPB  7             PUT BYTE IN MSB OF R7
          MOVB  7,@IN(3)      REWRITE TO IN
L1        INC   3             POINT TO NEXT INPUT BYTE
          SRA   6,1           /2
          JGT   L2            DO NEXT BYTE, IF MORE
          SWPB  4             PUT OUTPUT BYTE IN MSB OF R4
          MOVB  4,@DO(8)      STORE AT DO
          INC   8             POINT TO NEXT BYTE OF DO
          SRA   5,1           /2
          JGT   L3            CONSTRUCT NEXT OUTPUT BYTE
          LI    0,>1D05
          LI    1,>0000
          BLWP  @>6024        PUT LENGTH OF 4 IN PAB
          LI    0,>1E00
          LI    1,E1
          LI    2,4
          BLWP  @>6028        PUT ESC K SEQ. IN DATA BUFF
          LI    6,>1D09
          MOV   6,@>8356      POINT TO DEVICE NAME LENGTH
          BLWP  @>6038        DSRLNK TO WRITE ESC K SEQ.
          DATA  8
          LI    10,>0000
          MOV   10,@>7DEA
          LI    0,>1D05
          LI    1,>0800
          BLWP  @>6024        PUT LENGTH OF 8 IN PAB
          LI    0,>1E00
          LI    1,DO
```

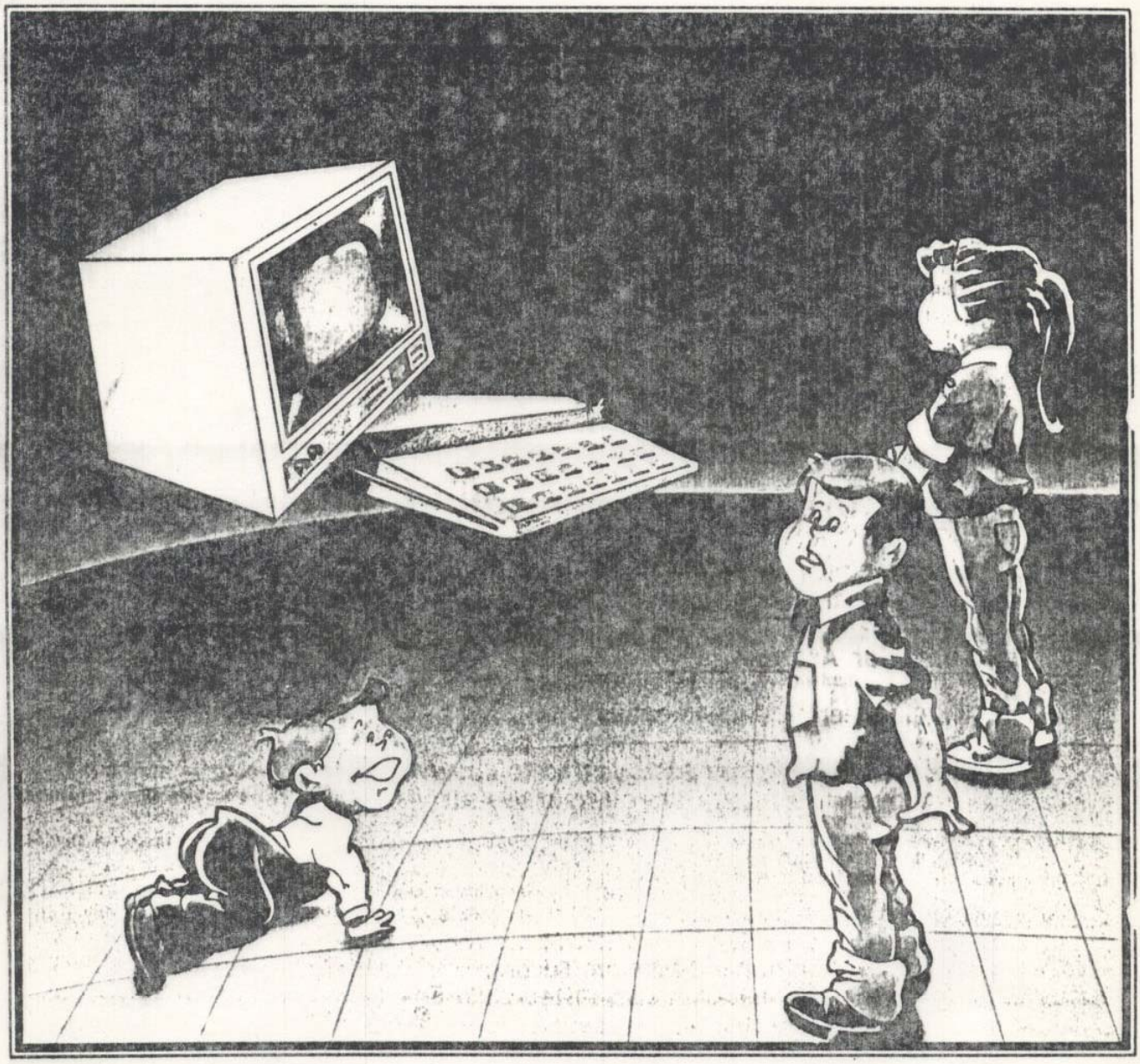
Listing 1 *Dump* continued

```
        LI    2,8
        BLWP  @>6028          PUT DO INTO DATA BUFFER
        MOV   6,@>8356        POINT TO DEVICE NAME LENGTH
        BLWP  @>6038          DSRLNK TO OUTPUT 8 CHARS
        DATA  8
        LI    10,50           DELAY
        DEC   10
        JNE   $-2
        INC   9               POINT TO NEXT SCREEN POSITION
        CI    9,767           DONE WITH SCREEN YET?
        JGT   L4              YES
        CZC   @MK,9           NO.  ARE WE AT END OF LINE?
        JNE   L0              NO-DO NEXT  SCREEN CHARACTER
        LI    0,>1D05         YES-OPUTPUT CR LF
        LI    1,>0200
        BLWP  @>6024          PUT LENGTH OF 2 IN PAB
        LI    0,>1E00
        LI    1,CR
        LI    2,2
        BLWP  @>6028          PUT CR LF INTO DATA BUFFER
        MOV   6,@>8356        POINT TO DEVICE NAME LENGTH
        BLWP  @>6038          DSRLNK TO OUTPUT CR LF
        DATA  8
        LI    10,>0400
        MOV   10,@>7DEA
        JMP   L0              DO NEXT SCREEN CHARACTER
L4      LI    0,>1D00         COME HERE WHEN FINISHED DUMP
        LI    1,>0100
        BLWP  @>6024          PUT CLOSE OP CODE IN PAB
        MOV   6,@>8356        POINT TO DEVICE NAME LENGTH
        BLWP  @>6038          DSRLNK TO CLOSE PRINTER
        DATA  8
        LI    10,50           DELAY
        DEC   10
        JNE   $-2
        MOVB  @S1,@>9C02      RESTORE SAVED DATA TO GRMWA
        SWPB  @S1
        MOVB  @S1,@>9C02
        SB    @>837C,@>837C   CLEAR ERROR BYTE FOR BASIC
        LI    10,50           DELAY
        DEC   10
        JNE   $-2
        B     *11             RETURN TO BASIC
IN      BSS   8               AREA FOR SCREEN PATTERN
DO      BSS   8               AREA  FOR PRINTER PATTERN
MK      DATA  >001F           MASK FOR EOL TEST
PD      DATA  >0012,>1E00,>FF00,>0000,>001A
*                             PAB DEFINITION
        TEXT  'RS232.PA=O.DA=8.BA=9600.CR'
*                             DEVICE NAME
CR      DATA  >0D0A           CR LF
E1      DATA  >1B4B,>FF00     ESC K GRAPHICS SEQUENCE
S1      BSS   2               SAVE AREA
E2      DATA  >0D1B,>4108     CR AND ESC A VERT SPACING
        END
```

Listing 2 *Screen Dump*

```
100 CALL CLEAR
110 CALL CHAR(96,"183C7EFFFF7E3C18")
120 CALL HCHAR(1,1,96,768)
130 CALL KEY(0,RVAL,STAT)
140 IF STAT=0 THEN 130
150 IF RVAL<>80 THEN 130
160 CALL LINK("DUMP")
170 END
```

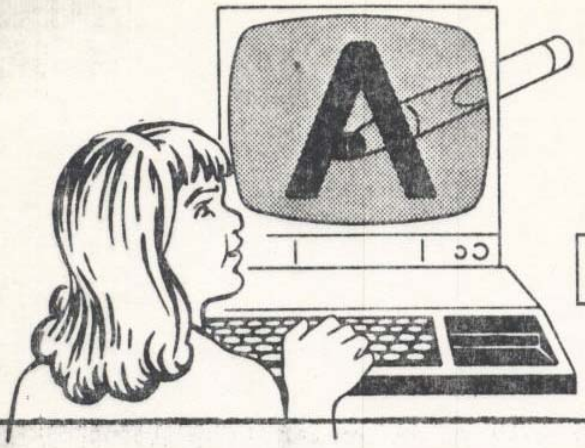
6
Computer-Assisted Instruction



6

Computer-Assisted Instruction

Preschool to college and beyond . . .
Is the computer a learners' magic wand?



TI BASIC

PRESCHOOL BLOCK LETTERS AND DATA COMPACTION

Most kids aged 100 weeks to 100 years old are fascinated by computers. And small kids are really fascinated by a computer's video screen; it's like a TV, but they can control it. When mine were just learning the alphabet, they would wriggle in between Dad and the computer, then push the "A" key so an "A" would pop onto the screen. But the popping part was the problem. A computer doesn't draw (write) letters on the video screen—it "pops" up the whole letter at once. (Or at least to our slow eyes it "pops" the whole letter at once.) But kids can't just squeeze a crayon and have a letter pop onto a piece of paper. They have to learn a series of hand motions in order to make a recognizable "A" on a piece of paper.

But just maybe the computer could make large letters by popping short line segments in sequence onto the screen if. . . . This was the start of my idea. The finished product is in the program that follows, *Preschool Block Letters*. And the intervening (gory) details are about data compaction.

Most home computers don't have point-addressable graphics, but they do have character graphics that can produce line segments at various angles. Thus, I thought, I would build the letters and numbers from short line segments. Easier said than done. What I really needed to build the letters and numbers was some help. Fortunately, my wife, a teacher, retaught me the correct way to form letters; I, in turn, taught the computer.

Then I had to store about 3500 pieces of information concerned with which line segments go where to form each letter and number. Each piece of information as a number requires 4 to 8 bytes depending on the computer. But *strings* require only one byte for each stored character. And among letters, numbers and punctuation marks, there are enough different characters so that over 40 unique values can be stored using only one byte per value.

Furthermore, strings can be very long, so this helps hold down the overhead to identify each string. Thus, to change the piece of information to the value of a valid ASCII character, I just added a constant to each piece of information. The characters were thus grouped into strings, and the strings stored in DATA statements. The SEG$ and ASC functions retrieve the information as required. And that's how computers came to *draw* rather than *pop* letters.

Note: Make sure ALPHA LOCK is DOWN.

EXPLANATION OF THE PROGRAM
Preschool Block Letters

Lines Nos.	
130-230	Program initialization.
240-330	Scan keyboard looking for a letter or number key.
340-360	Change ASCII code to number between 0 and 35.
360-490	Draw line segments of letter or number in an array.
560-590	Store geometry of "W" in array.
1700-2040	Define line segment characters used to make letters and numbers.
2050-2100	Input word from user.
2110-2320	Have little man hold up letter.
2330-2430	Get key pushed. If it matches letter that man is holding, then draw letter.

```
100 REM *** PRESCHOOL BLOCK LETTERS *
110 REM
120 REM
130 CALL SCREEN(8)
140 OPTION BASE 0
150 DIM AR$(35,2)
160 GOSUB 1690
170 GOSUB 490
180 CALL CLEAR
190 INPUT "DO YOU WANT TO WORK WITH
    WHOLE WORDS RATHER THAN INDIV
    IDUAL LETTERS? (Y/N)":ANS$
200 ANS$=SEG$(ANS$,1,1)
210 IF ANS$="Y" THEN 2040
220 CALL CLEAR
230 PRINT "PRESS A LETTER OR NUMBER KE
    Y"
240 GOSUB 260
250 GOTO 230
260 CALL KEY(0,KEE,STATUS)
270 IF STATUS=0 THEN 260
280 IF KEE<48 THEN 260
290 IF KEE>90 THEN 260
300 IF -((KEE>=58)*(KEE<=64))THEN 260
310 IF KEE<58 THEN 340
320 IF KEE>64 THEN 330
330 KEE=KEE-7
340 S1=KEE-48
350 CALL CLEAR
360 FOR DELAY=1 TO 10
370 NEXT DELAY
```

```
380 FOR I=1 TO LEN(ARS(S1,1))
390 REM
400 COL=ASC(SEGS(ARS(S1,1),I,1))-40
410 ROW=ASC(SEGS(ARS(S1,2),I,1))-42
420 C=ASC(SEGS(ARS(S1,0),I,1))+63
430 CALL SOUND(100,300,2)
440 CALL HCHAR(ROW,COL,C)
450 FOR DELAY=1 TO 10
460 NEXT DELAY
470 NEXT I
480 RETURN
490 REM
500 REM
510 RESTORE 600
520 FOR S1=0 TO 35
530 READ ARS(S1,1),AR$(S1,2),AR$(S1,0)
540 NEXT S1
550 W1$="WXYWXYWXYWXY"
560 W2$="Z"&CHR$(91)&CHR$(92)
570 W3$=W2$&W2$&W2$&W2$
580 AR$(32,0)=W1$&W3$&W1$&W3$
590 RETURN
600 DATA "321000000000123456777777776654
    "
610 DATA "001234567890:;;;;;;:9876543210
    0"
620 DATA "DECJKSSSSIHBFGEDCKJVVVVHIBGF
    "
630 DATA "0000000000000"
640 DATA "0123456789:;"
650 DATA "TTTTTTTTTTTT"
660 DATA "012345678876543210012345678"
670 DATA "210000001234567890:;<<<<<<<<<<"
680 DATA "ICEDRFGBIKCCCCCCCCCRRRRRRRRR"
690 DATA "012345678876556788876543210
    0"
700 DATA "000000000012345567890:;;;;;;;:98
    "
710 DATA "PPPPPPPPPCCCCFGBIUKCDEPGFBHI
    "
720 DATA "000000123456789966666666666666"
730 DATA "012345555555555012345678890:;"
740 DATA "VVVVVVPPPPPPPPVVVVVMVVVVVV"
750 DATA "000001234567776543210123456767
    "
760 DATA "123455555556789:;;;;;;;:00000000
    "
770 DATA "VVVVVEDRFGBIUKCDEPGFBPPPPPPP
    "
780 DATA "443322110012345678876543211"
790 DATA "012345678890:;;;;;;;:98777777
    8"
800 DATA "JKIKJKJKVVBFGPEDCKIGFRRDEC"
810 DATA "012345554433221100"
820 DATA "000000123456789:;"
830 DATA "PPPPPPPJKJKJKJKJKJ"
840 DATA "765432100123456788765432100
    12345677"
850 DATA "100000012345566678890:;;;;;;;:987
    66554321"
860 DATA "BGFRDECJHBFGRFGBIKCDEPGFBHJC
    EDEDCKJ"
870 DATA "765432100123456788888888888"
880 DATA "100000012344444322345678890:;"
890 DATA "BGFRDECJHFGPEDCCSSSSSSSSSSS"
900 DATA "554433221100667788899:;;;;3456
    78"
910 DATA "012345678890:;012345678890:;6666
    66"
920 DATA "IKIKIKIKIKIKIHIHIHIHIHIHIHRRRR
    RR"
930 DATA "000000000000012345678876543211
    2345678876543211"
940 DATA "012345678890:;000000012345555555
    6666678890:;;;;;;;;"
950 DATA "VVVVVVVVVVVVVRRRRFGBIKCDEPPPP
    RRRRFGBIKCDEPPPP"
960 DATA "887654321000000001234567888"
970 DATA "321000000123456789:;;;;;;;:98"
980 DATA "HIBGFRDECJKSSSSIHBFGPEDCKJ"
990 DATA "000000000000001234567788888776
    54321"
1000 DATA "012345678890:;000001234567890:;
     ;;;;;"
1010 DATA "VVVVVVVVVVVVRRRFGBIHITTKJKCD
     EPPP"
1020 DATA "000000000000012345671234512345
     67"
1030 DATA "012345678890:;000000055555;;;;;
     ;;;;"
1040 DATA "VVVVVVVVVVVVRRRRRRRPPPPPPPPP
     PPP"
1050 DATA "000000000000012345671234512345"
1060 DATA "012345678890:;000000055555"
1070 DATA "VVVVVVVVVVVVRRRRRRPPPPPP"
1080 DATA "887654321000000001234567888887"
1090 DATA "321000000123456789:;;;;;;;:9877
     7"
1100 DATA "HIBGFRDECJKSSSSIHBFGPEDCKJVL
     R"
1110 DATA "000000000000007777777777771234
     56"
1120 DATA "012345678890:;012345678890:;5555
     55"
1130 DATA "VVVVVVVVVVVVSSSSSSSSSSSSSPPPP
     PP"
1140 DATA "000000000000001010"
1150 DATA "012345678890:;00:;"
1160 DATA "VVVVVVVVVVVVLRMP"
1170 DATA "666666666665432100567"
1180 DATA "123456789:;<<<:;:9000"
1190 DATA "UUUUUUUUUKCDQFBHTPPP"
1200 DATA "000000000000012345633445566"
1210 DATA "012345678890:;5432104567890:;"
1220 DATA "VVVVVVVVVVVVCCCCCCIHIHIHIH"
1230 DATA "000000000000012345670"
1240 DATA "012345678890:;;;;;;;;;;;"
1250 DATA "VVVVVVVVVVVVPPPPPPP"
1260 DATA "000000000000012348765999999999
     9999"
1270 DATA "012345678890:;012301230123456767
     89:;"
1280 DATA "VVVVVVVVVVVVBBBBCCCCSSSSSSSSS
     SSSS"
1290 DATA "000000000000011223344556667777
     77777"
1300 DATA "012345678890:;012345678890:;0123
     456789:;"
1310 DATA "VVVVVVVVVVVVIHIHIHIHIHIHSSSSS
     SSSSS"
1320 DATA "432100000000012345678888888887
     65"
1330 DATA "000123456789:;;;;;;;:98765432100"
1340 DATA "RDECJKSSSSIHBFGPEDCKJVVVVHIB
     GF"
1350 DATA "000000000000012345677654321"
1360 DATA "012345678890:;000001234555555"
1370 DATA "VVVVVVVVVVVVRRRFGBIKCDEPPP"
1380 DATA "432100000000012345678888888887
     6578"
1390 DATA "000123456789:;;;;;;;:98765432100
     :;;"
1400 DATA "RDECJKSSSSIHBFGPEDCKJVVVVHIB
     GFAB"
1410 DATA "000000000000012345677654321445
     566"
1420 DATA "012345678890:;000001234555555567
     89TA"
1430 DATA "VVVVVVVVVVVVRRRFGBIKCDEPPPIH
     IHIH"
1440 DATA "876543210012345678876543210"
1450 DATA "210000012345556678890:;;;;;;;:9"
1460 DATA "IBGFRDECJHBFGPFGBIKCDEPGFBH"
1470 DATA "444444444444012345678890"
1480 DATA "012345678890:;0000000000"
1490 DATA "VVVVVVVVVVVVRRRRLRRRRR"
1500 DATA "000000000012345678888888888"
```

```
1510 DATA "0123456789:;;;;;;;:98765432109"
1520 DATA "SSSSSSSSIHBFGPEDCKJVVVVVVVV"
1530 DATA "0011223344555;;;;:998877665"
1540 DATA "0123456789:;:0123456789:;:"
1550 DATA "IHIHIHIHIHIHJKJKJKJKJKJK"
1560 DATA "0001111222333777666555544888899
     99:::;;;;???>>>===<<<"
1570 DATA "0123456789:;:0123456789:;:0123
     456789:;:0123456789:;:"
1580 DATA "IHIHIHIHIHIHJKJKJKJKJKJKIHIH
     IHIHIHIHJKJKJKJKJKJK"
1590 DATA "0011223344555554433221100"
1600 DATA "0123456789:;:0123456789:;:"
1610 DATA "IHIHIHIHIHIHJKJKJKJKJK"
1620 DATA "0123498765444444"
1630 DATA "0123401234567899:;:"
1640 DATA "BBBBBCCCCCVVVVVVV"
1650 DATA "0123456789:;:;:98765432100123
     456789:;:"
1660 DATA "000000000000000123456789:;<=====
     ======="
1670 DATA "PPPPPPPPPPPPPCCCCCCCCCCCCCRRRR
     RRRRRRRR"
1680 STOP
1690 RESTORE 1750
1700 FOR C=128 TO 156
1710 READ CD$
1720 CALL CHAR(C,CD$)
1730 NEXT C
1740 RETURN
1750 DATA "8142241818244281"
1760 DATA "8040201008040201"
1770 DATA "0102040810204080"
1780 DATA "030C30C"
1790 DATA "000000000030C30C"
1800 DATA "C0300C03"
1810 DATA "00000000C0300C03"
1820 DATA "0808040402020101"
1830 DATA "8080404020201018"
1840 DATA "0101020204040808"
1850 DATA "1010202040408081"
1860 DATA "FF0101010101010101"
1870 DATA "0101010101010101FF"
1880 DATA "8080808080808080FF"
1890 DATA "FF8080808080808080"
1900 DATA "0000000000000000FF"
1910 DATA "00000000FF"
1920 DATA "FF"
1930 DATA "8080808080808080"
1940 DATA "1010101010101010"
1950 DATA "0808080808080808"
1960 DATA "0101010101010101"
1970 DATA "8080804040404020"
```

```
1980 DATA "2010101010080808004"
1990 DATA "0404020202020101"
2000 DATA "0101010202020204"
2010 DATA "0404080808101010102"
2020 DATA "2020404040808080"
2030 DATA "8292707C10102828282"
2040 REM LETTER REPEATER
2050 PRINT :"IN THIS ACTIVITY A LITTLE
     MAN WILL HOLD UP EACH LETTEROF A
     WORD YOU INPUT."
2060 PRINT "AFTER HE HAS PICKED UP THE
     LETTER, PUSH THAT KEY ON THEKEYBO
     ARD."
2070 PRINT "YOUR DESIRED WORD MAY BE UP
     TO 16 LETTERS LONG.":
2080 INPUT "INPUT DESIRED WORD: ":ANS$
2090 CALL CLEAR
2100 BL=32
2110 RX0=5
2120 RX1=7
2130 RX3=8
2140 RXL=6
2150 RY0=21
2160 MAN=156
2170 RYL=20
2180 FOR RX=1 TO LEN(ANS$)
2190 CALL HCHAR(RY0,8+RX,ASC(SEG$(ANS$,
     RX,1)))
2200 NEXT RX
2210 FOR DELAY=1 TO 100
2220 NEXT DELAY
2230 FOR RX=1 TO LEN(ANS$)
2240 CALL GCHAR(RYL,RXL+RX,RC)
2250 CALL HCHAR(6,5,32,448)
2260 CALL VCHAR(RYL,RXL+RX,BL,2)
2270 CALL HCHAR(RY0,RX0+RX,RC)
2280 CALL HCHAR(RY0,RX1+RX,MAN)
2290 CALL GCHAR(RY0,RX3+RX,RC)
2300 CALL HCHAR(RY0,RX3+RX,BL)
2310 CALL HCHAR(RYL,RX1+RX,RC)
2320 IF RC=32 THEN 2410
2330 CALL KEY(0,KEE,STATUS)
2340 IF KEE=RC THEN 2350 ELSE 2330
2350 IF KEE<58 THEN 2370
2360 KEE=KEE-7
2370 S1=KEE-48
2380 GOSUB 380
2390 FOR DELAY=1 TO 200
2400 NEXT DELAY
2410 NEXT RX
2420 GOTO 2080
2430 END
```

HOMEWORK HELPER

5 FRACTIONS

Homework Helper: Students do their class assignments on paper in the usual way. . . . and then can use the Homework Helper to quickly correct their assignments.

The *Homework Helper* series is designed to quickly give answers to students checking their assignments. It is not meant to be a tutorial; it does not teach concepts nor quiz the student. Rather, it gives the answers to the problems without showing all the intermediate steps.

The students are encouraged to do their class assignments on paper in the usual way, writing the problems down and working the problems step-by-step. Then, they can use the *Homework Helper* to correct their assignments quickly.

Fractions

This program, involving fractions, is for correcting the homework problems of elementary school math students (4th, 5th, and 6th graders). Written in TI BASIC, it employs color graphics and sound, and is interactive. There are seven sections, each introduced with a simple color representation of what that section is doing with fractions. Musical phrases from Mendelssohn, Handel, and Beethoven are played at the same time.

1. **Equivalence**. Two fractions are of the form

$$\frac{A}{B} = \frac{C}{D}$$

Any one of the four positions can be the unknown. The user designates the unknown, and inputs the three given values. The computer finds the unknown and prints the equivalent fractions. A student can also use this section to find equivalent ratios.

2. **Simplification**. The user inputs a numerator and a denominator. The computer simplifies (reduces) the fraction or tells if it cannot be simplified.

3. **Multiplication**. The user designates the number of fractions to be multiplied, then enters the numerator and denominator for each one. The computer multiplies them and simplifies the final fraction.

4. **Division**. Two fraction are entered; the first is then divided by the second, and the answer is simplified.

5. **Addition—Like Denominators.** The user specifies the number of fractions to be added, the common denominator and then enters the numerators. The computer adds the numbers and simplifies the result.

6. **Addition—Unlike Denominators.** This section may be used to add fractions with like or unlike denominators. The user specifies the number of fractions up to five (which should be sufficient for elementary school mathematics), and then inputs the numerator and denominator of each. The computer adds the fractions and simplifies the result. A student can also use either Section 5 or 6 for subtraction problems by entering a negative numerator.

7. **Comparisons.** As many as ten fractions may be compared on a number line. The user enters the number of fractions to be compared (up to ten), and then enters the numerator and denominator of each. The computer then arranges the fractions from the smallest to the largest and prints them.

To stop any section of the program press SHIFT C. To restart, enter RUN.

Simplifying Fractions

One basic technique of simplifying fractions is to start with the numerator as the first factor and see if it can be divided evenly into the denominator. If it can, both numerator and denominator are divided by that factor immediately to yield the simplified fraction. If the denominator cannot be evenly divided, the factor is reduced by one, and the numerator and denominator are tested to see if they are divisible by the new factor.

In each successive test, the factor is reduced by one. When both numerator and denominator can finally be evenly divided by the factor, that factor is the greatest common factor. The numerator and denominator are then divided by this factor to yield the reduced fraction

For larger numbers, the technique can take a lot of time. In this program, the algorithm has been made more efficient by first checking to see which is smaller, the numerator or the denominator. In improper fractions the denominator will be smaller. The starting factor, PLIM, is set equal to the smaller number (Statements 1380 to 1410).

Another efficiency technique is not to test all even factors if either numerator or denominator is an odd number. This technique cuts the search time in half. In Statements 1420 to 1450 the step size, S, is set equal to -2 if either the numerator or the denominator is odd; S is set equal to -1 if both numerator and denominator are even numbers.

The simplifying algorithm is implemented with a FOR-NEXT loop. The starting trial factor is reduced by the step size, S, to a lower limit of 2 in line 1460.

Within the loop, Statements 1460 to 1510 set A = NS/P (where NS is the numerator) and set B = DS/P (where DS is the denominator). Then they check to see if A = INT(A); if equal, then B = INT(B) is checked. If both statements are true, the simplified fraction is A/B. Otherwise, P is incremented by S, and the loop continues. If the lower limit is reached without finding a successful factor, the user is notified that the fraction cannot be simplified (Statements 1520-1540).

When combining several fractions in multiplication or addition, another efficiency technique sets the starting

factor equal to the largest denominator of the orginal fractions (Statements 2250-2340). The common denominator may be much larger than the original denominators, but the largest factor will always be the largest original denominator.

Comparisons

The schoolroom technique for comparing fractions is to find the common denominator and then compare the adjusted numerators. This technique is far too slow for computers, especially when comparing many fractions and/or fractions with large numbers. A very fast technique which achieves the same result is to compute and compare the decimal equivalents of the fractions.

As the fractions are read in, the numerator NNN (I) is divided by the denominator DD (I) and stored as a decimal fraction in two identical arrays, FRC (I) and FRD (I) (Statements 5170-5230). A standard sort routine sorts the first array FRC from the smallest to the largest. The subscripts are changed as the decimal fractions are arranged in order (Statements 5250-5330).

The first element of FRC is compared with each element of the second array, FRD. When a match is made, the subscript value J is used to retrieve the numerator and denominator of the corresponding fraction for printing. The process is repeated in order for each element in the FRC array (Statements 5340-5390).

EXPLANATION OF THE PROGRAM

Homework Helper: Fractions

Line Nos.	
160-170	Sets T and T2 for the time in the music statements.
180-250	Defines characters and colors in four different character sets for use in graphics.
260-390	Prints title screen,"HOMEWORK HELPER".
410-550	Prints "Fractions" and blinks an outline of asterisks around it.
580-640	Prints the menu screen for the seven sections of the program.
670-730	The user presses a key to choose which of the 7 sections is wanted, and the computer branches to that section.
749-890	Prints the screen for Equivalence.
900-960	Asks for the unknown, A,B,C, or D.
970-1190	Depending on which is the unknown, asks for the given values and calculates the unknown. If the unknown is not a whole number, it will be rounded to two decimal places.
1200-1230	Prints the equivalent fractions.
1240-1310	Asks if there is another problem or to stop. If "2" is pressed, the menu screen is returned.
1320-1350	Prints screen for Simplifying.
1360-1370	Asks for the fraction.
1460-1540	Simplifies and prints the result.
1550-1620	Continue, or go to menu screen.
1530-1650	Prints screen for Multiplying.
1660-1710	Asks for the fractions.
1720-1770	Multiplies the fractions.
1780-1800	Prints the problem and the simplified answer.
1810-1880	Continue, or go to menu screen.
1890-1930	Subroutine for printing the problem.
1940-2150	Subroutine for simplifying and printing.
2160-2360	Subroutine for sorting and simplifying. These three subroutines are used for simplifying and printing in other sections of the program also.
2370-2340	Prints screen for Dividing.
2440-2470	Asks for the two fractions.
2480-2490	Performs division.
2500-2540	Prints problem and simplified solution.
2550-2620	Continue, or go to menu screen.
2630-2680	Prints screen for Adding with like denominators.
2690-2760	Asks for fractions and adds the numerators.
2770-2820	Prints the problem and the simplified sum.
2830-2990	Continue, or go to menu screen.
2910-2960	Prints screen for Adding with unlike denominators.
2970-3090	Asks for the fractions and calculates a common denominator.
3100-3150	Adds the adjusted numerators and prints the problem and the simplified result.
3160-3230	Continue, or go to menu screen.
3240-4090	Sound subroutines musical phrases.
4100-5020	Draws color graphics for each title screen.
5030-5120	Prints screen for Comparisons.
5130-5230	Asks for fractions and converts fractions to decimals.
5240-5330	Sorts fractions from the smallest to the largest.
5340-5390	Prints fractions in order.
5400-5470	Continue, or go to menu screen.
5480-5670	Music and graphics for Comparisons.

To stop the program, press SHIFT C (BREAK). For the student's convenience, at the end of each problem he can choose to do another problem of the same type or go to the menu screen and do a problem of a different type.

```
100 REM *********************
110 REM *                    *
120 REM *  HOMEWORK HELPERS:  *
130 REM *      FRACTIONS      *
140 REM *                    *
150 REM *********************
160 REM
170 REM
180 REM
190 T=300
200 T2=150
210 CALL CHAR(128,"F")
220 CALL COLOR(13,7,7)
230 CALL CHAR(136,"F")
240 CALL COLOR(14,11,11)
250 CALL CHAR(144,"F")
260 CALL COLOR(15,5,5)
270 CALL CHAR(152,"F")
280 CALL COLOR(16,13,13)
290 CALL CLEAR
300 CALL COLOR(2,16,7)
310 DATA 72,79,77,69,87,79,82,75
320 RESTORE 310
330 FOR Y=9 TO 23 STEP 2
340 READ L
350 CALL HCHAR(7,Y,L)
360 NEXT Y
370 DATA 72,69,76,80,69,82
380 RESTORE 370
390 FOR Y=11 TO 21 STEP 2
400 READ L
410 CALL HCHAR(10,Y,L)
420 NEXT Y
430 GOSUB 3270
440 CALL HCHAR(14,10,42,13)
450 CALL VCHAR(15,22,42,3)
460 CALL VCHAR(15,10,42,3)
470 CALL HCHAR(18,10,42,13)
480 DATA 70,82,65,67,84,73,79,78,83
490 RESTORE 480
500 FOR Y=12 TO 20
510 READ L
520 CALL HCHAR(16,Y,L)
530 NEXT Y
540 FOR I=1 TO 30
550 CALL COLOR(2,7,16)
560 CALL COLOR(2,16,7)
570 NEXT I
```

```
580 GOSUB 3380
590 CALL COLOR(2,2,1)
600 CALL CLEAR
610 PRINT "1 EQUIVALENCE"
620 PRINT "2 SIMPLIFY"
630 PRINT "3 MULTIPLY"
640 PRINT "4 DIVIDE"
650 PRINT "5 ADD-LIKE DENOMINATORS"
660 PRINT "6 ADD-UNLIKE DENOMINATORS"
670 PRINT "7 COMPARISONS":::::
680 GOSUB 4890
690 GOSUB 3470
700 CALL KEY(0,CHOICE,STATUS)
710 IF STATUS<=0 THEN 700
720 IF CHOICE<49 THEN 700
730 IF CHOICE>55 THEN 700
740 CHOICE=CHOICE-48
750 CALL CLEAR
760 ON CHOICE GOTO 770,1350,1660,2400,
    2660,2940,5060
770 DATA 69,81,85,73,86,65,76,69,78,67
    ,69
780 RESTORE 770
790 CALL CLEAR
800 FOR Y=11 TO 21
810 READ L
820 CALL HCHAR(9,Y,L)
830 NEXT Y
840 CALL CHAR(96,"00000FF")
850 GOSUB 4130
860 DATA 14,13,65,15,13,96,16,13,66,15
    ,16,61,14,19,67,15,19,96,16,19,68
870 RESTORE 860
880 FOR I=1 TO 7
890 READ X,Y,L
900 CALL HCHAR(X,Y,L)
910 NEXT I
920 GOSUB 3550
930 INPUT "WHICH IS THE UNKNOWN?":X$
940 IF X$="A" THEN 1000
950 IF X$="B" THEN 1060
960 IF X$="C" THEN 1120
970 IF X$="D" THEN 1180
980 PRINT "CHOSE A,B,C,OR D"
990 GOTO 930
1000 INPUT "ENTER B=":B
1010 INPUT "ENTER C=":C
1020 INPUT "ENTER D=":D
1030 A=B*C/D+.005
1040 A=1E-2*(INT(A*1E2))
1050 GOTO 1230
1060 INPUT "ENTER A=":A
1070 INPUT "ENTER C=":C
1080 INPUT "ENTER D=":D
1090 B=A*D/C+.005
1100 B=1E-2*(INT(B*1E2))
1110 GOTO 1230
1120 INPUT "ENTER A=":A
1130 INPUT "ENTER B=":B
1140 INPUT "ENTER D=":D
1150 C=A*D/B+.005
1160 C=1E-2*(INT(C*1E2))
1170 GOTO 1230
1180 INPUT "ENTER A=":A
1190 INPUT "ENTER B=":B
1200 INPUT "ENTER C=":C
1210 D=B*C/A+.005
1220 D=1E-2*(INT(D*1E2))
1230 IF A<0 THEN 1250
1240 IF B<0 THEN 1250
1245 GOTO 1260
1250 PRINT "A,B,D:::"
1260 CALL CLEAR
1270 PRINT "PRESS 1 FOR NEXT PROBLEM"
1280 PRINT "PRESS 2 TO STOP"
1290 CALL KEY(0,K,STATUS)
1300 IF STATUS<=0 THEN 1290
1310 IF K<>49 THEN 1330
1320 CALL CLEAR
1325 GOTO 780
1330 IF K=50 THEN 590
1340 GOTO 1290
1350 PRINT "** SIMPLIFYING FRACTIONS **
     "
1360 GOSUB 4230
1370 PRINT :::
1380 GOSUB 3650
1390 INPUT "NUMERATOR?":NS
1400 INPUT "DENOMINATOR?":DS
1410 IF DS>NS THEN 1440
1420 PLIM=DS
1430 GOTO 1450
1440 PLIM=NS
1450 S=-2
1460 IF DS/2=INT(DS/2)THEN 1480
1470 IF NS/2=INT(NS/2)THEN 1490
1480 S=-1
1490 FOR P=PLIM TO 2 STEP S
1500 A=NS/P
1510 IF A<>INT(A)THEN 1540
1520 B=DS/P
1530 IF B=INT(B)THEN 1570
1540 NEXT P
1550 PRINT :NS;"/";DS;" CANNOT BE SIMPL
     IFIED:::::
1560 GOTO 1580
1570 PRINT :NS;"/";DS;" = ";A;"/";B:::
     :
1580 PRINT "PRESS 1 FOR NEXT PROBLEM"
1590 PRINT "PRESS 2 TO STOP"
1600 CALL KEY(0,K,STATUS)
1610 IF STATUS<=0 THEN 1600
1620 IF K<>49 THEN 1640
1630 CALL CLEAR
1635 GOTO 1350
1640 IF K=50 THEN 590
1650 GOTO 1600
1660 PRINT "** MULTIPLYING FRACTIONS **
     ::::::
1670 GOSUB 4430
1680 GOSUB 3750
1690 INPUT "HOW MANY FRACTIONS? ":F
1700 FOR I=1 TO F
1710 PRINT "FRACTION";I
1720 INPUT "     NUMERATOR = ":NN(I)
1730 INPUT "     DENOMINATOR = ":DD(I)
1740 NEXT I
1750 TN=1
1760 TD=1
1770 FOR I=1 TO F
1780 TN=TN*NN(I)
1790 TD=TD*DD(I)
1800 NEXT I
1810 PRINT :::"** MULTIPLY **":::
1820 GOSUB 1920
1830 GOSUB 2190
1840 PRINT :::"PRESS 1 FOR NEXT PROBLEM
     "
1850 PRINT "PRESS 2 TO STOP"
1860 CALL KEY(0,K,STATUS)
1870 IF STATUS<=0 THEN 1860
1880 IF K<>49 THEN 1900
1890 CALL CLEAR
1895 GOTO 1660
1900 IF K=50 THEN 590
1910 GOTO 1360
1920 FOR I=1 TO F
1930 PRINT NN(I);"/";DD(I)
1940 NEXT I
1950 PRINT "IS EQUAL TO"
1960 IF TN<TD THEN 2000
1970 PLIM=TN
1980 GOTO 2010
1990 PLIM=TD
2000 GOTO 2010
2010 FOR P=PLIM TO 2 STEP -1
2020 A=TN/P
2030 IF A<>INT(A)THEN 2060
2040 B=TD/P
```

```
2050 IF B=INT(B)THEN 2090
2060 NEXT P
2070 A=TN
2080 B=TD
2090 IF A>=B THEN 2120
2100 PRINT :: :A;"/";B: : : :
2110 GOTO 2160
2120 IF B=1 THEN 2170
2130 C=INT(A/B)
2140 R=A-C*B
2150 PRINT C;" ";R;"/";B: : : :
2160 RETURN
2170 PRINT A: : : :
2180 RETURN
2190 FOR I=1 TO F
2200 P=DD(I)
2210 A=TN/P
2220 IF A<>INT(A)THEN 2270
2230 B=TD/P
2240 IF B<>INT(B)THEN 2270
2250 TN=A
2260 TD=B
2270 NEXT I
2280 SW=0
2290 FOR I=1 TO F-1
2300 IF DD(I)<=DD(I+1)THEN 2350
2310 J=DD(I)
2320 DD(I)=DD(I+1)
2330 DD(I+1)=J
2340 SW1=1
2350 NEXT I
2360 IF SW=1 THEN 2280
2370 PLIM=DD(F)
2380 GOSUB 2010
2390 RETURN
2400 PRINT "** DIVIDING FRACTIONS **":
     : :
2410 GOSUB 4520
2420 PRINT "THE FIRST FRACTION IS"
2430 PRINT "DIVIDED BY THE "
2440 PRINT "SECOND FRACTION": :
2450 PRINT "(N1/D1) / (N2/D2)": :
2460 GOSUB 3830
2470 INPUT "ENTER N1 = ":N1
2480 INPUT "ENTER D1 = ":D1
2490 INPUT "ENTER N2 = ":N2
2500 INPUT "ENTER D2 = ":D2
2510 TN=N1*D2
2520 TD=D1*N2
2530 PRINT : : :N1;"/";D1
2540 PRINT " ---- ": :
2550 PRINT N2;"/";D2: : :
2560 PRINT "EQUALS": :
2570 GOSUB 1970
2580 PRINT :"PRESS 1 FOR NEXT PROBLEM"
2590 PRINT "PRESS 2 TO STOP"
2600 CALL KEY(0,K,STATUS)
2610 IF STATUS<=0 THEN 2600
2620 IF K<>49 THEN 2640
2630 CALL CLEAR
2635 GOTO 2400
2640 IF K=50 THEN 590
2650 GOTO 2600
2660 PRINT "** ADDING FRACTIONS **": :
2670 GOSUB 4590
2680 PRINT "THIS SECTION ADDS"
2690 PRINT "FRACTIONS THAT ALL HAVE"
2700 PRINT "THE SAME DENOMINATOR.": : :
2710 GOSUB 3910
2720 INPUT "HOW MANY FRACTIONS? ":F
2730 INPUT "WHAT IS THE DENOMINATOR?":T
     D
2740 PRINT "ENTER THE NUMERATORS"
2750 TN=0
2760 FOR I=1 TO F
2770 INPUT NN(I)
2780 TN=TN+NN(I)
2790 NEXT I
2800 FOR I=1 TO F
2810 DD(I)=TD
2820 NEXT I
2830 PRINT : : :
2840 GOSUB 1920
2850 GOSUB 1970
2860 PRINT "PRESS 1 FOR NEXT PROBLEM"
2870 PRINT "PRESS 2 TO STOP"
2880 CALL KEY(0,K,STATUS)
2890 IF STATUS<=0 THEN 2880
2900 IF K<>49 THEN 2920
2910 CALL CLEAR
2915 GOTO 2660
2920 IF K=50 THEN 590
2930 GOTO 2880
2940 PRINT "** ADDING FRACTIONS **"
2950 GOSUB 4740
2960 PRINT : : :"THIS SECTION ADDS"
2970 PRINT "FRACTIONS THAT MAY HAVE"
2980 PRINT "UNLIKE DENOMINATORS": : :
2990 GOSUB 4020
3000 INPUT "HOW MANY FRACTIONS? ":F
3010 IF F<=5 THEN 3040
3020 PRINT "SORRY, I CAN ONLY HANDLE UP
     TO 5"
3030 GOTO 3000
3040 TN=0
3050 TD=1
3060 FOR I=1 TO F
3070 PRINT "FRACTION";I
3080 INPUT "     NUMERATOR = ":NN(I)
3090 INPUT "     DENOMINATOR = ":DD(I)
3100 TD=TD*DD(I)
3110 NEXT I
3120 PRINT : :
3130 FOR I=1 TO F
3140 C=TD/DD(I)
3150 TN=TN+NN(I)*C
3160 NEXT I
3170 GOSUB 1920
3180 GOSUB 2190
3190 PRINT "PRESS 1 FOR NEXT PROBLEM"
3200 PRINT "PRESS 2 TO STOP"
3210 CALL KEY(0,K,STATUS)
3220 IF STATUS<=0 THEN 3210
3230 IF K<>49 THEN 3250
3240 CALL CLEAR
3245 GOTO 2940
3250 IF K=50 THEN 590
3260 GOTO 3210
3270 CALL SOUND(T,880,2,698,5)
3280 CALL SOUND(T,932,2,784,5)
3290 CALL SOUND(T,784,2,659,5)
3300 CALL SOUND(T,880,2,698,5)
3310 CALL SOUND(T,698,2,587,5)
3320 CALL SOUND(T2,784,2)
3330 CALL SOUND(T2,698,2)
3340 CALL SOUND(T2,659,2)
3350 CALL SOUND(T2,784,2)
3360 CALL SOUND(T,698,2,587,5)
3370 RETURN
3380 CALL SOUND(T,440,2)
3390 CALL SOUND(T2,466,2)
3400 CALL SOUND(T2,523,2)
3410 CALL SOUND(T2,587,2)
3420 CALL SOUND(T2,523,2)
3430 CALL SOUND(T,466,2)
3440 CALL SOUND(T,440,2)
3450 CALL SOUND(1000,392,2,330,5)
3460 RETURN
3470 CALL SOUND(T2,440,2)
3480 CALL SOUND(T2,466,2)
3490 CALL SOUND(T,523,2)
3500 CALL SOUND(T,440,2)
3510 CALL SOUND(T,587,2)
3520 CALL SOUND(T,784,2)
3530 CALL SOUND(500,659,2)
3540 RETURN
3550 CALL SOUND(T,698,2)
3560 CALL SOUND(T,932,2)
```

```
3570 CALL SOUND(T,784,2)
3580 CALL SOUND(T,880,2)
3590 CALL SOUND(T2,932,2)
3600 CALL SOUND(T2,880,2)
3610 CALL SOUND(T2,784,2)
3620 CALL SOUND(T2,880,2)
3630 CALL SOUND(500,698,2)
3640 RETURN
3650 CALL SOUND(T2,659,2)
3660 CALL SOUND(T2,587,2)
3670 CALL SOUND(T,523,2)
3680 CALL SOUND(T,440,2)
3690 CALL SOUND(T,698,2,440,5)
3700 CALL SOUND(T,784,2,587,5)
3710 CALL SOUND(T2,698,2,392,5)
3720 CALL SOUND(T2,659,2)
3730 CALL SOUND(1000,698,2,440,5)
3740 RETURN
3750 DATA 262,349,392,440,523,440,392,3
     49,392
3760 RESTORE 3750
3770 FOR I=1 TO 9
3780 READ M
3790 CALL SOUND(T2,M,2)
3800 NEXT I
3810 CALL SOUND(500,440,2)
3820 RETURN
3830 CALL SOUND(600,262,10)
3840 CALL SOUND(600,311,7)
3850 CALL SOUND(450,392,4)
3860 CALL SOUND(150,349,4)
3870 CALL SOUND(300,311,6)
3880 CALL SOUND(300,294,8)
3890 CALL SOUND(500,262,10)
3900 RETURN
3910 CALL SOUND(T,523,2)
3920 CALL SOUND(T,440,2)
3930 CALL SOUND(T2,440,2)
3940 CALL SOUND(T2,494,2)
3950 CALL SOUND(T,523,2)
3960 CALL SOUND(T2,494,2)
3970 CALL SOUND(T2,523,2)
3980 CALL SOUND(T2,494,2)
3990 CALL SOUND(T2,392,2)
4000 CALL SOUND(1000,440,2)
4010 RETURN
4020 CALL SOUND(400,440,8)
4030 CALL SOUND(200,392,8)
4040 CALL SOUND(200,440,7)
4050 CALL SOUND(400,587,6)
4060 CALL SOUND(200,523,5)
4070 CALL SOUND(200,587,4)
4080 CALL SOUND(400,494,3)
4090 CALL SOUND(200,440,4)
4100 CALL SOUND(200,494,5)
4110 CALL SOUND(500,392,6)
4120 RETURN
4130 CALL HCHAR(12,4,128,3)
4140 CALL HCHAR(13,4,128,3)
4150 FOR Y=4 TO 6
4160 CALL VCHAR(14,Y,144,4)
4170 NEXT Y
4180 CALL VCHAR(11,27,136,3)
4190 CALL VCHAR(11,28,136,3)
4200 CALL VCHAR(14,27,152,6)
4210 CALL VCHAR(14,28,152,6)
4220 RETURN
4230 FOR X=10 TO 14 STEP 2
4240 FOR Y=8 TO 12 STEP 2
4250 CALL HCHAR(X,Y,144)
4260 CALL HCHAR(X,Y+1,136)
4270 NEXT Y
4280 NEXT X
4290 FOR X=11 TO 13 STEP 2
4300 FOR Y=8 TO 12 STEP 2
4310 CALL HCHAR(X,Y,144)
4320 CALL HCHAR(X,Y+1,136)
4330 NEXT Y
4340 NEXT X
4350 CALL HCHAR(12,16,61)
4360 FOR Y=19 TO 21
4370 CALL VCHAR(10,Y,136,5)
4380 NEXT Y
4390 FOR Y=22 TO 24
4400 CALL VCHAR(10,Y,144,5)
4410 NEXT Y
4420 RETURN
4430 Y=6
4440 FOR I=1 TO 5
4450 CALL VCHAR(10,Y,136,2)
4460 CALL VCHAR(10,Y+1,136,2)
4470 CALL VCHAR(12,Y,128,4)
4480 CALL VCHAR(12,Y+1,128,4)
4490 Y=Y+5
4500 NEXT I
4510 RETURN
4520 CALL HCHAR(10,11,136,13)
4530 CALL HCHAR(11,11,136,13)
4540 FOR X=12 TO 14
4550 CALL HCHAR(X,11,144,13)
4560 NEXT X
4570 CALL VCHAR(7,17,93,11)
4580 RETURN
4590 CALL HCHAR(10,8,128,2)
4600 CALL VCHAR(11,8,152,4)
4610 CALL VCHAR(11,9,152,4)
4620 CALL VCHAR(10,13,128,4)
4630 CALL VCHAR(10,14,128,4)
4640 CALL HCHAR(14,13,152,2)
4650 CALL VCHAR(10,18,128,2)
4660 CALL VCHAR(10,19,128,2)
4670 CALL VCHAR(12,18,152,3)
4680 CALL VCHAR(12,19,152,3)
4690 CALL VCHAR(10,23,128,3)
4700 CALL VCHAR(10,24,128,3)
4710 CALL VCHAR(13,23,152,2)
4720 CALL VCHAR(13,24,152,2)
4730 RETURN
4740 CALL VCHAR(10,8,128,4)
4750 CALL VCHAR(14,8,136,3)
4760 CALL VCHAR(10,12,144,2)
4770 CALL VCHAR(10,13,144,2)
4780 CALL VCHAR(12,12,128,3)
4790 CALL VCHAR(12,13,128,3)
4800 CALL HCHAR(10,17,136,3)
4810 CALL HCHAR(11,17,152,3)
4820 CALL HCHAR(12,17,152,3)
4830 CALL HCHAR(13,17,152,3)
4840 CALL VCHAR(10,23,152,2)
4850 CALL VCHAR(10,24,152,2)
4860 CALL VCHAR(12,23,144,4)
4870 CALL VCHAR(12,24,144,4)
4880 RETURN
4890 CALL HCHAR(4,15,128,2)
4900 CALL HCHAR(5,14,128,3)
4910 CALL HCHAR(6,13,128,4)
4920 CALL HCHAR(7,13,128,4)
4930 CALL HCHAR(4,17,136,2)
4940 CALL HCHAR(5,17,136,3)
4950 CALL HCHAR(6,17,136,4)
4960 CALL HCHAR(7,17,136,4)
4970 CALL HCHAR(8,17,152,4)
4980 CALL HCHAR(9,17,152,4)
4990 CALL HCHAR(10,17,152,3)
5000 CALL HCHAR(11,17,152,2)
5010 CALL HCHAR(11,15,144,2)
5020 CALL HCHAR(10,14,144,3)
5030 CALL HCHAR(9,13,144,4)
5040 CALL HCHAR(8,13,144,4)
5050 RETURN
5060 DATA 67,79,77,80,65,82,73,83,79,78
     ,83,32
5070 RESTORE 5060
5080 FOR Y=11 TO 22
5090 READ L
5100 CALL HCHAR(14,Y,L)
5110 NEXT Y
5120 GOSUB 5510
```

```
5130 DIM NNN(10),DDD(10),FRC(10),FRD(10
     )
5140 PRINT "THIS ARRANGES FRACTIONS"
5150 PRINT "FROM SMALLEST TO LARGEST.":
     :
5160 INPUT "HOW MANY FRACTIONS?":NF
5170 IF NF<11 THEN 5200
5180 PRINT "SORRY; UP TO 10 ONLY."
5190 GOTO 5160
5200 FOR I=1 TO NF
5210 PRINT :: "FRACTION ";I
5220 INPUT "     NUMERATOR:":NNN(I)
5230 INPUT "     DENOMINATOR:":DDD(I)
5240 FRC(I)=NNN(I)/DDD(I)
5250 FRD(I)=FRC(I)
5260 NEXT I
5270 PRINT :: :: ::
5280 SW=0
5290 FOR I=1 TO NF-1
5300 IF FRC(I)<=FRC(I+1)THEN 5350
5310 FF=FRC(I)
5320 FRC(I)=FRC(I+1)
5330 FRC(I+1)=FF
5340 SW=1
5350 NEXT I
5360 IF SW=1 THEN 5280
5370 FOR I=1 TO NF
5380 FOR J=1 TO NF
5390 IF FRC(I)=FRD(J)THEN 5410
5400 NEXT J
5410 PRINT I;"    ";NNN(J);"/";DDD(J)
5420 NEXT I
5430 PRINT :: :: :"PRESS 1 FOR NEXT PROBL
     EM"
5440 PRINT "PRESS 2 TO STOP"
5450 CALL KEY(0,K,STATUS)
5460 IF STATUS<=0 THEN 5450
5470 IF K<>49 THEN 5490
5480 CALL CLEAR
5485 GOTO 5070
5490 IF K=50 THEN 590
5500 GOTO 5450
5510 CALL SOUND(400,330,2,262,5)
5520 CALL VCHAR(4,8,136,3)
5530 CALL VCHAR(4,9,136,3)
5540 CALL SOUND(100,330,2)
5550 CALL HCHAR(7,8,144,2)
5560 CALL SOUND(100,262,3)
5570 CALL SOUND(400,330,1)
5580 CALL VCHAR(4,15,128,4)
5590 CALL VCHAR(8,15,152,2)
5600 CALL SOUND(100,330,2)
5610 CALL SOUND(100,262,3)
5620 CALL SOUND(400,330,1)
5630 CALL VCHAR(4,23,152,3)
5640 CALL VCHAR(4,24,152,3)
5650 CALL SOUND(200,392,5)
5660 CALL VCHAR(7,23,136,3)
5670 CALL SOUND(200,524,3)
5680 CALL VCHAR(7,24,136,3)
5690 CALL SOUND(400,660,1)
5700 RETURN
```

HOMEWORK HELPER ..... DIVISION

Homework Helper: Students do their class assignments on paper in the usual way. . . . and then use the Homework Helper to quickly correct their assignments.

Division gives the answers to three types of homework problems an elementary school student may encounter: division with a remainder, division with a decimal in the quotient, and division to convert a fraction to a decimal.

Only the answers are given, not the step-by-step process of long division. The student is encouraged to do the homework—writing each step in the division process and then using this program to check the answers. Music and graphics enhance the interaction.

1. Division with Remainder. Most math problems can simply be corrected with a calculator. However if there is a remainder, a calculator converts it to a decimal equivalent. This program keeps the answer in quotient-plus-remainder form. The student enters the divisor and dividend; the quotient and remainder are printed.

2. Division with Decimal. Usually after students master the idea of a remainder, they are taught how to place a decimal and keep dividing. In this section, a student enters the divisor and dividend; the quotient with a decimal fraction is printed.

3. Convert Fraction to Decimal. A fraction is converted to a decimal by dividing the numerator by the denominator. The student enters the numerator then the denominator; the equivalent decimal fraction is returned.

After each problem, a student may enter another problem of the same type. If there are no more problems of the same kind or the student wishes to stop, he enters zero and the menu screen will return.

EXPLANATION OF THE PROGRAM
Homework Helper: Division

Line Nos.	
130-770	Prints title screen and blinks color while special graphics characters are defined.
780-1680	Plays music; prints menu screen and branches appropriately for student's response.
1690-1810	Subroutine to print labels of division problem.
1820-1940	Routine for division with remainder.
1950-2050	Routine for division with decimal.
2060-2320	Routine for converting fraction to decimal.

```
100 REM  *  HOMEWORK HELPER  *
110 REM  *     DIVISION      *
120 REM
130 T=300
140 CALL CLEAR
150 CALL CHAR(96,"0000078444444444478")
160 CALL CHAR(97,"0000038101010101038")
170 CALL CHAR(98,"0000044444444428100")
180 PRINT TAB(7);"HOMEWORK"
190 CALL CHAR(99,"0000038443008443800")
200 PRINT :TAB(9);"HELPER"
210 CALL CHAR(100,"000007C444444444C")
220 CALL CHAR(101,"0000078444447844844")
```

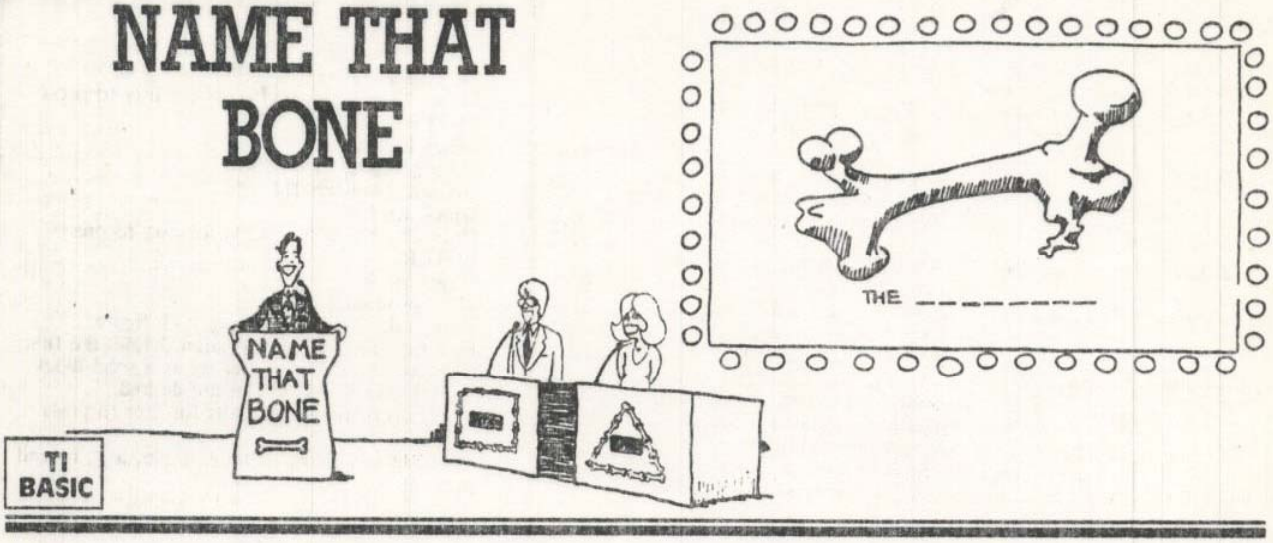
```
 230 PRINT "::::::::::TAB(7);"DIVISO
     N":::::::::"
 240 CALL CHAR(102,"3F1010080808101
     02")
 250 CALL CHAR(103,"FF0078444444447
     8")
 260 CALL COLOR(2,7,16)
 270 CALL HCHAR(14,7,42,19)
 280 CALL VCHAR(15,7,42,3)
 290 CALL VCHAR(15,25,42,3)
 300 CALL HCHAR(18,7,42,19)
 310 CALL COLOR(2,16,7)
 320 CALL CHAR(104,"FF0038101010101
     038")
 330 CALL CHAR(105,"FF0044444444428
     1")
 340 CALL COLOR(2,7,16)
 350 CALL CHAR(106,"FF007C407840407
     C")
 360 CALL CHAR(107,"FF0044645454C44
     ")
 370 CALL COLOR(2,16,7)
 380 CALL CHAR(108,"0172FCFEFEFE7E3
     C")
 390 CALL CHAR(112,"38444454C3C")
 400 CALL COLOR(2,7,16)
 410 CALL CHAR(113,"4444444444438")
 420 CALL CHAR(114,"7C444444447C")
 430 CALL COLOR(2,16,7)
 440 CALL CHAR(115,"7C10101010101")
 450 CALL CHAR(116,"3810101010038")
 460 CALL COLOR(2,7,16)
 470 CALL CHAR(117,"7C407840407C")
 480 CALL CHAR(118,"4464545C44C44")
 490 CALL COLOR(2,16,7)
 500 CALL CHAR(119,"007844447848444")
 510 CALL CHAR(109,"000000C1C3870E")
 520 CALL COLOR(2,7,16)
 530 CALL CHAR(110,"0103070E1C387")
 540 CALL CHAR(111,"80C0E070381C0E07")
 550 CALL COLOR(2,16,7)
 560 CALL CHAR(136,"80C0E0F0F8FCFEFF")
 570 CALL CHAR(137,"7F3F1F0F07030301")
 580 CALL COLOR(2,7,16)
 590 CALL CHAR(121,"000000000003070F0F")
 600 CALL COLOR(2,16,7)
 610 CALL CHAR(122,"030F3FFFFFFFFFFF")
 620 CALL CHAR(123,"C0F0FCFFFFFFFFFF")
 630 CALL COLOR(2,7,16)
 640 CALL CHAR(124,"000000000C0E0F0F")
 650 CALL CHAR(125,"1F1F3F3F7F7FFFFF")
 660 CALL COLOR(2,16,7)
 670 CALL CHAR(126,"F8F8FCFCFEFEFFFF")
 680 CALL CHAR(127,"FFFF7F7F3F3F1F1F")
 690 CALL COLOR(2,7,16)
 700 CALL CHAR(128,"FFFFFEFEFCFCF8F8")
 710 CALL CHAR(129,"0F0F07030")
 720 CALL COLOR(2,16,7)
 730 CALL CHAR(130,"FFFFFFFFFF3F0F03")
 740 CALL CHAR(131,"FFFFFFFFFFFCF0C")
 750 CALL COLOR(2,7,16)
 760 CALL CHAR(132,"F0F0E0C")
 770 CALL COLOR(2,2,1)
 780 CALL CLEAR
 790 CALL CHAR(120,"FFFFFFFFFFFFFFFF")
 800 CALL COLOR(12,1,1)
 810 CALL COLOR(13,1,1)
 820 CALL SOUND(T/2,262,5)
 830 CALL VCHAR(15,6,120,5)
 840 CALL SOUND(T/2,294,5)
 850 CALL HCHAR(15,4,121)
 860 CALL SOUND(T,330,4,131,10)
 870 CALL HCHAR(15,5,122)
 880 CALL HCHAR(15,7,123)
 890 CALL HCHAR(15,8,124)
 900 CALL SOUND(T,330,4,196,10)
 910 CALL HCHAR(16,4,125)
 920 CALL HCHAR(16,5,120,3)
 930 CALL HCHAR(16,8,126)
 940 CALL SOUND(T/2,330,4)
 950 CALL HCHAR(17,4,120,5)
 960 CALL SOUND(T/2,392,4)
 970 CALL HCHAR(18,4,127)
 980 CALL SOUND(T,349,3,147,9)
 990 CALL HCHAR(18,5,120,3)
1000 CALL HCHAR(18,8,128)
```

```
1010 CALL HCHAR(19,4,129)
1020 CALL HCHAR(19,5,130)
1030 CALL SOUND(T,349,3,196,9)
1040 CALL HCHAR(19,7,131)
1050 CALL HCHAR(19,8,132)
1060 CALL COLOR(12,7,1)
1070 CALL COLOR(13,7,1)
1080 CALL SOUND(T/2,349,3)
1090 CALL COLOR(14,15,1)
1100 CALL COLOR(9,5,1)
1110 CALL SOUND(T/2,330,3)
1120 CALL HCHAR(13,9,136)
1130 CALL HCHAR(14,9,137)
1140 CALL SOUND(T,294,3,175,9)
1150 CALL HCHAR(14,10,136)
1160 CALL HCHAR(15,10,137)
1170 CALL HCHAR(15,11,136)
1180 CALL HCHAR(16,11,137)
1190 CALL SOUND(T/2,294,3,196,9)
1200 CALL HCHAR(16,12,136)
1210 CALL HCHAR(17,12,137)
1220 CALL SOUND(T/2,330,3)
1230 CALL HCHAR(17,13,136)
1240 CALL SOUND(T/2,349,2)
1250 CALL HCHAR(18,13,137)
1260 CALL SOUND(T/2,370,2)
1270 CALL HCHAR(18,14,136)
1280 CALL SOUND(T,392,1,165,7)
1290 CALL COLOR(10,7,1)
1300 CALL HCHAR(18,15,109)
1310 CALL HCHAR(19,14,110)
1320 CALL HCHAR(19,15,111)
1330 CALL SOUND(T,392,1,196,7)
1340 CALL HCHAR(20,16,111)
1350 CALL COLOR(11,5,1)
1360 CALL SOUND(T/2,440,2)
1370 CALL SOUND(T/2,494,2)
1380 CALL SOUND(T,523,1,165,7)
1390 CALL SOUND(T,523,1,196,7)
1400 CALL SOUND(T/2,587,2)
1410 CALL CHAR(120,"7FBFDFEFF7FBFDFE")
1420 CALL SOUND(T/2,523,2)
1430 CALL SOUND(T,494,1,175,7)
1440 CALL SOUND(T,784,1,196,7)
1450 CALL HCHAR(17,18,61)
1460 CALL SOUND(T/2,587,1)
1470 CALL SOUND(T/2,523,1)
1480 CALL SOUND(T,494,0,175,6)
1490 CALL HCHAR(13,22,108,7)
1500 CALL SOUND(T,880,0,196,6)
1510 CALL HCHAR(15,22,108,7)
1520 CALL HCHAR(17,22,108,7)
1530 CALL SOUND(T,494,0)
1540 CALL HCHAR(19,22,108,7)
1550 CALL HCHAR(21,22,108,4)
1560 CALL SOUND(T*2,523,1,165,7,131,9)
1570 CALL HCHAR(24,1,67)
1580 CALL HCHAR(24,2,72)
1590 PRINT "OOSE:"
1600 PRINT "1 DIVISION WITH REMAINDER"
1610 PRINT "2 DIVISION WITH DECIMAL"
1620 PRINT "3 CONVERT FRACTION TO DECI
     MA"
1630 CALL HCHAR(23,31,76)
1640 PRINT "4 END PROGRAM"
1650 CALL KEY(0,K,S)
1660 IF (K<49)+(K>52)THEN 1650
1670 CALL CLEAR
1680 ON K-48 GOTO 1820,1950,2060,2320
1690 DATA 96,97,98,97,99,100,101,102,10
     3,104,105,104,103,106,107,103,32
1700 DATA 112,113,114,115,116,117,118,1
     15,32
1710 CALL COLOR(10,5,1)
1720 RESTORE 1690
1730 FOR I=7 TO 23
1740 READ G
1750 CALL HCHAR(19,I,G)
1760 NEXT I
```

```
1770 FOR I=15 TO 23
1780 READ G
1790 CALL HCHAR(18,I,G)
1800 NEXT I
1810 RETURN
1820 PRINT "DIVISION WITH REMAINDER":::
     :::::::
1830 GOSUB 1710
1840 CALL HCHAR(18,25,119)
1850 CALL HCHAR(18,26,35,2)
1860 INPUT "DIVISOR:  ":D
1870 IF D=0 THEN 780
1880 INPUT "DIVIDEND: ":N
1890 C=INT(N/D)
1900 R=N-C*D
1910 PRINT :"QUOTIENT =";C;" R";R
1920 PRINT :::"ENTER NEXT PROBLEM"
1930 PRINT "OR '0' TO STOP.":::::::::::::
1940 GOTO 1830
1950 PRINT "DIVISION WITH DECIMAL":::::
     :::::
1960 GOSUB 1710
1970 CALL HCHAR(18,23,46)
1980 CALL HCHAR(18,24,35,3)
1990 INPUT "DIVISOR:  ":D
2000 IF D=0 THEN 780
2010 INPUT "DIVIDEND: ":N
2020 PRINT :"QUOTIENT =";N/D
2030 PRINT :::"ENTER NEXT PROBLEM"
2040 PRINT "OR '0' TO STOP.":::::::::::::
```

```
2050 GOTO 1960
2060 PRINT "CONVERT FRACTION TO DECIMAL
     ":::::::::::::
2070 CALL CHAR(120,"FFFFFFFFFFFFFFFF")
2080 CALL CHAR(138,"0000000FFFF")
2090 CALL HCHAR(16,8,120,4)
2100 CALL HCHAR(17,8,120,4)
2110 CALL HCHAR(18,7,138,6)
2120 CALL HCHAR(19,8,120,4)
2130 CALL HCHAR(20,8,120,4)
2140 CALL HCHAR(21,8,120,4)
2150 CALL HCHAR(18,15,61)
2160 CALL HCHAR(18,18,46)
2170 CALL COLOR(10,13,13)
2180 CALL HCHAR(18,20,108)
2190 CALL HCHAR(18,22,108)
2200 CALL HCHAR(18,24,108)
2210 PRINT :::
2220 INPUT "NUMERATOR:  ":N
2230 IF N=0 THEN 780
2240 INPUT "DENOMINATOR:  ":D
2250 IF D<>0 THEN 2280
2260 PRINT "SORRY, CANNOT DIVIDE BY ZER
     O":::
2270 GOTO 2240
2280 PRINT :N;"/";D;"=";N/D
2290 PRINT :::"ENTER NEXT PROBLEM"
2300 PRINT "OR ENTER '0' TO STOP."
2310 GOTO 2210
2320 END
```

NAME THAT BONE



Time to review Ezekiel's "Dry Bones" song: "Leg bone connected to the hip bone. . ." Or was it the ankle bone? Or what bone is where?? This program is designed to teach the names of the major bones of the human body and where they are located, and then turn what could be a dry, repetitious drill into an enjoyable game of *Name That Bone*.

The menu screen of the program gives the choice of major parts of the body, head, arms, torso, and legs, or end the program. Each section will label the main bones of the part of the body chosen:

1. HEAD: frontal, parietal, zygomatic, temporal, maxilla, mandible.
2. ARMS: humerus, ulna, radius, carpus, metacarpus, phalanges.
3. TORSO: spine, ribs, clavicle, scapula, sternum, ilium, ischium, sacrum, coccyx.
4. LEGS: femur, tibia, fibula, patella, tarsus, metatarsus, phalanges.

You may study the labeled diagram of the bones as long as you wish, then press ENTER. The labels will be erased and it will be your turn to *Name That Bone*. The bones are listed in a random order at the left of the screen for your choice of answers. A bone will be chosen randomly and will blink red and white until you press a number corresponding to the name of the bone. If you are correct, an arpeggio is played; if you are incorrect, a noise is sounded. You must press the correct answer to continue, and it won't take long for you to learn the names of your bones.

After each bone is chosen once, you will be asked TRY AGAIN? (Y/N). If the response is N, the program returns to the menu screen. If the response is Y, the names of the bones will be rearranged and the bones will be chosen in a different order.

Programming Techniques

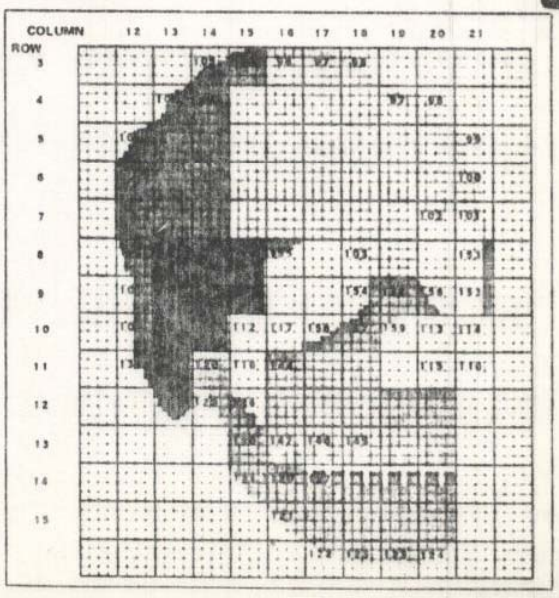
There are four main parts of the body from which to choose, and each part uses the same program logic, so subroutines are used. The subroutines are located at the beginning program. For some microcomputers, execution is faster for subroutines called closer to the beginning; however, the speed in TI BASIC does not seem to depend upon the location of the subroutine.

For each part of the body, different characters are defined. The appropriate DATA statement is RESTOREd, then the subroutine to define characters (lines 160-210) is called. After the labels for the bones are printed, the bones are drawn, again RESTOREing the corresponding DATA statement and calling a subroutine (lines 320-360).

The main procedure is in Lines 370-980. The program will read from DATA the names of the bones and the character set number, then randomly print the bones and choose the bones for the quiz.

The graphics characters were designed so that a specific bone could be blinked by using CALL COLOR statements. The characters of one bone must be in one character set, and another bone in another character set. When the main part of the body is first drawn, all the characters are yellow, but as the bone is chosen, the characters in that set will blink. An example is shown with the skull bones.

(NOTE: The wrist and hand bones are known either as the *carpus* and *metacarpus* or *carpals* and *metacarpals*. The carpals are the elements of the carpus (wrist bone). You may wish to relabel these parts to be consistent with the way you teach them.)



EXPLANATION OF THE PROGRAM

Name That Bone

Line Nos.	
150	Branches to title screen.
160-210	Subroutine reads C and C$ from DATA to define graphics characters.
220-310	Subroutine prints PRESS ENTER and waits for the user to respond.
320-360	Subroutine reads DATA to draw graphics.
370-980	Subroutine for main program logic.
370-390	For the number of bones R, reads the name of the bone and the corresponding character set number.
400-520	Randomly prints the names of the bones for the multiple-choice answers and arranges the corresponding character set number and answer number.
530-580	Prints NAME THAT BONE at the top of the screen.
590-660	Randomly chooses a bone and blinks it red and white while waiting for the user to press the answer.
670-780	If the answer is correct, plays an arpeggio and goes to the next bone; if the answer is incorrect, sounds a noise and awaits another key press.
790-980	Prints TRY AGAIN? (Y/N) and branches appropriately after Y or N is pressed.
990-1100	Defines graphic characters for head.
1110-1120	Labels head bones.
1130-1260	Draws skull and waits for user to press ENTER.
1270-1300	Clears labels.
1310-1350	Main procedure for head.
1360-1440	Defines character for arm.
1450-1510	Labels arm bones.
1520-1590	Draws arm bones and waits for user to press enter.
1600-1630	Clears labels.
1640-1680	Main procedure for arm.
1690-1830	Defines characters and colors for torso.
1840-1860	Labels torso bones.
1870-2030	Draws torso bones and waits for user to press ENTER.
2040-2090	Clears labels.
2100-2170	Main procedure for torso.
2180-2250	Defines characters for leg.
2260-2270	Labels leg bones.
2280-2380	Draws leg bones and waits for user to press ENTER.
2390-2420	Clears labels.
2430-2470	Main prodecure for leg.
2500-2610	Prints title screen and draws stick figure.
2620-2710	First time through the program defines the first character in each character set as a solid block. It then asks if instructions are desired.
2720-2800	Prints instructions and waits for user to press ENTER.
2810-2900	Prints choices of head, arms, torso, legs, or end program.
2910-3000	Waits for user's choice and branches appropriately.

```
100 REM ********************
110 REM *  NAME THAT BONE  *
120 REM ********************
130 REM
140 REM
150 GOTO 2490
160 FOR I=1 TO N
170 READ C,C$
180 CALL CHAR(C,C$)
190 NEXT I
200 CALL CLEAR
210 RETURN
220 DATA 80,82,69,83,83,32,60,69,78,84
    ,69,82,62,32
230 RESTORE 220
240 FOR Y=19 TO 32
250 READ G
260 CALL HCHAR(24,Y,G)
270 NEXT Y
280 CALL KEY(0,K,S)
290 IF K<>13 THEN 280
300 CALL HCHAR(24,19,32,13)
310 RETURN
320 FOR I=1 TO N
330 READ X,Y,G,R
340 CALL HCHAR(X,Y,G,R)
350 NEXT I
360 RETURN
370 FOR I=1 TO R
380 READ BONE$(I),B(I)
390 NEXT I
400 RANDOMIZE
410 FOR I=1 TO R
420 RR=INT(RND*R+1)
430 IF BONE$(RR)="" THEN 420
440 B$(RR)=BONE$(RR)
450 BB(RR)=B(RR)
460 ANS(RR)=I
470 CALL HCHAR(15+I,2,48+I)
480 FOR J=1 TO LEN(B$(RR))
490 CALL HCHAR(15+I,J+3,ASC(SEG$(B$(RR
    ),J,1)))
500 NEXT J
510 BONE$(RR)=""
520 NEXT I
530 DATA 78,65,77,69,32,84,72,65,84,32
    ,66,79,78,69,32
540 RESTORE 530
550 FOR Y=9 TO 23
560 READ G
570 CALL HCHAR(1,Y,G)
580 NEXT Y
590 FOR I=1 TO R
600 RR=INT(RND*R+1)
610 IF B$(RR)="" THEN 600
620 CALL HCHAR(14,2,63,3)
630 CALL KEY(0,K,S)
640 CALL COLOR(BB(RR),16,1)
650 CALL COLOR(BB(RR),7,1)
660 IF S<1 THEN 630
670 IF K-48=ANS(RR)THEN 700
680 CALL SOUND(500,-5,1)
690 GOTO 630
700 CALL HCHAR(14,2,32,3)
710 CALL SOUND(150,262,1)
720 CALL SOUND(150,330,1)
730 CALL SOUND(150,392,1)
740 CALL SOUND(150,330,1)
750 CALL SOUND(200,262,1)
760 CALL COLOR(BB(RR),12,1)
770 B$(RR)=""
780 NEXT I
790 DATA 84,82,89,32,65,71,65,73,78,63
    ,32
800 RESTORE 790
810 FOR Y=22 TO 32
820 READ G
830 CALL HCHAR(23,Y,G)
840 NEXT Y
850 CALL HCHAR(24,26,40)
860 CALL HCHAR(24,27,89)
870 CALL HCHAR(24,28,47)
880 CALL HCHAR(24,29,78)
890 CALL HCHAR(24,30,41)
900 CALL KEY(0,K,S)
910 IF K=78 THEN 2490
920 IF K<>89 THEN 900
930 FOR Y=16 TO 24
940 CALL HCHAR(Y,2,32,12)
950 NEXT Y
```

```
 960 CALL HCHAR(23,22,32,10)
 970 CALL HCHAR(24,26,32,5)
 980 RETURN
 990 RESTORE 1000
1000 DATA 97,F0FCFFFFFFFFFFFF,98,0000C0
     E0F8FCFEFF,99,000C0C0E0E0E0E,100,
     E0E0F0F0F8F8FCFC
1010 DATA 101,FCFCFEFE3F1F0F07,102,FFFF
     FFFFFEF8F00C,103,FFFF7F1F1F0F0707,1
     09,0F7FFFFFFFFFFFFF
1020 DATA 105,0000000031F3F7FFF,106,0107
     0F0F1F1F3F7F,107,7F7F3F3F3F3F3F1F,
     108,0F0F0F0F0F0F0F0F
1030 DATA 137,0F0F0F0F0303030101,138,FFFF
     7F7F3F1E0C,155,FFFEFCF,153,0303030
     3030303030303,159,F0C0808
1040 DATA 154,0303030303030307F0F,156,0080
     C0C0E0E0F0F8,157,0F3FFFFFFFFFFFFE,
     158,0000000010303FFFFF
1050 DATA 113,7C7E7F7F3F3F1F1F,114,0F0F
     FFFFFFFFFFFF,115,1F1F0F0F0F0F1F1F,
     116,FFFFFFFFFEE0C0C
1060 DATA 117,0000000080E0FCFFFF,118,FFFF
     FFFF3F1F07,121,FF7F3F1F1F0F07071,
     122,7F1F0701
1070 DATA 123,FFFFFFFF0F,124,FEFCF0C08,
     125,CFCF8787030301101,126,0080E0F0F
     0F8F8F8
1080 DATA 127,00BBBBBBFFFFFFFF,145,FFFF
     FFFF7777777,146,FFFF7B7B7B7B,147,
     FFFFFFFFFFDB1BC
1090 N=37
1100 GOSUB 160
1110 PRINT " PARIETAL": :TAB(20);"FRONT
     AL": : :" TEMPORAL"
1120 PRINT TAB(20);"ZYGOMATIC": : : :TAB(
     19);"MAXILLA": : : : MANDIBLE"
     : : : : : : : :
1130 RESTORE 1140
1140 DATA 3,14,105,1,3,15,109,1,3,16,96
     ,1,3,17,97,1,3,18,98,1,4,13,106,1,
     4,14,104,1
1150 DATA 4,15,96,4,4,19,97,1,4,20,98,1
     ,5,12,106,1,5,13,104,2,5,15,96,6,5
     ,21,99,1
1160 DATA 6,12,104,3,6,15,96,6,6,21,100
     ,1,7,12,104,3,7,15,96,5,7,20,102,1
     ,7,21,101,1
1170 DATA 8,12,107,1,8,13,136,3,8,16,15
     5,1,8,18,103,1,8,19,96,1,8,21,153,
     1,9,12,108,1
1180 DATA 9,13,136,3,9,18,154,1,9,19,15
     2,1,9,20,156,1,9,21,153,1,10,12,10
     8,1,10,13,136,2
1190 DATA 10,15,112,1,10,16,117,1,10,17
     ,158,1,10,18,157,1,10,19,159,1,10,
     20,113,1,10,21,114,1
1200 DATA 11,12,137,1,11,13,136,1,11,14
     ,120,1,11,15,118,1,11,16,144,3,11,
     20,115,1,11,21,116,1
1210 DATA 12,13,138,1,12,14,125,1,12,15
     ,126,1,12,16,114,5,13,15,120,1,13,
     16,147,1,13,17,146,1
1220 DATA 13,18,145,3,14,15,121,1,14,16
     ,120,1,14,17,127,4,15,16,121,1,15,
     17,120,4
1230 DATA 16,17,122,1,16,18,123,2,16,20
     ,124,1,16,21,32,1
1240 R=66
1250 GOSUB 320
1260 GOSUB 230
1270 RESTORE 1280
1280 DATA 4,3,32,8,6,22,32,7,9,4,32,8,1
     0,22,32,9,13,21,32,7,16,9,32,8
1290 N=6
1300 GOSUB 320
1310 R=6
1320 DATA FRONTAL,9,PARIETAL,10,ZYGOMAT
     IC,11,MANDIBLE,12,TEMPORAL,14,MAXI
     LLA,15
1330 RESTORE 1320
1340 GOSUB 370
1350 GOTO 1330
1360 RESTORE 1370
1370 DATA 97,0001030307070707,98,0F0F0F
     0F0F0F0F0F,99,07070707070303,100,8
     1C3C7EFFFFFFFFF
1380 DATA 101,F7F3F0F0F0F3F7FF,102,FFFF
     FFFF7F7E3C1,113,000000000000FFFFFF,
     121,7E7E7E7E007E7E7E
1390 DATA 129,FEFEFEFEFE000000,130,FEFE
     FEFEFE0000FE,131,FEFEFE0000FEFEFE,
     137,FFFFFFFE00008C0C
1400 DATA 138,FFFFFFFFFF0000FF,139,80C0
     C0C0800000FE,140,FFFFFFFF0000FFFFF,
     141,FFFFFF0000FFFFFF
1410 DATA 142,FFFFFF0000C0E0E0,143,C0C0
     8,145,000001070F1F7FFF,146,1878F8F
     8F0E0C08
1420 DATA 64,00000001F1010101,94,10101010
     1010101010,95,10101010101F
1430 N=23
1440 GOSUB 160
1450 PRINT TAB(21);"@PHALANG": :TAB(12);"
     RADIUS
1460 CALL HCHAR(22,31,69)
1470 CALL HCHAR(22,32,83)
1480 PRINT : : :TAB(12);"ULNA  ^_METACA
     RPU"
1490 CALL HCHAR(23,31,83)
1500 PRINT " HUMERUS": :TAB(18);"_CARPU
     S": : : : : : : : :
1510 CALL VCHAR(12,20,94,2)
1520 RESTORE 1530
1530 DATA 8,2,97,1,9,2,98,1,10,2,99,1,8
     ,3,96,8,9,3,96,8,10,3,96,8,8,11,10
     0,1
1540 DATA 9,11,101,1,10,11,102,1,8,12,1
     04,8,9,12,113,8,10,12,112,8,8,20,1
     21,1,9,20,121,1
1550 DATA 10,20,121,1,7,21,145,1,8,21,1
     29,1,9,21,130,1,10,21,131,1,7,22,1
     46,1,8,22,138,2
1560 DATA 9,22,140,2,8,24,139,1,9,24,13
     7,1,10,22,141,1,10,23,142,1,10,24,
     143,1,10,29,32,1
1570 N=27
1580 GOSUB 320
1590 GOSUB 230
1600 N=7
1610 RESTORE 1620
1620 DATA 6,23,32,10,7,14,32,6,7,23,32,
     1,11,14,32,18,12,4,32,19,13,20,32,
     1,14,20,32,7
1630 GOSUB 320
1640 R=6
1650 DATA HUMERUS,9,RADIUS,10,ULNA,11,C
     ARPUS,12,METACARPUS,13,PHLANGES,14
1660 RESTORE 1650
1670 GOSUB 370
1680 GOTO 1660
1690 RESTORE 1700
1700 DATA 43,FFFF00FFFFFFFFFF,33,FFFFFF
     FF1F1F1F0F,34,FFFFFFFFFE0F8FCFE,35,
     FFFFFFFFF
1710 DATA 36,FFFFFFFF0F071F3F7F,37,FFFFFF
     FFFFF8F8F8F8F8,87,07070707070707078
     ,E0E0E0E0E0E0E0E0E0E0
1720 DATA 98,00C0F0FCFFFFFF,101,000300
     F3F7FFFFFFFF,99,FF1F0701,102,FFF8E0
     8,105,3FFFFFFFF1F1FFFFF
1730 DATA 106,FFF7E7F7F7F7F7FF,107,FCFF
     FFFFF8F8FFFF,108,0707070FFFFFFF0101,
     109,E0E0E0E0FFFFFF8080
```

```
1740 DATA 113,000000FFFFFF,114,060F1F03
     C0C0FFFF,117,C0F0F8C00303FFFF,115,
     0000C0E18F1F
1750 DATA 116,06060CF8F8F,119,6060301F1
     F0F,121,0001030700F1F3F7F,125,0080C
     0E0F0F8FCFE
1760 DATA 123,7F3F1F0F070301,127,FEFCF8
     F0E0C08,133,EFF7E7E7C381,132,F7EFE
     7E7C381,145,7D413E3E1C08
1770 DATA 122,000000FFFFFFFFFF,137,7C7E
     3F3F0F0F0701,141,3E7EFCFCF0F0E08,1
     38,00010101C7FFFFFF
1780 DATA 140,00808080E3FFFFFF,139,FFFF
     FFF7F7E3C1,129,FFC1DDF0F3E7CFC1,15
     1,FFFFFFC1FDFDE1FD
1790 DATA 130,00C1E1F7F7F7F7F7,134,0033
     87EFEFEFEFEF,131,EFEFEFEFEFEFEFEF,
     135,F7F7F7F7F7F7F7F7
1800 N=42
1810 GOSUB 160
1820 CALL COLOR(1,12,1)
1830 CALL COLOR(2,12,1)
1840 PRINT TAB(18);"CLAVICLE": : :TAB(2
     2);"SCAPULA":"1 STERNUM"
1850 PRINT : :TAB(22);"RIBS": : : :TAB(
     16);"SPINE": : :"2 SACRUM"
1860 PRINT TAB(20);"ILIUM":"3 COCCYX":
     :TAB(18);"ISCHIUM": : :
1870 RESTORE 1890
1880 CALL VCHAR(2,17,45,14)
1890 DATA 4,11,33,1,4,12,34,1,4,13,35,3
     ,4,16,36,1,4,18,34,1,4,19,35,3,4,2
     2,36,1,4,23,37,1
1900 DATA 5,11,97,1,5,12,96,1,5,13,98,1
     ,5,15,114,1,5,16,105,1,5,17,106,1,
     5,18,107,1,5,19,117,1
1910 DATA 5,21,101,1,5,22,96,1,5,23,100
     ,1,6,11,97,1,6,12,96,2,6,14,113,2,
     6,16,108,1
1920 DATA 6,17,104,1,6,18,109,1,6,19,11
     3,2,6,21,96,2,6,23,100,1,7,12,99,1
     ,7,13,113,9
1930 DATA 7,17,104,1,7,22,102,1,8,12,11
     3,11,8,17,104,1,9,12,113,4,9,16,11
     6,1,9,18,119,1,9,19,113,4
1940 DATA 10,11,113,5,10,19,113,5,11,11
     ,113,4,11,15,116,1,11,19,119,1,11,
     20,113,4,12,10,113,5
1950 DATA 12,20,113,5,13,10,113,4,13,14
     ,116,1,13,20,119,1,13,21,113,4,16,
     13,121,1,16,14,120,3
1960 DATA 16,16,130,1,16,17,128,1,16,18
     ,134,1,16,19,120,2,16,21,125,1,17,
     13,120,3,17,16,131,1
1970 DATA 17,17,129,1,17,18,135,1,17,19
     ,120,3,18,13,120,3,18,16,133,1,18,
     17,151,1,18,18,132,1
1980 DATA 18,19,120,3,19,13,123,1,19,14
     ,120,2,19,17,145,1,19,19,120,2,19,
     21,127,1,20,14,123,1
1990 DATA 20,15,120,1,20,16,122,3,20,19
     ,120,1
2000 DATA 20,20,127,1,21,15,137,1,21,16
     ,138,1,21,17,139,1,21,18,140,1,21,
     19,141,1,24,30,32,1
2010 N=84
2020 GOSUB 320
2030 GOSUB 230
2040 RESTORE 2050
2050 DATA 3,20,32,8,7,3,32,9,6,24,32,7,
     10,24,32,4,14,18,32,5,18,22,32,5,1
     7,3,32,8
2060 DATA 19,3,32,8,21,20,32,7,5,17,104
     ,1,17,17,128,1,18,17,128,1,18,17,1
     28,1
2070 N=13
2080 GOSUB 320
2090 CALL CHAR(145,"7F7F3E3E1C08")
2100 DATA CLAVICLE,1,SPINE,2,SCAPULA,9,
     STERNUM,10,RIBS,11,ILIUM,12,SACRUM
     ,13,COCCYX,15,ISCHIUM,14
2110 R=9
2120 RESTORE 2100
2130 GOSUB 370
2140 CALL VCHAR(16,13,121)
2150 CALL VCHAR(17,13,120,2)
2160 CALL VCHAR(19,13,123)
2170 GOTO 2120
2180 RESTORE 2190
2190 DATA 97,F0F8F8F8F0E0C08,98,071F7F7
     F7F7F7F7F,99,3F1F0F0707030301,100,
     FFFFFFFFFFFEFC
2200 DATA 105,7E7F7F7F7F7F7F3E,113,7FFF
     FFFF7F7F7F7F,114,00C0FFFFFFFFFFFF,
     115,FFFFFFFFFFFFFFE8
2210 DATA 122,FFCF8F8F0F0F0F0F,121,0F0F
     0F0F0F0F0F0F,129,3E7FFFFFFFFFFF7F,
     137,1C3F1F0F07030F0F
2220 DATA 138,38BEBFBF00F8FEFF,139,FFFF
     8F0F070707,140,FFFF01FEFF01FEFF,14
     5,FCFEFF03F8FE07E
2230 DATA 146,F80EF678,147,1F0F03,130,0
     0F0FEFFFFFFFFFE
2240 N=19
2250 GOSUB 160
2260 PRINT TAB(14);"FEMUR": : : : : : :
     TAB(14);"PATELLA": : :TAB(7);"TIBI
     A"
2270 PRINT :TAB(14);"FIBULA": : : : :TA
     B(14);"TARSUS": :"   METATARSUS
     PHALANGES"
2280 CALL VCHAR(2,14,96,9)
2290 CALL VCHAR(2,15,96,8)
2300 CALL VCHAR(13,14,112,7)
2310 CALL VCHAR(14,15,121,7)
2320 RESTORE 2330
2330 DATA 2,13,98,1,2,16,97,1,3,13,99,1
     ,10,15,100,1,11,14,100,1,11,15,105
     ,1,12,14,113,1
2340 DATA 12,15,114,1,13,15,122,1,20,14
     ,115,1,21,14,129,1,21,15,130,1,22,
     14,137,1,22,15,138,1
2350 DATA 23,15,139,1,23,16,140,2,23,18
     ,145,1,24,18,146,1,24,17,147,1,24,
     30,32,1
2360 N=20
2370 GOSUB 320
2380 GOSUB 230
2390 RESTORE 2400
2400 DATA 4,16,32,5,11,16,32,7,14,9,32,
     5,16,16,32,6,21,16,32,6,23,5,32,10
     ,23,19,32,9
2410 N=7
2420 GOSUB 320
2430 R=7
2440 DATA FEMUR,9,PATELLA,10,TIBIA,11,F
     IBULA,12,TARSUS,13,METATARSUS,14,P
     HALANGES,15
2450 RESTORE 2440
2460 GOSUB 370
2470 GOTO 2450
2480 STOP
2490 CALL CLEAR
2500 PRINT TAB(7);"NAME THAT BONE": : :
     : : : : : : : : : : : : : : : :
2510 A$="FFFFFFFFFFFFFFFF"
2520 CALL CHAR(96,A$)
2530 CALL COLOR(9,7,1)
2540 CALL HCHAR(7,15,96,3)
2550 CALL HCHAR(8,15,96,3)
2560 CALL HCHAR(9,15,96,3)
2570 CALL VCHAR(10,16,96,6)
2580 CALL HCHAR(11,13,96,7)
2590 CALL VCHAR(15,15,96,6)
```

```
2600 CALL VCHAR(15,17,96,6)
2610 CALL COLOR(2,2,1)
2620 IF FLAG=2 THEN 2830
2630 FOR I=1 TO 7
2640 CALL COLOR(9+I,12,1)
2650 CALL CHAR(96+8*I,A$)
2660 NEXT I
2670 FLAG=2
2680 PRINT "INSTRUCTIONS? (Y/N)"
2690 CALL KEY(0,K,S)
2700 IF K=78 THEN 2830
2710 IF K<>89 THEN 2690
2720 CALL CLEAR
2730 PRINT "YOU MAY STUDY THE NAMES OF
     THE BONES AS LONG AS YOU
2740 PRINT "WISH. THEN PRESS <ENTER>."
2750 PRINT "IT IS YOUR TURN TO NAME
     THE LABELS WILL CLEAR, T
     HEN "IT IS YOUR TURN TO NAME
2760 PRINT "THAT BONE - CHOOSE THE
     CORRECT NUMBER."
2770 PRINT "YOU MUST NAME THE BONES
     CORRECTLY TO CONTINUE."
2780 GOSUB 230
2790 FLAG=2
2800 GOTO 2490
2810 DATA 67,72,79,79,83,69,58,49,32,72
     ,69,65,68,32,50,32,65,82,77,83,32,
     51,32
2820 DATA 84,79,82,83,79,52,32,76,69,71
     ,83,32,53,32,69,78,68,32,32
2830 RESTORE 2810
2840 CALL HCHAR(23,1,32,21)
2850 FOR X=7 TO 17 STEP 2
2860 FOR Y=23 TO 29
2870 READ G
2880 CALL HCHAR(X,Y,G)
2890 NEXT Y
2900 NEXT X
2910 CALL KEY(0,K,S)
2920 IF S<1 THEN 2910
2930 IF K=53 THEN 2990
2940 IF (K>52)+(K<49)=-1 THEN 2910
2950 CALL CLEAR
2960 PRINT "ONE MOMENT PLEASE"
2970 CALL COLOR(9,12,1)
2980 ON K-48 GOTO 990,1360,1690,2180
2990 CALL CLEAR
3000 END
```

Computer Techniques for Tutoring the Mentally Handicapped

Huzzah, the revolution has just started! And the fact that you're reading *The Best of 99'er* signifies that you are very much a part of it—a revolution fueled by the availability and affordability of computer power to millions of consumers. As more and more software—computer programs that can meet a large number of everyday needs, as well as solve problems encountered in special areas—is developed, the computer will become as common in our homes as the telephone.

Our task in this generation is to learn to take advantage of this tool in a variety of areas, disciplines and endeavors. In this article, we would like to focus the application of computer technology on what may seem at first to be a most unlikely area—tutoring the developmentally disabled.

Retardation is defined as "below average intellectual functioning that originates during the developmental stages with associated maladaptive behavior." In the search for tools to combat retardation, the microcomputer has shown itself to be extremely valuable by assisting the retarded population to develop skills, abilities, concepts, and even behaviors. Preliminary testing demonstrates that not only can these individuals use a keyboard, but they can learn it very quickly—finding it attractive, novel and magnetic. Options such as the light pen, joystick, and voice synthesizer provide capabilities that can be used to adapt numerous programs for this special population.

Help for the Schools

Of more than eight million handicapped children in the U.S., reportedly only half are receiving appropriate educational services. School districts under ever-tightening budgets struggle to meet the needs of these children. It therefore appears highly probable that using microcomputers to assist in meeting the needs of these children will be both an economic boon to schools, and a valuable enhancement to the learning process of these youngsters.

Despite traditional controversies regarding the learning process, there are some areas of general agreement. These areas have provided us with a basis for software geared to the special learning needs of the retarded—programs utilizing the unique qualities of the computer to further stimulate learning.

Fascination With the Medium

Retarded and non-retarded alike are able to learn more, as well as more easily, from teaching aids that effectively focus their attention on the content. Attention management for the retarded youngster is especially critical. In this regard, the computer, keyboard, and CRT have a fascination that commands attention with an immediacy that is unparalleled. When a youngster is seated before a console, the attraction of the mechanism coupled with the allure of a good interactive program provides an incredible amount of motivation and drive. If you have children who play computer games or other electronic games using a microprocessor, you already know just how difficult it is to distract them and draw their attention to something else—like homework, eating, or cleaning their room.

Nothing Succeeds Like Success

As human beings, we tend to strive toward success or try to avoid failure. In the search for success, the "locus of control" is usually internal. This is to say that in the process of maturing, a person begins to realize a power or ability to control events, and begins to set goals. We begin to become efficient in attaining goals. Actually attaining them brings a sense of success which is its own reward and prompts one to continue to strive for success.

Avoiding failure, on the other hand, means maintaining a mere minimum of effort so as not to incur some type of punishment. Consequently, the locus of control is external. For a majority of developmentally disabled, avoiding punishment becomes the usual way of behaving. They are not given to setting goals since they have not come to experience the internal locus of control and the possibility of success.

With the use of computers, a learning environment can be created which can provide a retarded child or adult with the experience of success. As the experience is repeated, the locus of control begins to shift from without to within. This is a natural reward process which has more lasting effects than punishment or negative reinforcement. As the repertoire is gradually expanded, the retarded individual begins to realize a potential: a power for success.

A Multisensory Lens

Another important element in the learning process of the retarded person is the ability to focus in on significant cues. Once again, the hardware's attractiveness (or novelty, if you will) is so engaging and attention-riveting (thereby limiting external or irrelevant stimuli or signals) that the person learns to be attentive to only the important and discriminating cues. Furthermore, the multisensory impact of the computer provides an additional quality which is extremely valuable in the learning process of the retarded person: The more you can use, engage, and impact many sensory modalities—

and do it repeatedly in an interesting manner—the greater the likelihood of retention and learning.

An Example Program

The following is a simple program designed for teaching retarded persons the extremely abstract concepts of number recognition, counting and subtraction. We feel that the program demonstrates the principles stated in this article, as well as the uniqueness of the computer as a tool especially well-suited to meeting the learning needs of the developmentally disabled. We wish to emphasize that the computer does not totally substitute for a teacher. The retarded individuals on whom we tested the program required personal assistance and encouragement at the beginning of the lesson. Reaction to the computer ranged from reluctance to eager enthusiasm. In some cases, we first used another program (a keyboard trainer) to familiarize the student with the key locations on the console. The TI-99/4 keyboard is highly suited for use by those unfamiliar with typewriters. We found it helpful, however, to cover the letter keys with masking tape to reduce distractions. Also, we noted some confusion created by the shift characters above each number—a small problem we hope to overcome by trying a number of key covers. Based on field testing of this program, we are convinced that this approach can be extended to many areas of work with this group, a group whose needs are so unique that conventional methods have been only moderately successful. Using this technology, we have a potential for far greater success and the possibility of doing things that were unthought-of for this segment of the population.

The Program

The program opens with several options which must be selected. The instructor is informed that a performance rating of the student's progress is available by pressing the AID key. This rating gives the number of trials, correct answers, and percent correct. If you wish to reset the options later, simply press the BACK command and re-enter. The AID and BACK commands can be entered during the main lesson, thus giving the instructor flexibility in choosing the set of options most appropriate to the student's level of ability. The program also has a speech selection option that permits its use without the Speech Synthesizer and *Speech Editor* Command Cartridge. [The extensive use of graphics in this program precludes the use of the speech editor resident in the *Extended BASIC* Command Cartridge with its fewer available character sets.—Ed.] Although the actual lesson is designed for non-readers, the initial option selection must be performed by an instructor or someone who can read. These options can be selected in any combination from the following list:

Select:
- 1 = Random presentation
- 2 = Serial presentation
- 3 = End lesson
- —Display the number above the gulls (Y/N)
- —Pronounce each number as it is printed (Y/N)
- —Computer says press__(number) after a row of gulls is put on the screen (Y/N)

Select format for placing of gulls on the ocean:
- 1 = Horizontal Row
- 2 = Diagonal Pattern
- 3 = Random row placement

After the options are selected, the screen clears and a seascape is painted on the screen. A picture of a deep blue ocean and a steamship liner on the horizon moving toward a tropical island focuses the student's attention immediately. The gulls appear on the water from left to right, and a shark's fin begins to circle the gulls while waiting for the student to press the key representing the number of gulls. If the response is correct, a musical fanfare is played, followed by the computer saying "Right (number)," and the ship moves one column to the left, emitting smoke puffs from the stacks (the number of puffs equal to the number of gulls). However, if the student's response is incorrect, the computer says, "Uh oh," and the shark stops circling the gulls, emerges from the water and devours the last gull (with sound effects)! Then the computer asks the student, "What number is left?" and waits for the student to press the key representing the number of remaining gulls. If incorrect again, the computer says, "That is incorrect," gives a short laugh, and then engulfs the next gull! This can continue until no gulls remain; the program then recycles and another trial begins. On a correct response the computer says "Right (number)" and the ship is advanced one column to the left with the appropriate number of smoke puffs. Each correct response advances the ship toward the island until it is "docked" and the computers says, "You win." It then recycles the program, placing the ship back at the right side of the screen, and continues the lesson.

We recommend that students start with the Serial option rather that the Random. This starts with the number 1 and adds a number on each correct trial, but will not add a number on an error. In this way, a student cannot be challenged by the larger numbers until he has displayed mastery of the smaller ones. In general, we also recommend the strategy of starting a student with all prompt options operating, then removing them as the student demonstrates competence.

EXPLANATION OF THE PROGRAM
Computer Techniques for Tutoring the Mentally Handicapped

Line Nos.	
160-280	Sets all variables to zero.
290-820	Instructor selects program options.
830-1310	Defines characters and color codes.
1320-1450	Constructs seascape, boat, and island.
1460-1550	Calculates the appropriate number of gulls to place on screen.
1560-1590	Clears screen from row 10 to 24.
1600-1820	Places gulls in the water.
1830-1890	Controls movement of shark fin from left to right.
1900-1960	Evaluates key response while shark circles gulls.
1970-2120	Musical fanfare on correct response.
2130-2220	Controls movement of shark fin from right to left background.
2130-2290	Evaluates key response.
2300-2530	Controls animation of shark eating gulls.
2540-2620	Evaluates key response and clears screen to right of last gull after shark "eats" it.
2630-2660	Controls loop to eat next gull.
2670-2740	Verbal response to correct key press; increments number of trials and right responses.
2750-2810	Moves ship, controls puffs of smoke and sound effects from ship stacks.
2920-2950	Prompts to press a number if a letter was pressed.
2960-3060	Routine when boat reaches island.
3070-3100	Calculates performance scores.
3110-3170	Prints option to end and branches appropriately.

```
100 REM  ********************
110 REM  *  COUNTING LESSON  *
120 REM  ********************
130 REM
140 REM
150 REM
160 CALL CLEAR
170 CALL SCREEN(8)
180 REM  SET VARIABLES TO          ZERO
190 FOR LOOP=1 TO 10
200 P(LOOP)=0
210 NEXT LOOP
220 T=0
230 DN=0
240 PN=0
250 L=0
260 N=0
270 CT=0
280 NM=0
290 REM  MENUS
300 PRINT "     INSTRUCTOR PLEASE NOTE"
310 PRINT "**********************
    *"::
320 PRINT "DURING THE LESSON":"YOU CAN
    PRESS <AID> TO GET":"A READOUT OF
    PERCENT OF":"CORRECT RESPONSES"
330 PRINT :"IF YOU WISH TO RESET PROGR
    AM":"OPTIONS PRESS THE <BACK> KEY"
    ::
340 PRINT :"DO YOU WANT TO USE SPEECH"
    :"OPTION? (Y/N) (SPEECH EDITOR"::"A
    ND SYNTHESIZER REQUIRED)"
350 CALL KEY(0,K,S)
360 IF S=0 THEN 350
370 IF K=89 THEN 430
380 IF K<>78 THEN 350
390 SP=1
400 PN=1
410 CT=1
420 GOTO 440
430 SP=0
440 PRINT ::::::"  SELECT FROM THE FOLL
    OWING":"**********
    *"::
450 PRINT TAB(8);"1-RANDOM GROUPS":TAB
    (8);"2-SEQUENTIAL":TAB(10);"PRESEN
    TATION"
460 PRINT TAB(8);"3-END LESSON"
470 CALL KEY(0,K,S)
480 IF S=0 THEN 470
490 IF K=50 THEN 530
500 IF K=51 THEN 3070
510 IF K<>49 THEN 470
520 L=1
530 PRINT :"DO YOU WANT TO DISPLAY THE
    ":"NUMBERS ABOVE THE GULLS?":"(Y/N
    )"::
540 CALL KEY(0,K,S)
550 IF S=0 THEN 540
560 IF K=89 THEN 590
570 IF K<>78 THEN 540
580 DN=1
590 IF SP=1 THEN 720
600 PRINT "DO YOU WANT TO PRONOUNCE":"
    EACH NUMBER AS IT IS PRINT-":"ED?
    (Y/N)"::
610 CALL KEY(0,K,S)
620 IF S=0 THEN 610
630 IF K=89 THEN 660
640 IF K<>78 THEN 610
650 PN=1
660 PRINT "DO YOU WANT THE COMPUTER TO
    ":"TELL THE STUDENT WHICH KEY":"TO
     PRESS FOR EACH TRIAL?":"(Y/N)"
670 CALL KEY(0,K,S)
680 IF S=0 THEN 670
690 IF K=89 THEN 720
700 IF K<>78 THEN 670
710 CT=1
720 PRINT :::"DO YOU WANT THE GULLS TO
    BE":"PRESENTED:":"     1-ALL ON SAME
     LINE":"     2-DIAGONAL":"
    3-RANDOM"
730 CALL KEY(0,K,S)
740 IF S=0 THEN 730
750 IF K=49 THEN 820
760 IF K=50 THEN 800
770 IF K<>51 THEN 730
780 LN=2
790 GOTO 840
800 LN=1
810 GOTO 840
820 LN=0
830 REM  DEFINE CHARACTERS
840 RANDOMIZE
850 CALL CLEAR
860 CALL SCREEN(8)
870 A$="E0F0F8FCFEFFFFFB"
880 B$="7F3F1F0F070FFFFF"
890 C$="00000000000080C0"
900 D$="E0F0F8FCFEFFFFFF"
910 E$=""
920 F$="0103070F1F3F7FFF"
930 G$="0001010101030707F"
940 H$="8080807F0F8FCFEFF"
950 I$="FFF7FFBAFFBFEBFF"
960 J$="3337F7FF5FFAFFFF"
970 K$="0000000003F1F0F07"
980 L$="FFAAFFFFFFFFFFFF"
990 M$="0000007FFFFFFFEC"
1000 N$="000000000C3C3C3C3"
1010 O$="0000000000008142"
1020 P$="000E0F0C86FE7E3C"
1030 Q$=""
1040 R$="41820C300C3C3C3"
1050 S$=""
1060 CALL CHAR(96,A$)
1070 CALL CHAR(97,B$)
1080 CALL CHAR(98,C$)
1090 CALL CHAR(99,D$)
1100 CALL CHAR(104,E$)
1110 CALL CHAR(100,F$)
1120 CALL CHAR(101,G$)
1130 CALL CHAR(112,H$)
1140 CALL CHAR(113,I$)
1150 CALL CHAR(114,J$)
1160 CALL CHAR(120,K$)
1170 CALL CHAR(121,L$)
1180 CALL CHAR(122,M$)
1190 CALL CHAR(128,N$)
1200 CALL CHAR(136,O$)
1210 CALL CHAR(102,P$)
1220 CALL CHAR(140,Q$)
1230 CALL CHAR(144,R$)
1240 CALL CHAR(145,S$)
1250 CALL COLOR(14,8,6)
1260 CALL COLOR(9,16,6)
1270 CALL COLOR(11,13,11)
1280 CALL COLOR(10,11,11)
1290 CALL COLOR(12,7,11)
1300 CALL COLOR(13,2,11)
1310 CALL COLOR(15,16,11)
1320 CALL HCHAR(1,1,104,288)
1330 CALL HCHAR(10,1,140,480)
1340 CALL HCHAR(9,10,112)
1350 CALL HCHAR(9,1,113,9)
1360 CALL HCHAR(8,9,112)
1370 CALL HCHAR(8,1,113,8)
1380 CALL HCHAR(7,8,112)
1390 CALL HCHAR(7,1,113,7)
1400 CALL HCHAR(6,7,112)
1410 CALL HCHAR(6,1,114,6)
1420 CALL HCHAR(9,27-X,120)
1430 CALL HCHAR(9,28-X,121,3)
1440 CALL HCHAR(9,31-X,122)
1450 CALL HCHAR(8,29-X,128)
1460 REM  CALCULATE NUMBER OF GULLS
1470 IF L<>0 THEN 1540
```

```
1480 IF T=1 THEN 1540
1490 N=N+1
1500 IF N>9 THEN 1540
1510 GOTO 1570
1520 N=1
1530 GOTO 1570
1540 N=INT(RND*9)+1
1550 T=1
1560 REM  CLEAR 'WATER' AREA
1570 FOR ERASE=10 TO 24
1580 CALL HCHAR(ERASE,1,140,32)
1590 NEXT ERASE
1600 REM  LOOP PLACES GULLS IN WATER
1610 FOR J=1 TO (2*N) STEP 2
1620 NM=NM+1
1630 REM  CALCULATE POSITION OF GULLS
1640 IF LN=0 THEN 1690
1650 IF LN=1 THEN 1680
1660 P(NM)=(RND*8)+1
1670 GOTO 1690
1680 P(NM)=I/2
1690 CALL HCHAR(12+P(NM),J+9,136)
1700 IF DN=1 THEN 1720
1710 CALL HCHAR(11+P(NM),J+10,ASC(STR$(
     NM)))
1720 CALL HCHAR(12+P(NM),J+10,102)
1730 IF PN=1 THEN 1750
1740 CALL SAY(STR$(NM))
1750 NEXT J
1760 C=1
1770 CALL HCHAR(10,1,140,32)
1780 FS='80C0E0F0F8FCFEFF'
1790 CALL CHAR(100,FS)
1800 IF CT=1 THEN 1840
1810 CALL SAY('PRESS')
1820 CALL SAY(STR$(NM))
1830 REM  START SHARK FIN ACROSS SCREEN
      IN FOREGROUND
1840 CALL HCHAR(22,C,100)
1850 CALL SOUND(25,110,0)
1860 CALL HCHAR(22,C,136)
1870 C=C+1
1880 FB=0
1890 IF C=32 THEN 2140
1900 CALL KEY(0,K,STATUS)
1910 IF STATUS=0 THEN 1840
1920 IF K=1 THEN 3080
1930 IF K=15 THEN 160
1940 IF K<48 THEN 2920
1950 IF K>57 THEN 2920
1960 NN=K-48
1970 IF NN<>NM THEN 2330
1980 NM=0
1990 REM  PLAY MUSICAL FLOURISH ON CORR
     ECT RESPONSE
2000 CALL SOUND(100,349,0)
2010 CALL SOUND(100,440,0)
2020 CALL SOUND(100,523,0)
2030 CALL SOUND(100,523,0)
2040 CALL SOUND(100,523,0)
2050 CALL SOUND(100,440,0)
2060 CALL SOUND(100,440,0)
2070 CALL SOUND(100,440,0)
2080 CALL SOUND(100,349,0)
2090 CALL SOUND(100,440,0)
2100 CALL SOUND(100,349,0)
2110 CALL SOUND(100,262,0)
2120 GOTO 2690
2130 REM  START SHARK FIN ACROSS SCREEN
      IN BACKGROUND
2140 FS='0000010307070F1F3F'
2150 CALL HCHAR(22,1,140,32)
2160 CALL CHAR(100,FS)
2170 CALL HCHAR(10,C,100)
2180 CALL SOUND(25,110,5)
2190 CALL HCHAR(10,C,136)
2200 IF C=1 THEN 1760
2210 C=C-1
2220 FB=1
2230 CALL KEY(0,K,STATUS)
2240 IF STATUS=0 THEN 2170
2250 IF K=1 THEN 3080
2260 IF K=15 THEN 160
2270 IF K<48 THEN 2920
2280 IF K>57 THEN 2920
2290 NN=K-48
2300 IF NN<>N THEN 2330
2310 GOTO 1980
2320 REM  SHARK EATS GULL ON WRONG ANS
     WER
2330 R=10
2340 IF SP=1 THEN 2360
2350 CALL SAY('UHOH')
2360 TRIAL=TRIAL+1
2370 FS='FF7E3C1800000000'
2380 CALL CHAR(100,FS)
2390 FOR I=9+(2*N) TO 10 STEP -2
2400 FOR J=1 TO 2
2410 BS='7F3F3F3F1F0F0703'
2420 CALL CHAR(97,BS)
2430 CALL HCHAR(11+P(NM),I,96)
2440 CALL HCHAR(11+P(NM),I+1,98)
2450 CALL HCHAR(12+P(NM),I,97)
2460 CALL HCHAR(12+P(NM),I+1,99)
2470 CALL HCHAR(13+P(NM),I+3,100)
2480 CALL SOUND(50,-5,0)
2490 BS='7F3F1F0F070FFFFF'
2500 CALL CHAR(97,BS)
2510 NEXT J
2520 NM=NM-1
2530 N=N-1
2540 REM  AFTER EATING GULL/WAITS FOR C
     ORRECT ANSWER FOR NUMBER LEFT
2550 IF SP=1 THEN 2570
2560 CALL SAY('WHAT+NUMBER+IS+LEFT')
2570 CALL KEY(0,K,S)
2580 IF S=0 THEN 2570
2590 FOR LOOP=10 TO 24
2600 CALL HCHAR(LOOP,I,140,32-I)
2610 NEXT LOOP
2620 IF K=ASC(STR$(N)) THEN 2690
2630 TRIAL=TRIAL+1
2640 IF SP=1 THEN 2660
2650 CALL SAY('#THAT IS INCORRECT# A1+A
     1+A1')
2660 NEXT I
2670 NM=0
2680 IF N<1 THEN 1470
2690 IF SP=1 THEN 2720
2700 CALL SAY('RIGHT')
2710 CALL SAY(CHR$(K))
2720 RIGHT=RIGHT+1
2730 TRIAL=TRIAL+1
2740 NM=0
2750 REM  MOVES SHIP ACROSS SCREEN ONE
      COLUMN
2760 X=X+1
2770 IF X=17 THEN 2970
2780 CALL HCHAR(9,11,145,21)
2790 CALL HCHAR(8,11,145,21)
2800 CALL HCHAR(9,27-X,120)
2810 CALL HCHAR(9,28-X,121,3)
2820 CALL HCHAR(9,31-X,122)
2830 CALL HCHAR(8,29-X,128)
2840 FOR I=1 TO N
2850 CALL HCHAR(7,29-X,144)
2860 CALL SOUND(200,-5,0)
2870 FOR D=1 TO 50
2880 NEXT D
2890 CALL HCHAR(7,29-X,145)
2900 NEXT I
2910 GOTO 1470
2920 IF SP=1 THEN 2940
2930 CALL SAY('PRESS+A+NUMBER+PLEASE')
2940 TRIAL=TRIAL+1
2950 IF FB=1 THEN 2170 ELSE 1840
2960 REM  ROUTINE WHEN BOAT REACHES ISL
     AND
```

```
2970 X=0
2980 IF SP=1 THEN 3000
2990 CALL SAY("YOU WIN!")
3000 CALL SOUND(2000,220,0)
3010 CALL SOUND(50,220,30)
3020 CALL SOUND(1000,220,0)
3030 CALL SOUND(50,220,30)
3040 CALL SOUND(500,220,0)
3050 CALL CLEAR
3060 GOTO 1320
3070 REM  CALCULATE SCORES
3080 CALL CLEAR
```

```
3090 NM=0
3100 PRINT "TRIALS=";TRIAL::"RIGHT=";RI
     GHT::"PERCENT CORRECT=";INT(100*(R
     IGHT/TRIAL))
3110 PRINT :"TO RETURN TO LESSON PRESS
     1":"TO END LESSON PRESS 2"::
3120 CALL KEY(0,K,S)
3130 IF S=0 THEN 3120
3140 IF K=49 THEN 850
3150 IF K<>50 THEN 3120
3160 PRINT "SO LONG FOR NOW!"
3170 END
```

Typing for Accuracy

TI BASIC

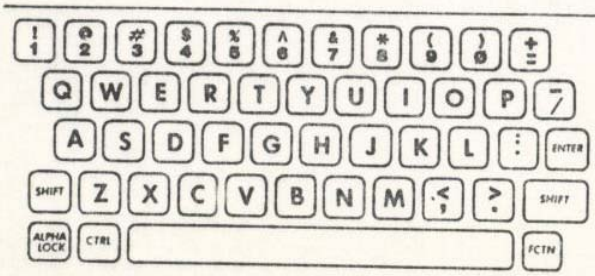
Typing for Accuracy provides practice for students who are somewhat familiar with the keyboard.

Seven finger-placement categories using different typewriter keys are offered: home keys; home row; top row, middle finger; top row, pointer finger; ring finger; little finger; and bottom row. A typist may choose one of the categories for each drill.

The program uses graphics and sound effects to liven up the drill: A rocket appears on the screen, and a word is printed on the rocket while a 1.5-second tone sounds. A student then types and ENTERs the word. If it has been typed incorrectly, the rocket blasts; if it has been typed correctly, a second tone sounds and the score goes up. The rocket then takes off (with gases trailing behind), and a different word appears.

At the end of ten words the student's score is tabulated and displayed as a percent accuracy rating. The student may then choose from the seven drills or may exit the program.

This drill is not meant to be a speed drill because beginning typing students must gain accuracy and familiarity with the keyboard before working on speed. However, if the student wants a timed test, an approximate words-per-minute rate can be estimated using the tones—i.e., if the student presses ENTER as the tone ends, the rate is 40 wpm.



EXPLANATION OF THE PROGRAM
Typing for Accuracy

Line Nos.	
120	Dimensions the array A$ to allow for twenty words.
130	Sets the y-coordinate for drawing the rocket.
140-200	Words used in the drills.
210	Prints the title screen.
220	Prints instructions.
230	Prints the menu screen of the seven categories.
240-400	Awaits the student's choice. Depending on the category chosen, a certain DATA statement is RESTOREd which contains the words for that particular category.
410-440	Draws the rocket.
450-480	Reads the number of words in the category and stores the words in the A$ array.
490	Initializes the score.
500-530	Randomly chooses a word. Once a word is chosen it is not used again in the drill.
540	Calculates the coordinate for printing the word.
550	Sounds the tone for 1.5 seconds.
560-580	Prints the word to be typed on the rocket.
590	Awaits the student's typed word.
600-640	Compares the student's word with the given word. If it is incorrect, a white noise is sounded; if it is correct, a tone sounds and the score is incremented.
650-700	If it is the first word, draws the bottom of the rocket.
710-730	If it is the second word, completes the fins of the rocket.
740-780	The rocket moves up and has a trail under it. The words are cleared.
790	A$ set to zero so the word cannot be used again.
800	Returns for the next word.
810-890	Prints score and waits until student is ready to continue.
910	END.
Subroutines:	
920-1530	Prints title screen with music.
1540-1680	Prints instructions.
1690-1780	Prints menu screen of seven categories.
1700-2110	Draws the rocket.

```
100 REM  * TYPING FOR ACCURACY *
110 REM
120 DIM A$(20)
130 Y=20
140 DATA 15,FAD,A,AS,DAD,AD,SAD,LAD,FA
    LL,ALFALFA,SASS,LASS,DADS,LADS,FAL
    LS,FADS
150 DATA 16,HAD,HAS,GAS,SAG,HALL,HALLS
    ,LADS,SAGS,HAG,LAG,LAGS,SLAG,SHALL
    ,SASH,DASH,FLASH
160 DATA 17,DEAF,DEED,SEED,FEED,HEED,L
    IKE,KILL,FILL,FEEL,FEES,LIED,DIAL,
    SLIDE,FLIES,LIFE,SLID,GLIDE
170 DATA 17,TAG,HAT,TALL,THY,DAY,HAY,J
    AY,GAY,LAY,TAR,RAT,STAR,STAFF,FAST
    ,TRY,SAY,YARD
180 DATA 20,WISH,EXAM,EXACT,TEXT,TWO,W
    ON,SOW,WASH,WORLD,OWE,WORD,LOOK,LO
    SE,SOD,WOW,TOW,TEXAS,OXEN,MIX,WORS
    E
190 DATA 16,QUAKE,QUIZ,QUIP,ZAP,QUIT,P
    IQUE,PLAQUE,PUZZLE,PLAZA,SAP,ZIPPE
    R,PRIZE,QUICK,SQUEEZE,ZEAL,ZIP
200 DATA 18,CALM,CAN,MEN,NIMBLE,EXACT,
    EXAM,MIX,NIX,BUZZ,ZOOM,NAVY,CAB,BA
    CK,BOMB,ZOMBIE,CAVE,VACATE,VARMINT
210 GOSUB 920
220 GOSUB 1540
230 GOSUB 1690
240 CALL KEY(0,KEY,STS)
250 IF KEY<49 THEN 240
260 IF KEY>56 THEN 240
270 ON (KEY-48)GOTO 280,300,320,340,36
    0,380,400,910
280 RESTORE 140
290 GOTO 410
300 RESTORE 150
310 GOTO 410
320 RESTORE 160
330 GOTO 410
340 RESTORE 170
350 GOTO 410
360 RESTORE 180
370 GOTO 410
380 RESTORE 190
390 GOTO 410
400 RESTORE 200
410 CALL CLEAR
420 CALL COLOR(1,2,1)
430 CALL SCREEN(8)
440 GOSUB 1890
450 READ N
460 FOR I=1 TO N
470 READ A$(I)
480 NEXT I
490 SCORE=0
500 RANDOMIZE
510 FOR K=1 TO 10
520 W=INT(N*RND)+1
530 IF A$(W)="0" THEN 520
540 XC=24-K
550 CALL SOUND(1500,-1,2)
560 FOR J=1 TO LEN(A$(W))
570 CALL HCHAR(XC,J+17,ASC(SEG$(A$(W),
    J,1)))
580 NEXT J
590 INPUT B$
600 IF B$=A$(W)THEN 630
610 CALL SOUND(1000,-7,1)
620 GOTO 650
630 CALL SOUND(1000,-2,1)
640 SCORE=SCORE+1
650 IF K<>1 THEN 710
660 CALL HCHAR(23,Y-4,98)
670 CALL HCHAR(23,Y+4,98)
680 CALL HCHAR(23,Y-3,99)
690 CALL HCHAR(23,Y+3,100)
700 GOTO 740
710 IF K<>2 THEN 740
720 CALL HCHAR(23,Y-4,99)
730 CALL HCHAR(23,Y+4,100)
740 CALL HCHAR(23,Y-1,105,3)
750 CALL SOUND(1,44000,30)
760 CALL HCHAR(XC-1,Y-2,98,7)
770 CALL HCHAR(XC-1,Y+5,32,3)
780 CALL HCHAR(23,1,32,15)
790 A$(W)="0"
800 NEXT K
810 CALL CLEAR
820 FOR I=2 TO 8
830 CALL COLOR(I,2,1)
840 NEXT I
850 SC=10*SCORE
860 PRINT ::::::"YOUR SCORE IS";SC;"PER
    CENT ACCURACY."
870 PRINT ::::"PRESS ENTER TO CONTINUE.
    "
880 CALL KEY(0,KEY,ST)
890 IF KEY<>13 THEN 880
900 GOTO 230
910 END
920 CALL CLEAR
930 CALL SCREEN(5)
940 T=500
950 CALL SOUND(T,880,3,698,8,294,10)
960 PRINT ::::TAB(11);"T Y P I N G"
970 CALL CHAR(104,"FFFF00FFFF00FFFF")
980 CALL SOUND(T,932,3,784,8,196,11)
990 PRINT ::TAB(15);"FOR"
1000 CALL COLOR(10,16,6)
1010 CALL COLOR(11,13,1)
1020 CALL SOUND(T,784,3,659,8,262,10)
1030 PRINT ::TAB(12);"ACCURACY"
1040 CALL CHAR(112,"00E0F8FEFFFFFFFF")
1050 CALL SOUND(T,880,3,698,8,175,12)
1060 CALL CHAR(113,"0000000000E0F8FE")
1070 CALL CHAR(114,"80E0FCFFFFFCE080")
1080 CALL CHAR(115,"FEF8E080")
1090 CALL SOUND(T,698,3,587,8,233,10)
1100 PRINT ::::::::
1110 PRINT ::::
1120 CALL CHAR(116,"FFFFFFFFFFEF8E")
1130 CALL SOUND(T/2,784,3,165,10)
1140 CALL CHAR(96,"80C0E0F8F8E0C080")
1150 CALL SOUND(T/2,698,3,165,10)
1160 CALL CHAR(97,"80C0E0F0F8FCFEFF")
1170 CALL SOUND(T/2,659,3,277,10)
1180 CALL CHAR(98,"FFFFFFFFFFFFFFFF")
1190 CALL CHAR(117,"FFFFFFFFFFFFFFFF")
1200 CALL SOUND(T/2,784,3,277,10)
1210 CALL CHAR(99,"FFFEFCF8F0E0C080")
1220 CALL SOUND(T*2,698,2,587,8,147,12)
1230 CALL HCHAR(15,7,117,7)
1240 CALL HCHAR(16,7,117,9)
1250 CALL HCHAR(17,7,117,7)
1260 CALL HCHAR(16,16,114)
1270 CALL HCHAR(15,15,113)
1280 CALL HCHAR(17,15,115)
1290 CALL HCHAR(15,14,112)
1300 CALL HCHAR(17,14,116)
1310 CALL SOUND(T,466,4,165,10)
1320 FOR XX=15 TO 17
1330 CALL HCHAR(XX,1,104,6)
1340 NEXT XX
1350 CALL SOUND(T,440,3,175,10)
1360 CALL COLOR(2,16,1)
1370 FOR YY=18 TO 26 STEP 2
1380 CALL HCHAR(16,YY,42)
1390 NEXT YY
1400 CALL SOUND(T,698,3,440,8,294,10)
1410 CALL HCHAR(16,28,49)
1420 CALL HCHAR(16,29,48,2)
1430 CALL HCHAR(16,31,37)
1440 CALL SOUND(T,784,3,587,8,233,10)
1450 CALL CHAR(100,"7F3F1F0F070301")
1460 CALL CHAR(101,"000103070F1F3F7F")
1470 CALL SOUND(T/2,698,3,392,8,262,10)
1480 CALL CHAR(102,"0000181818C3C7EFFF")
```

```
1490 CALL SOUND(T/2,659,2,262,10)
1500 CALL CHAR(105,"DBDBDBDBDBDBDBDB")
1510 CALL SOUND(4*T,698,2,440,8,175,10)
1520 CALL SOUND(1,44000,30)
1530 RETURN
1540 CALL CLEAR
1550 CALL SCREEN(7)
1560 CALL COLOR(2,2,1)
1570 PRINT "PICK A TYPING CATEGORY."
1580 PRINT :::"YOU WILL SEE A WORD"
1590 PRINT "IN THE ROCKET."
1600 PRINT :"TYPE AND ENTER IT BEFORE T
     HE TONE ENDS."
1610 PRINT :"IF YOU ARE CORRECT,"
1620 PRINT "ANOTHER TONE SOUNDS."
1630 PRINT :"IF YOU ARE INCORRECT."
1640 PRINT "YOU WILL BE BLASTED."::::
1650 PRINT "PRESS ENTER TO CONTINUE."
1660 CALL KEY(0,KEY,ST)
1670 IF KEY<>13 THEN 1660
1680 RETURN
1690 CALL CLEAR
1700 CALL SCREEN(12)
1710 CALL COLOR(1,2,12)
1720 FOR I=2 TO 8
1730 CALL COLOR(I,1,12)
1740 NEXT I
1750 PRINT :::::"CHOOSE ONE"::
1760 PRINT :" 1 HOME KEYS"
1770 PRINT :" 2 HOME ROW"
1780 PRINT :" 3 TOP ROW, MIDDLE FINGER
     "
1790 PRINT :" 4 TOP ROW, POINTER FINGE
     R"
```

```
1800 PRINT :" 5 RING FINGER"
1810 PRINT :" 6 LITTLE FINGER"
1820 PRINT :" 7 BOTTOM ROW"
1830 PRINT :" 8 END PROGRAM":::
1840 CALL SCREEN(5)
1850 FOR I=2 TO 8
1860 CALL COLOR(I,2,12)
1870 NEXT I
1880 RETURN
1890 CALL COLOR(9,1,1)
1900 CALL COLOR(10,1,1)
1910 CALL VCHAR(12,Y,98,13)
1920 CALL VCHAR(13,Y-1,98,12)
1930 CALL VCHAR(13,Y+1,98,12)
1940 CALL VCHAR(14,Y-2,98,11)
1950 CALL VCHAR(14,Y+2,98,11)
1960 CALL VCHAR(13,Y-2,101)
1970 CALL VCHAR(13,Y+2,97)
1980 CALL VCHAR(12,Y-1,101)
1990 CALL VCHAR(12,Y+1,97)
2000 CALL VCHAR(11,Y,102)
2010 CALL VCHAR(22,Y-3,98,3)
2020 CALL VCHAR(22,Y+3,98,3)
2030 CALL VCHAR(23,Y-4,98,2)
2040 CALL VCHAR(23,Y+4,98,2)
2050 CALL VCHAR(22,Y-4,101)
2060 CALL VCHAR(22,Y+4,97)
2070 CALL VCHAR(21,Y-3,101)
2080 CALL VCHAR(21,Y+3,97)
2090 CALL COLOR(9,7,1)
2100 CALL COLOR(10,16,1)
2110 RETURN
2120 END
```

CIVIL ENGINEERING

SIMPLE BEAMS

TI BASIC

The purpose of this program is to tutor civil engineering students who are studying statics or structures. It is limited to a simple determinate beam supported at the ends and loaded with a concentrated load, a uniform load, or a combination of a concentrated load and a uniform load. Basic knowledge of elementary statics is a prerequisite.

1. Concentrated load at the center

Newton's laws of force and moments are reviewed. The general solution of a load P applied at the center of a beam of length L is developed for the reaction forces A and B at each end of the beam. The student then does two problems. The load P and the length L are chosen randomly for the problems. If the student enters an incorrect solution, the correct one is given, and he is given another problem,

2. Concentrated load anywhere

The general solution of a load P applied a distance D from end A on a beam of length L is derived for the reaction forces A and B at each end. An example problem is given and solved. Then a problem is given for which the student enters his answers. The program prints the method of solution. For the next problem the student enters his solution. If he is incorrect, the program shows him how to solve the problem, and he is given another problem to solve.

3. Uniform load

The uniform load is considered as an equivalent concentrated load acting at the centroid of the loading pattern. The first example is a uniform load for the length of the beam and is solved in general terms. The student is then given a problem. If he enters an incorrect answer, he is shown the correct solution and given another problem.

4. Combination loads

Instructions are provided for placing a beam with one concentrated load and one uniform load. The student is then given a problem with combination loads chosen randomly. The program draws and labels the beam for each problem. If the student enters an incorrect solution, the correct solution is printed and he is given another problem.

5. Problems

No instruction is given. The program randomly chooses a beam length and loading pattern, and prints the problem. It then draws and labels the beam. The student enters his answers; if he is incorrect, the correct answers are given and another problem is printed.

6. Your own problem

The student enters the beam length and loading specifications. The program computes the reaction forces A and B at the ends.

After each section has been completed with correct solutions, the student is given the choice of having more of the same kind of problems, entering his own problems, or returning to the menu screen.

Programming Techniques

This program is a teaching aid or tutor, so it incorporates pauses, allowing the student to work on the problem before continuing. The student must enter a correct solution to the problem before he or she can go on to a different kind of problem. If the student enters an incorrect solution, the correct answers are printed and another problem of the same type is presented.

The numbers for each problem are chosen randomly (yet appropriately) for each beam. The length of the beam is between 10 and 20 feet. The concentrated load is 100 times a random number from one to twenty (i.e., 100 to 2000 pounds), and is placed at a distance D from end A (randomly chosen within the bounds of the length of the beam).

The uniform load is 10 times a random number from one to ten (i.e.,to 100 pounds per foot). For some of the problems, the uniform load is acting over the length of the beam.

For more advanced problems, it acts between two endpoints, L1 and L2, given as distances from end A. L1 must be equal to or greater than zero and less than the total length of the beam. L2 must be greater than L1 and less than or equal to the total length of the beam.

The problems are written in "story problem" form by using PRINT statements in subroutines, with the program using only the statements that are necessary for each loading condition.

After the student has had time to draw and label the problem on his own paper, the computer will ask him to "PRESS ENTER TO CONTINUE", and the beam will be drawn on the screen with approximate proportions.

The general beam is drawn with a pin at end A and a roller at end B.

The distances are approximated by using a variable y-coordinate—an integer value of the fraction of the distance (D or L1) divided by the total beam length multiplied by the number of characters printed in the general beam. For example, Statement 6750 is:

$$Y = INT(L1/LL*22) + 6$$

Y is the y-coordinate used in CALL HCHAR or CALL VCHAR statements. And 6 is the displacement of the end of the beam from the left side of the screen.

In statement 6760:

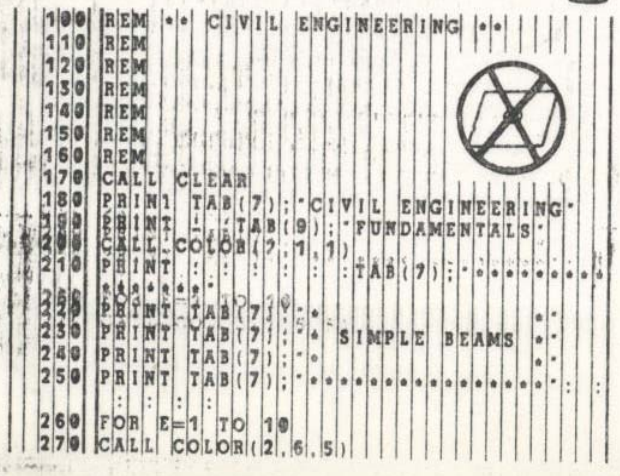
$$Z = INT(D2/LL*22) - 1$$

Z is the number of characters to be printed horizontally for the uniform load. D2 is the distance L2 – L1.

Figure 1.

LB$=STR$(PP)	Converts PP to a string variable.
For II=1 TO LEN(LB$)	LEN finds the length of LB$.
JJ=II+J-4	Calculation for y-coordinate.
CALL HCHAR(I-5,JJ,ASC (SEG$(LB$,II,1)))	Prints each digit in order.
NEXT II	
CALL HCHAR(I-5,JJ+1,32)	Prints a space after last digit.
CALL HCHAR(I-5,JJ+2,76)	Prints L.
CALL HCHAR(I-5,JJ+3,66)	Prints B.
CALL HCHAR(I-5,JJ+4,83)	Prints S.

The labels for the values on the beam are variable and are printed using string variables. For example, the concentrated load P may be three or four digits long (100 pounds to 2000 pounds) in the written problems, but the student may input an even longer number. This label printed by using Statements 5850-5930 (see Figure 1).



```
100 REM  **  CIVIL ENGINEERING  **
110 REM
120 REM
130 REM
140 REM
150 REM
160 REM
170 CALL CLEAR
180 PRINT TAB(7);"CIVIL ENGINEERING"
190 PRINT TAB(9);"FUNDAMENTALS"
200 CALL COLOR(7,1,1)
210 PRINT :::::::::TAB(7);"**********"
220 PRINT TAB(7);"*        *"
230 PRINT TAB(7);"*  SIMPLE BEAMS  *"
240 PRINT TAB(7);"*        *"
250 PRINT TAB(7);"**********"
260 FOR E=1 TO 10
270 CALL COLOR(2,6,5)
```

EXPLANATION OF THE PROGRAM

Civil Engineering Fundamentals

Line Nos.	
100-250	Prints the title screen.
250-330	Blinks a blue border.
340	Clears the screen.
350-540	Defines special graphics characters for drawing the beam and loading, and sets the color for them.
550-680	Prints second screen, diagram of simple beam.
690	Goes to menu screen for choice of problems.
700	Choices 1 and 2, concentrated loads, branch to here.
710-810	Prints instruction screen.
820-900	Prints second instruction screen.
910	For choice 2, branches to 1720.
920-1000	Prints problem.
1010-1070	Draws and labels general beam.
1080-1180	Shows solution of reaction forces in general terms.
1190-1270	Draws and labels beam with centrally-applied load.
1280-1330	General statement for central load.
1340-1370	Chooses random numbers for problem.
1380-1400	Writes the problem.
1410-1440	Draws and labels the beam.
1450	Asks for A and B from student.
1460-1540	Compares student's answers with calculated solution and prints appropriate remark.
1550-1580	Has another problem.
1590-1610	Asks if student wants more problems and branches accordingly.
1620-1700	Draws and labels a beam for student's problem.
1720-1790	Prints instructions for second type of beam, concentrated load anywhere.
1800-1870	Draws and labels beam.
1880-1970	Solves the problem.
1980-2050	Chooses a problem and prints it.
2060-2160	Draws and labels the beam.
2170-2190	Solves the problem.
2210-2240	Compares input answers with calculated solution.
2250-2270	If student is incorrect, solves the problem in detail.
2280-2290	Returns for another problem.
2300-2330	Solution was correct. If it is the second problem, another problem is chosen.
2340-2360	Offers the student the choice for more problems.
2370-2530	Solves a problem the student enters.
2540-2600	Prints the general problem for a uniform load.

```
280 FOR DELAY=1 TO 75
290 NEXT DELAY
300 CALL COLOR(2,5,6)
310 FOR DELAY=1 TO 75
320 NEXT DELAY
330 NEXT E
340 CALL CLEAR
350 CALL COLOR(2,2,1)
360 CALL COLOR(9,2,1)
370 CALL COLOR(12,11,1)
380 CALL CHAR(120,"0000000000000000")
390 CALL CHAR(121,"FFFFFFFFFFFFFFFF")
400 CALL CHAR(122,"0F0F0F0F0F0F0F0F")
410 CALL CHAR(99,"18242424242424FF")
420 CALL CHAR(100,"18244281814224FF")
430 CALL CHAR(101,"0F0912244890")
440 CALL CHAR(102,"FF24499224")
450 CALL CHAR(103,"F8902040")
460 CALL CHAR(104,"1010383854549292")
470 CALL CHAR(105,"1010101010101010")
```

```
480 CALL CHAR(106,"92925454383810")
490 CALL CHAR(112,"FFFFFFFFFFFFFFFF")
500 CALL CHAR(113,"F0F0F0F0F0F0F0F0")
510 CALL CHAR(114,"0F0F0F0F0F0F0F0F")
520 CALL CHAR(115,"FF")
530 CALL CHAR(98,"FF60180608304 0FF")
540 CALL COLOR(11,6,1)
550 PRINT TAB(8);"SIMPLE BEAM":::
560 PRINT TAB(5);"SUPPORTED AT ENDS":
    :::::::::
570 I=17
580 GOSUB 5380
590 I=12
600 GOSUB 5560
610 CALL HCHAR(I-5,12,80)
620 FOR L=I-3 TO I-1
630 CALL HCHAR(L,17,112,10)
640 NEXT L
650 CALL VCHAR(I-3,27,113,3)
660 CALL HCHAR(I-2,19,87)
670 PRINT "  FIND THE REACTION FORCES"
680 GOSUB 5530
690 GOTO 5000
700 CALL CLEAR
710 PRINT "NEWTON'S LAWS"
720 PRINT "ARE NECESSARY TO"
730 PRINT "SOLVE REACTION PROBLEMS.":
    :
740 PRINT "1. EQUILIBRIUM OF FORCES"
750 PRINT "          SUM OF FORCES = 0"
760 PRINT "          SUM OF MOMENTS = 0"
770 PRINT ::"2. FORCES ALWAYS OCCUR IN"
780 PRINT "     PAIRS  OF EQUAL AND"
790 PRINT "     OPPOSITE FORCES;"
800 PRINT "     ACTION = REACTION":::
810 GOSUB 5590
820 CALL CLEAR
830 PRINT "TO SOLVE A PROBLEM:":::
840 PRINT "DRAW AND LABEL THE PROBLEM.
    "
850 PRINT ::"WITH 2 UNKNOWN REACTIONS,"
860 PRINT "SOLVE 2 EQUATIONS:":::
870 PRINT "     SUM OF MOMENTS = 0"
880 PRINT "     SUM OF FORCES = 0"
890 PRINT ::"USE CORRECT UNITS.":::
900 GOSUB 5590
910 IF CHOICE=2 THEN 1720
920 CALL CLEAR
930 PRINT "PROBLEM:"
940 PRINT :::"GIVEN A SIMPLE BEAM"
950 PRINT "SUPPORTED AT THE ENDS."
960 PRINT ::"IT IS LENGTH L."
970 PRINT ::"A CONCENTRATED LOAD P"
980 PRINT "IS AT THE CENTER."
990 PRINT ::"IGNORE WEIGHT OF THE BEAM.
    "
1000 GOSUB 5810
1010 I=16
1020 GOSUB 5370
1030 CALL HCHAR(I+1,12,76)
1040 I=16
1050 GOSUB 5560
1060 CALL HCHAR(I-5,16,80)
1070 GOSUB 6040
1080 PRINT "TAKING MOMENTS AT A,":
1090 PRINT "P*L/2 - B*L = 0"
1100 PRINT "          B*L = P*L/2"
1110 PRINT TAB(11);"B = P/2":
1120 PRINT ::"NOW TAKE SUM OF FORCES=0"
1130 GOSUB 5590
1140 PRINT "A + B - P = 0"
1150 PRINT "     A + B = P"
1160 PRINT "          A = P - B = P - P/2
     "
1170 PRINT TAB(9);"A = P/2"
1180 GOSUB 5590
1190 GOSUB 5370
1200 CALL HCHAR(I+1,12,76)
1210 GOSUB 5560
1220 CALL HCHAR(11,16,80)
1230 FOR Y=4 TO 26 STEP 22
1240 CALL HCHAR(22,Y,80)
1250 CALL HCHAR(22,Y+1,47)
1260 CALL HCHAR(22,Y+2,50)
1270 NEXT Y
1280 PRINT "IF THE CONCENTRATED LOAD"
1290 PRINT "IS IN THE CENTER,"
1300 PRINT "A = B = P/2":::
1310 PRINT "FOR EXAMPLE, IF P=1000 LBS.
     "
1320 PRINT "     A=500 LBS. AND B=500 LBS
     "
1330 GOSUB 5590
1340 RANDOMIZE
1350 EX=2
1360 PP=100*(INT(20*RND)+1)
1370 LL=INT(6*RND)+10
1380 GOSUB 5630
1390 GOSUB 5690
1400 GOSUB 5810
1410 GOSUB 5370
1420 GOSUB 5560
1430 GOSUB 5850
1440 GOSUB 5950
1450 GOSUB 6250
1460 IF A<>PP/2 THEN 1510
1470 IF B<>PP/2 THEN 1510
1480 PRINT :::"*** CORRECT!! ***"
1490 GOSUB 5590
1500 GOTO 1560
1510 PRINT :"SORRY, THE REACTIONS ARE"
1520 PRINT :"A =";PP/2
1530 PRINT "B =";PP/2
1540 GOSUB 5590
1550 GOTO 1570
1560 IF EX>2 THEN 1590
1570 EX=EX+1
1580 GOTO 1360
1590 GOSUB 6290
1600 IF KEY=49 THEN 1360
1610 IF KEY=51 THEN 5000
1620 I=16
1630 J=16
1640 GOSUB 5370
1650 GOSUB 5560
1660 CALL HCHAR(I+1,12,76)
1670 CALL HCHAR(I-5,16,80)
1680 INPUT "LENGTH OF BEAM = ":LL
1690 GOSUB 5310
1700 INPUT "LOAD P = ":PP
1710 GOTO 1410
1720 CALL CLEAR
1730 PRINT "GIVEN A BEAM OF LENGTH L"
1740 PRINT "SUPPORTED AT ENDS A AND B."
1750 PRINT "A CONCENTRATED FORCE OF"
1760 PRINT "P POUNDS IS APPLIED"
1770 PRINT "D FEET FROM A. IGNORE"
1780 PRINT "THE WEIGHT OF THE BEAM.":::
1790 GOSUB 5810
1800 I=16
1810 J=12
1820 D=5
1830 GOSUB 5370
1840 GOSUB 5560
1850 CALL HCHAR(I-5,J,80)
1860 CALL HCHAR(I-1,9,68)
1870 CALL HCHAR(I+1,16,76)
1880 GOSUB 6040
1890 PRINT :"TAKING MOMENTS AT A"
1900 PRINT :"P*D - B*L = 0"
1910 PRINT TAB(7);"B*L = P*D"
1920 PRINT TAB(9);"B = P*D/L"
1930 PRINT :"NEXT SUM FORCES = 0"
1940 GOSUB 5590
1950 PRINT :"A + B - P = 0"
1960 PRINT "          A = P-B = P - P*D/L
     "
```

```
1970 GOSUB 5590
1980 RANDOMIZE
1990 EX=2
2000 PP=100*(INT(20*RND)+1)
2010 LL=INT(6*RND)+10
2020 D=INT(10*RND)+1
2030 GOSUB 5630
2040 GOSUB 5690
2050 GOSUB 5810
2060 GOSUB 5370
2070 J=INT(D/LL*21)+5
2080 GOSUB 5560
2090 GOSUB 5850
2100 GOSUB 5950
2110 DD$=STR$(D)
2120 FOR E=1 TO LEN(DD$)
2130 EE=J-5
2140 CALL HCHAR(I-1,EE+E,ASC(SEG$(DD$,E
     ,1)))
2150 NEXT E
2160 CALL HCHAR(I-1,EE+E,39)
2170 BB=PP*D/LL+.005
2180 BB=1E-2*(INT(BB*1E2))
2190 AA=PP-BB
2200 IF EX=2 THEN 2250
2210 GOSUB 6250
2220 IF AA<>A THEN 2240
2230 IF BB=B THEN 2300
2240 PRINT :"OUR ANSWERS DON'T AGREE.":
     :
2250 GOSUB 6040
2260 GOSUB 6110
2270 GOSUB 6190
2280 EX=3
2290 GOTO 2000
2300 PRINT :::"** YOU ARE CORRECT **"
2310 GOSUB 5590
2320 EX=EX+1
2330 IF EX<4 THEN 2000
2340 GOSUB 6290
2350 IF KEY=49 THEN 2000
2360 IF KEY=51 THEN 5000
2370 CALL CLEAR
2380 I=16
2390 J=12
2400 GOSUB 5370
2410 GOSUB 5560
2420 CALL HCHAR(I-5,J,80)
2430 CALL HCHAR(I-1,9,68)
2440 CALL HCHAR(I+1,16,76)
2450 INPUT "LENGTH OF BEAM = ":LL
2460 GOSUB 5310
2470 INPUT "LOAD P = ":PP
2480 INPUT "DISTANCE FROM A = ":D
2490 IF D>LL THEN 2510
2500 IF D>=0 THEN 2530
2510 PRINT :"SORRY, 0 <= D <= L":
     :
2520 GOTO 2480
2530 GOTO 2060
2540 CALL CLEAR
2550 PRINT "GIVEN A SIMPLE BEAM"
2560 PRINT "SUPPORTED AT THE ENDS."
2570 PRINT "IT IS LENGTH L."
2580 PRINT "THERE IS A UNIFORM LOAD"
2590 PRINT "OF W POUNDS PER FOOT."
2600 GOSUB 5810
2610 CALL CLEAR
2620 PRINT "A UNIFORM LOAD CAN BE"
2630 PRINT "THOUGHT OF AS AN"
2640 PRINT "EQUIVALENT RESULTANT"
2650 PRINT "LOAD ACTING AT THE"
2660 PRINT "CENTROID OF THE LOADING":
     :::::::
2670 CALL HCHAR(21,5,112,7)
2680 CALL HCHAR(20,7,87)
2690 CALL HCHAR(20,8,47)
2700 CALL HCHAR(20,9,76)
2710 CALL HCHAR(22,8,76)
2720 CALL HCHAR(21,15,61)
2730 CALL HCHAR(22,19,115,7)
2740 I=22
2750 J=22
2760 GOSUB 5560
2770 CALL HCHAR(18,21,87)
2780 CALL HCHAR(21,23,76)
2790 CALL HCHAR(21,24,47)
2800 CALL HCHAR(21,25,50)
2810 GOSUB 5590
2820 GOSUB 6420
2830 PRINT "EQUIVALENT LOAD IS"
2840 PRINT "W*L ACTING AT CENTER."
2850 PRINT "SOLVING, A=B=W*L/2"
2860 GOSUB 5590
2870 EX=2
2880 RANDOMIZE
2890 WW=10*(INT(10*RND))+1)
2900 LL=INT(10*RND)+10
2910 GOSUB 5630
2920 GOSUB 5780
2930 GOSUB 5810
2940 GOSUB 6420
2950 GOSUB 5950
2960 GOSUB 6500
2970 GOSUB 6250
2980 AA=WW*LL/2
2990 BB=AA
3000 IF AA<>A THEN 3050
3010 IF BB<>B THEN 3050
3020 PRINT :"** CORRECT **"
3030 GOSUB 5590
3040 GOTO 3100
3050 PRINT :"A=B=W*L/2"
3060 PRINT "A=B=";AA;"POUNDS"
3070 GOSUB 5590
3080 EX=EX+1
3090 GOTO 2890
3100 I=18
3110 Y=16
3120 Z=5
3130 EX=EX+1
3140 GOSUB 6450
3150 PRINT "L = 16 FEET"
3160 PRINT "W = 80 LBS/FT"
3170 PRINT "     ACTING 8 FT FROM A"::
3180 PRINT "     TO 12 FT FROM A"::
3190 PRINT "EQUIVALENT FORCE IS"
3200 PRINT "80 LBS/FT * (12 FT - 8 FT)"
3210 PRINT "     = 320 LBS"
3220 PRINT "APPLIED 10' FROM A."
3230 PRINT :"SUM MOMENTS ABOUT A."
3240 GOSUB 5590
3250 GOSUB 5370
3260 J=19
3270 GOSUB 5560
3280 LL=16
3290 PP=320
3300 GOSUB 5850
3310 GOSUB 5950
3320 CALL HCHAR(I-1,10,49)
3330 CALL HCHAR(I-1,11,48)
3340 CALL HCHAR(I-1,12,39)
3350 PRINT "320# * 10' - B*16' = 0"
3360 PRINT TAB(5);"B=3200/16 LBS = 200
     LBS":::
3370 PRINT "NOW SUM FORCES"
3380 GOSUB 5590
3390 PRINT :"320# - 200# - A = 0"
3400 PRINT "A = 120 LBS"
3410 GOSUB 5590
3420 EX=EX+1
3430 RANDOMIZE
3440 GOSUB 6660
3450 GOSUB 6610
3460 GOSUB 6730
3470 GOSUB 5590
3480 GOSUB 5370
3490 GOSUB 6370
3500 GOSUB 6500
```

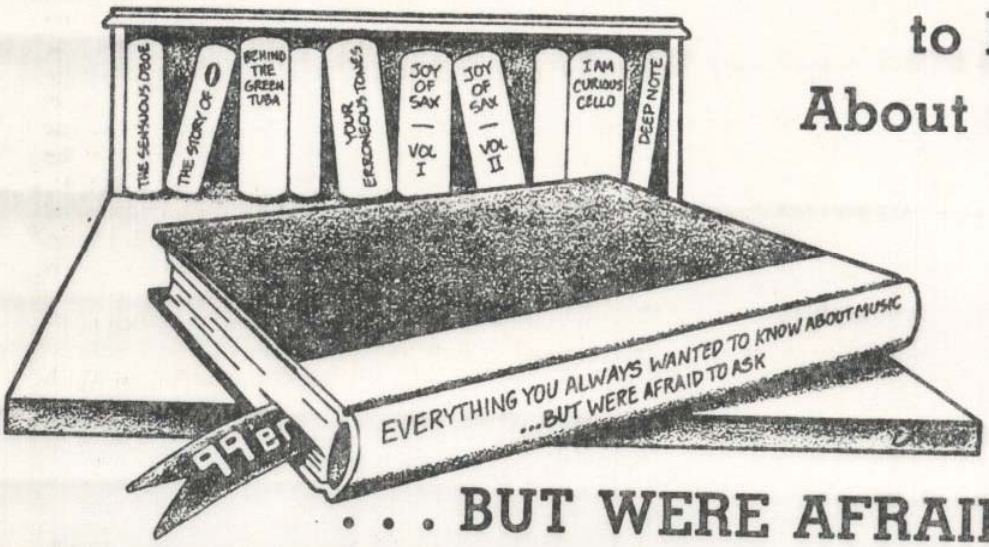
```
3510 GOSUB 5950
3520 GOSUB 6250
3530 BB=LOAD*((D2/2+L1)/LL+.005
3540 BB=1E-2*(INT(BB*1E2))
3550 AA=LOAD-BB
3560 IF AA<>A THEN 3610
3570 IF BB<>B THEN 3610
3580 PRINT :"** CORRECT **"
3590 GOSUB 5590
3600 GOTO 3660
3610 PRINT :"SORRY. IT IS "
3620 PRINT "A = ";AA
3630 PRINT "B = ";BB:
3640 GOSUB 5590
3650 GOTO 3420
3660 GOSUB 6290
3670 IF KEY=49 THEN 3420
3680 IF KEY=51 THEN 5000
3690 CALL CLEAR
3700 I=16
3710 Y=16
3720 Z=5
3730 GOSUB 6450
3740 INPUT "LENGTH OF BEAM IN FT = ":LL
3750 GOSUB 5310
3760 INPUT "LOADING W LB/FT = ":WW
3770 IF WW<>0 THEN 3800
3780 PRINT "IF W=0, A=B=0"
3790 GOTO 3760
3800 INPUT "ACTING AT DISTANCE FROM A "
     :L1
3810 IF L1<0 THEN 3830
3820 IF L1<LL THEN 3850
3830 PRINT "SORRY, 0 <= L1 < LL"
3840 GOTO 3800
3850 INPUT "TO - DISTANCE FROM A ":L2
3860 IF L2<=L1 THEN 3880
3870 IF L2<=LL THEN 3900
3880 PRINT "SORRY, L1 < L2 <= L"
3890 GOTO 3850
3900 GOSUB 6730
3910 GOTO 3480
3920 CALL CLEAR
3930 PRINT TAB(4);"COMBINATION LOADS":
     :
3940 PRINT :"TO SOLVE THIS TYPE PROBLEM
     :"
3950 PRINT :"DRAW AND LABEL THE BEAM."
3960 PRINT :"SUM MOMENTS ABOUT A OR B."
3970 PRINT :"SUM FORCES": :
3980 GOSUB 5590
3990 CALL CLEAR
4000 RANDOMIZE
4010 GOSUB 6660
4020 GOSUB 6730
4030 PP=100*(INT(15*RND))
4040 D=INT(LL*RND)
4050 GOSUB 5650
4060 IF PP=0 THEN 4080
4070 GOSUB 5710
4080 IF WW=0 THEN 4130
4090 GOSUB 5780
4100 IF L1=LL THEN 4120
4110 GOSUB 6630
4120 GOSUB 5810
4130 I=16
4140 J=INT(D/LL*22)+5
4150 GOSUB 5370
4160 GOSUB 5950
4170 IF WW=0 THEN 4200
4180 GOSUB 6370
4190 GOSUB 6500
4200 IF PP=0 THEN 4230
4210 GOSUB 5560
4220 GOSUB 5850
4230 GOSUB 6250
4240 BB=(PP*D+LOAD*((D2/2+L1)))/LL+.005
4250 BB=1E-2*(INT(BB*1E2))
4260 AA=PP+LOAD-BB
4270 IF ABS(AA-A)>.01 THEN 4340
4280 IF ABS(BB-B)>.01 THEN 4340
4290 PRINT : :"** CORRECT **"
4300 GOSUB 5590
4310 GOSUB 6290
4320 IF KEY=49 THEN 3990
4330 IF KEY=51 THEN 5000 ELSE 4390
4340 PRINT : :"SORRY, THE ANSWER I GET
     IS"
4350 PRINT :"A = ";AA
4360 PRINT "B = ";BB
4370 GOSUB 5590
4380 GOTO 3990
4390 CALL CLEAR
4400 PRINT "YOU MAY ENTER YOUR OWN"
4410 PRINT :"PROBLEMS. JUST ENTER"
4420 PRINT :"ALL VARIABLES AS "
4430 PRINT :"SHOWN ON THE DIAGRAM."
4440 PRINT :: :"I WILL GIVE THE ANSWERS.
     ": : : : :
4450 GOSUB 5590
4460 I=16
4470 J=12
4480 L1=0
4490 L2=0
4500 D2=0
4510 LOAD=0
4520 GOSUB 5370
4530 GOSUB 5560
4540 CALL HCHAR(I-5,J,30)
4550 Y=16
4560 Z=5
4570 GOSUB 6370
4580 CALL HCHAR(I-3,Y+2,87)
4590 CALL HCHAR(I+2,13,76)
4600 CALL HCHAR(I+1,Y,76)
4610 CALL HCHAR(I+1,Y+1,49)
4620 CALL HCHAR(I+1,Y+Z-1,76)
4630 CALL HCHAR(I+1,Y+Z,50)
4640 CALL HCHAR(I-1,8,68)
4650 INPUT "LENGTH OF BEAM L = ":LL
4660 GOSUB 5310
4670 INPUT "FORCE P = ":PP
4680 IF PP=0 THEN 4740
4690 INPUT "DISTANCE D = ":D
4700 IF D>LL THEN 4720
4710 IF D>=0 THEN 4740
4720 PRINT :"SORRY, 0 <= D <= L":
4730 GOTO 4690
4740 INPUT "LOADING W = ":WW
4750 IF WW=0 THEN 4880
4760 INPUT "DISTANCE FROM A,L1 = ":L1
4770 IF L1<0 THEN 4790
4780 IF L1<LL THEN 4810
4790 PRINT "SORRY, 0 <= L1 <L"
4800 GOTO 4760
4810 INPUT "DISTANCE FROM A,L2 = ":L2
4820 IF L2<=L1 THEN 4840
4830 IF L2<=LL THEN 4860
4840 PRINT "SORRY, L1 < L2 <= LL"
4850 GOTO 4810
4860 D2=L2-L1
4870 LOAD=WW*D2
4880 BB=(PP*D+LOAD*((D2/2+L1)))/LL+.005
4890 BB=1E-2*(INT(BB*1E2))
4900 AA=PP+LOAD-BB
4910 PRINT : :"A = ";AA;"POUNDS"
4920 PRINT "B = ";BB;"POUNDS"
4930 GOSUB 5590
4940 PRINT :"ANOTHER PROBLEM?(Y/N) "
4950 CALL KEY(0,KEY,ST)
4960 IF KEY=89 THEN 4480
4970 IF KEY=78 THEN 5000
4980 GOTO 4950
4990 END
5000 CALL CLEAR
5010 CALL SCREEN(5)
5020 FOR E=1 TO 8
5030 CALL COLOR(E,12,12)
```

```
5040 NEXT E
5050 PRINT :::::::::::
5060 PRINT "SELECT":::
5070 PRINT :"1 CONCENTRATED LOAD, CENTE
     R"
5080 PRINT :"2 CONCENTRATED LOAD ANYWHE
     RE"
5090 PRINT :"3 UNIFORM LOADS"
5100 PRINT :"4 COMBINATION LOADS"
5110 PRINT :"5 PROBLEMS ONLY"
5120 PRINT :"6 YOUR OWN PROBLEMS"
5130 PRINT :"7 END PROGRAM"::::
5140 CALL VCHAR(1,2,32,24)
5150 CALL VCHAR(1,31,32,24)
5160 CALL VCHAR(1,1,32,24)
5170 CALL VCHAR(1,32,32,24)
5180 FOR E=1 TO 8
5190 CALL COLOR(E,2,12)
5200 NEXT E
5210 CALL KEY(0,KEY,ST)
5220 CHOICE=KEY-48
5230 IF CHOICE<1 THEN 5210
5240 IF CHOICE>7 THEN 5210
5250 CALL CLEAR
5260 CALL SCREEN(8)
5270 FOR E=1 TO 8
5280 CALL COLOR(E,2,1)
5290 NEXT E
5300 ON CHOICE GOTO 700,700,2540,3920,3
     990,4390,4990
5310 IF LL>=1 THEN 5360
5320 PRINT "HEY, WHAT KIND OF BEAM"
5330 PRINT "HAS A LENGTH LIKE THAT?!!"
5340 INPUT "TRY AGAIN; L = ":LL
5350 GOTO 5310
5360 RETURN
5370 CALL CLEAR
5380 CALL HCHAR(1,5,120)
5390 CALL HCHAR(1,6,121,21)
5400 CALL HCHAR(1,27,122)
5410 CALL HCHAR(I+1,5,99)
5420 CALL HCHAR(I+1,27,100)
5430 FOR K=4 TO 26 STEP 22
5440 CALL HCHAR(I+2,K,101)
5450 CALL HCHAR(I+2,K+1,102)
5460 CALL HCHAR(I+2,K+2,103)
5470 CALL VCHAR(I+3,K+1,104)
5480 CALL VCHAR(I+4,K+1,105,2)
5490 NEXT K
5500 CALL HCHAR(I+1,4,65)
5510 CALL HCHAR(I+1,28,66)
5520 RETURN
5530 FOR DELAY=1 TO 1000
5540 NEXT DELAY
5550 RETURN
5560 CALL VCHAR(I-4,J,105,3)
5570 CALL VCHAR(I-1,J,106)
5580 RETURN
5590 PRINT :"PRESS ENTER TO CONTINUE"
5600 CALL KEY(0,KEY,ST)
5610 IF KEY<>13 THEN 5600
5620 RETURN
5630 CALL CLEAR
5640 PRINT "PROBLEM":::
5650 PRINT "GIVEN A SIMPLE BEAM"
5660 PRINT "SUPPORTED AT THE ENDS."
5670 PRINT "IT IS";LL;"FEET LONG."
5680 RETURN
5690 IF CHOICE>2 THEN 5710
5700 PRINT :"IGNORE WEIGHT OF THE BEAM.
     "
5710 PRINT "A CONCENTRATED LOAD OF"
5720 PRINT PP;"POUNDS IS"
5730 IF CHOICE=1 THEN 5760
5740 PRINT D;"FEET FROM END A."
5750 RETURN
5760 PRINT "AT THE CENTER OF THE BEAM."
5770 RETURN
5780 PRINT :"THERE IS A UNIFORM LOAD"
5790 PRINT "OF";WW;"POUNDS/FOOT"
5800 RETURN
5810 PRINT :"FIND THE REACTION FORCES."
     ::
5820 PRINT "DRAW AND LABEL THE PROBLEM.
     ":::
5830 GOSUB 5590
5840 RETURN
5850 LB$=STR$(PP)
5860 FOR II=1 TO LEN(LB$)
5870 JJ=II+J-4
5880 CALL HCHAR(I-5,JJ,ASC(SEG$(LB$,II,
     1)))
5890 NEXT II
5900 CALL HCHAR(I-5,JJ+1,32)
5910 CALL HCHAR(I-5,JJ+2,76)
5920 CALL HCHAR(I-5,JJ+3,66)
5930 CALL HCHAR(I-5,JJ+4,83)
5940 RETURN
5950 FT$=STR$(LL)
5960 FOR II=1 TO LEN(FT$)
5970 JJ=12+II
5980 CALL HCHAR(I+1,JJ,ASC(SEG$(FT$,II,
     1)))
5990 NEXT II
6000 CALL HCHAR(I+1,JJ+2,70)
6010 CALL HCHAR(I+1,JJ+3,69,2)
6020 CALL HCHAR(I+1,JJ+5,84)
6030 RETURN
6040 CALL HCHAR(23,3,98)
6050 CALL HCHAR(23,4,77)
6060 CALL HCHAR(23,6,61)
6070 CALL HCHAR(23,8,48)
6080 PRINT "WRITE THE EQUATION"
6090 GOSUB 5590
6100 RETURN
6110 PRINT :"TAKING MOMENTS AT A."
6120 PRINT :"+D - B*L = 0"
6130 PRINT "                B = P * D/L"
6140 PRINT "                B =";PP;"*";D;"/";L
     L
6150 PRINT "                B =";BB;"POUNDS"
6160 PRINT :"NOW FIND A."
6170 GOSUB 5590
6180 RETURN
6190 PRINT :"SUM OF FORCES = 0"
6200 PRINT "+P-A-B = 0"
6210 PRINT "          A = P-B =";PP;"-";BB
6220 PRINT "          A =";AA;"POUNDS"
6230 GOSUB 5590
6240 RETURN
6250 PRINT :"WHAT ARE A AND B IN POUNDS
     ?"
6260 INPUT "A =":A
6270 INPUT "B =":B
6280 RETURN
6290 PRINT :::"DO YOU WANT MORE PROBLEM
     S?"
6300 PRINT :"1 YES, SAME KIND"
6310 PRINT "2 YES, MY OWN PROBLEMS"
6320 PRINT "3 NO, DO SOMETHING ELSE"
6330 CALL KEY(0,KEY,ST)
6340 IF KEY<49 THEN 6330
6350 IF KEY>51 THEN 6330
6360 RETURN
6370 CALL HCHAR(I-1,Y,112,Z)
6380 CALL HCHAR(I-2,Y,112,Z)
6390 CALL VCHAR(I-2,Y-1,114,2)
6400 CALL VCHAR(I-2,Y+Z,113,2)
6410 RETURN
6420 I=16
6430 Y=6
6440 Z=21
6450 GOSUB 5370
6460 GOSUB 6370
6470 CALL HCHAR(I-3,16,87)
6480 CALL HCHAR(I+1,16,76)
```

```
6490 RETURN
6500 X=INT(Y+Z/2-3)
6510 UL$=STR$(WW)
6520 FOR E=1 TO LEN(UL$)
6530 CALL HCHAR(I-3,X+E-1,ASC(SEG$(UL$,
     E,1)))
6540 NEXT E
6550 CALL HCHAR(I-3,X+E,76)
6560 CALL HCHAR(I-3,X+E+1,66)
6570 CALL HCHAR(I-3,X+E+2,47)
6580 CALL HCHAR(I-3,X+E+3,70)
6590 CALL HCHAR(I-3,X+E+4,84)
6600 RETURN
6610 GOSUB 5630
6620 GOSUB 5780
6630 PRINT "ACTING FROM";L1;"FEET FROM
     A"
```

```
6640 PRINT "TO";L2;"FEET FROM A"
6650 RETURN
6660 LL=INT(8*RND)+12
6670 WW=10*(INT(4*RND)+5)
6680 LIM1=INT(3*LL/4)
6690 L1=INT(LIM1*RND)
6700 LIM2=LL-L1
6710 L2=INT(LIM2*RND)+L1+1
6720 RETURN
6730 D2=L2-L1
6740 LOAD=WW*D2
6750 Y=INT(L1/LL*22)+6
6760 Z=INT(D2/LL*22)-1
6770 RETURN
```

Almost Everything You Ever Wanted to Know About Music . . .



. . . BUT WERE AFRAID TO ASK

Is music terminology Greek to you? Do you feel deficient in certain areas of your musical ability? How are your listening skills? If you enjoy music and want to test and improve your abilities, TI's *Music Skills Trainer* can be a valuable tool. This program provides practice in aural recognition of pitches, intervals, and chords, and develops your ability to remember musical phrases. You can control the complexity of each drill by selecting various options including note range, use of sharps (♯) and flats (♭), types of chords and intervals, and the playing of random music between examples.

Since the program is designed to provide drill and does not teach the underlying concepts involved, this article will first cover relevant aspects of music theory. We'll then follow it up with a review of *Music Skills Trainer*.

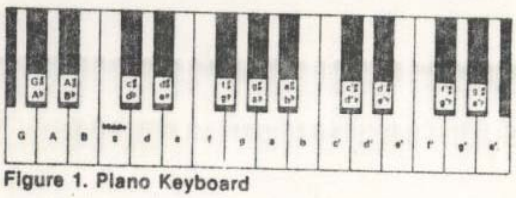


Figure 1. Piano Keyboard

The Scale

The fundamental concept involved is that of the *scale*—an ordered group of tones within an octave. The C Major scale, with which almost everyone is familiar, provides the standard pattern for every major scale (Do-Re-Mi-Fa-Sol-La-Ti-Do). This pattern originated with the Greeks and is based upon the *tetrachord*. A tetrachord can be thought of as half a scale; it consists of four tones arranged so that they contain two whole steps followed by a half step. Refer to the diagram of a piano keyboard in Figure 1. Starting at middle **c**, each progression up the keyboard represents a half step or semitone. For example, all the following represent half steps: **c-c♯**, **c♯-d**, **d-d♯**, **d♯-e**, **e-f**, etc. The first tetrachord for the C Major scale consists of the following two whole steps: **c-d** and **d-e** followed by the natural half step **e-f**. The second tetrachord begins with **g** and again consists of two whole steps followed by a half step, ending with **c'** (an octave above middle **c**). This tetrachord pattern (1 + 1 + ½) was referred to as a *diatonic scale*.

In order to accommodate Oriental and other music, Greek theorists modified the two middle tones of the diatonic tetrachord in several ways. One of these, called the *chromatic tetrachord*, consisted of the pattern 1½ + ½ + ½ (e.g., **c, d♯, e, f,**). Various combinations of these two tetrachords necessitate the division of an octave into the familiar twelve equally spaced intervals referred to as the chromatic scale: **c, c♯, d, d♯, e, f, f♯, g, g♯, a, a♯, b, c'**.

Pitch refers to the location of one of these tones in a scale, and is defined by a regular frequency of vibrations. In the United States the standard assignment for **a** above middle **c** is 440 vibrations per second. It happens that a pure octave differs from any reference pitch by a factor of exactly 2, so that **a** two octaves above middle **c** = 880 and **A** below middle **c** = 220.

Although knowledge of frequencies is not required for use of the *Music Skills Trainer*, you may be interested to know how frequencies are assigned to other scale positions. Because each octave is divided into twelve equally-spaced intervals, the factor $2^{\frac{1}{12}}$ is used to define the relative frequencies of successive tones. For example,

$$\text{If } \mathbf{a} = 440;$$
$$\mathbf{a\sharp} = 440 \times 2^{\frac{1}{12}},$$
$$\mathbf{b} = \mathbf{a\sharp} \times 2^{\frac{1}{12}} = \mathbf{a} \times 2^{\frac{1}{12}} \times 2^{\frac{1}{12}} = \mathbf{a} \times (2^{\frac{1}{12}})^2.$$

Given any reference frequency, f_0, then the relative pitch of any other scale position, f, can be calculated by counting the number of half steps to that position, N, and using the formula:

$$f = f_0(2^{\frac{1}{12}})^N.$$

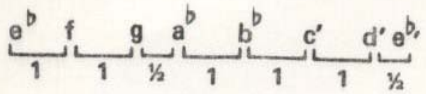
The following program calculates and plays a chromatic scale beginning with middle **c** (262).

```
100 REM
110 REM  MUSIC 1
120 REM
130 REM
```

```
140 REM
150 REM
160 REM
170 F0=262
180 FOR N=0 TO 12
190 F=F0*(2^(1/12))^N
200 CALL SOUND(-600,F,0)
210 NEXT N
220 STOP
```

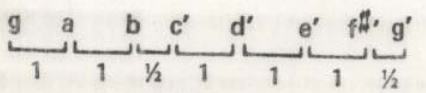
Scales in Various Keys

Now let us return to the diatonic (major) scale. A major scale can have a starting or *root* note of any of the twelve chromatic pitches. As in the case discussed above, a major scale is constructed, starting from the root, with two diatonic tetrachords (1 + 1 + ½) separated by a whole step. A more convenient way to construct a major scale is simply to remember that half steps occur between the third and fourth and the seventh and eighth tones. Referring to Figure 1, a major scale with $e^{\flat}$ as the root would be constructed using the following steps:



Steps

This scale is referred to as an $E^{\flat}$ Major scale, or a scale in the Key of $E^{\flat}$, since $e^{\flat}$ is the root. Similarly, a major scale in the key of G is constructed as follows:



Steps

Steps

While there are twelve such different diatonic scales, they all sound the same because they are based on the same pattern of diatonic steps. The following program plays these scales beginning with C Major.

```
100 REM  ********** 
110 REM  * MUSIC 2 *
120 REM  ********** 
130 REM
140 REM
150 REM
160 REM
170 F0=262
180 FOR I=0 TO 11
190 F=F0*(2^(1/12))^I
200 FOR J=1 TO 8
210 N=2
220 IF J=4 THEN 240
230 IF J=8 THEN 240 ELSE 250
240 N=1
250 F=F*(2^(1/12))^N
260 CALL SOUND(-600,F,0)
270 NEXT J
280 NEXT I
290 STOP
```

Intervals

An *interval* is the difference in pitch between two notes. Interval names indicate the number of included tones of the major scale. Starting with middle c in Figure 1, the basic interval names are as follows: **c-c**, unison (prime); **c-d**, second; **c-e**, third; **c-f**, fourth; **c-g**, fifth; **c-a**, sixth; **c-b**, seventh; and **c-c'**, octave. **c-f** is a fourth because it includes the following diatonic tones of the C Major scale: **c**, **d**, **e**, and **f**. Similarly in the $E^{\flat}$ Major scale, a fourth is $e^{\flat}$-$a^{\flat}$, and in the G Major scale a fourth is **g-c'**. However, as in the case of scales, an interval in one key sounds like that interval in another.

Four of the eight intervals can exist in one of four forms. If the upper note of the interval lies within the major scale of the lower or root note, the interval may be classified as *major*. If the upper note is lowered a half step, however, the interval then becomes *minor*. For example, **c-e** is a major third and **c-$e^{\flat}$** a minor third. This rule applies to four intervals; the second, third, sixth, and seventh. The remaining intervals—fourth, fifth, and octave—are classified as *perfect*: They do not exist in major and minor forms. The following program plays all of the intervals above in the C Major scale, i.e., with middle **c** as the lower or root note.

The remaining two categories of intervals—*augmented* and *diminished*—are not used in the *TI Music Skills Trainer* and so will not be discussed in detail. They are formed as follows: augmented—a major or perfect interval is made one half step larger; diminished—a minor or perfect interval is made one half step smaller.

Finally, intervals may be classified according to which note is played first. If the lower note is played first, the interval is said to be ascending (**c-e**), and if the upper note is played first, it is descending (**e-c**).

Chords

A chord is several notes played simultaneously, usually three or more. When a chord consists of three tones it is called a *triad*. Given any major scale, four triads can be formed from the starting note (root) of that scale: major, minor, augmented, and diminished. A major triad consists of the root, the third, and the fifth. For example, in a C Major scale, starting with the root **c**, the third is **c-e**, and the fifth is **c-g**. The major chord is then **c-e-g**. Similarly, in the $E^{\flat}$ Major scale, given the root $e^{\flat}$, the third **g**, and the fifth $b^{\flat}$, the major chord is $e^{\flat}$-**g**-$b^{\flat}$.

The major chord is changed to a minor chord by lowering the second note (i.e., the third) one half step. For example, the C Major chord **c-e-g** becomes the **C** minor chord **c-$e^{\flat}$-g** and the $E^{\flat}$ Major chord becomes the $E^{\flat}$ minor chord $e^{\flat}$-$g^{\flat}$-$b^{\flat}$.

A minor chord can further be changed to a diminished chord by lowering the third note (i.e., the fifth) one half step. For example, the C minor chord **c-$e^{\flat}$-g** becomes the c diminished chord **c-$e^{\flat}$-$g^{\flat}$** and the $E^{\flat}$ minor chord $e^{\flat}$-$g^{\flat}$-$b^{\flat}$ becomes the $E^{\flat}$ diminished chord $e^{\flat}$-$g^{\flat}$-$b^{\flat\flat}$. ($b^{\flat\flat}$ is called "**b** double flat" and is the same note as **a**.)

The augmented chord is formed by raising the third note of the major chord (i.e., the fifth) one half step. For example, the C Major chord **c-e-g** becomes the **C** augmented chord **c-e-g**♯ and the $E^{\flat}$ Major chord becomes the $E^{\flat}$ augmented chord $e^{\flat}$-**g**-**b**.

As in the case of scales and intervals, chords with the same name sound alike. All major chords sound alike; all minor chords sound alike, etc.

If the lowest note of the chord is the root, the chord is said to be in *root position*. All four types of triads (chords), however, can be played in inverted form. For example, the C Major chord **c-e-g** may be altered from its root position form to one of the following inversions by making the lowest note either the third or the fifth: **e-g-c'** and **g-c'-e'**. Similar-

ly, the inverted forms for the E♭ minor—which in root position is written or played e♭-g♭-b♭—are g♭-b♭-e♭' and b♭-e♭'-g♭'.

Chords of more than three notes can be formed, and there are several different varieties. One of them, the seventh, is used in the *Music Skills Trainer*. The seventh chord contains the root, third, fifth, and the seventh lowered by a half step. For example, a seventh in the key of C Major is c-e-g and b lowered by a half step or b♭. Similarly, in the key of e♭ the seventh chord is e♭-g-b♭-d♭ (d lowered by a half step).

While the seventh chord contains four notes, the TI-99/4A can play only three notes simultaneously; therefore, following traditional rules of harmony the fifth of the chord (third note) may be omitted to give a seventh in the form of c-e-b♭. As in the case of triads, the seventh may appear in inverted forms.

TI *Music Skills Trainer*

The *Music Skills Trainer* from Texas Instruments is a program written in TI BASIC (it will also run in Extended BASIC without modification). The program is available on cassette or diskette.

Four types of drill are provided: Pitch Guess, Interval Recognition, Chord Recognition, and Phrase Recall. The user selects the type of drill desired from a menu.

Pitch Guess

In this drill, you try to identify the pitch of a single note. While it might seem that this would require perfect pitch, you will find after several examples that you have "tuned in" and are able to identify pitches by relating each new one to the one that has preceded. The difficulty of this exercise can be controlled by specifying the starting note and range size in half steps (up to two octaves). In addition, you can choose to have notes selected from either the C Major diatonic or chromatic scales by answering "No" or "Yes" to the option of including sharps and flats. TI has included yet another means of increasing the level of difficulty—Random Music. If chosen, random music is played between examples, making it more difficult to remember the preceding note. The program provides up to ten examples and keeps score: 10 points for each correct answer.

We recommend that when first using this drill, you use c as the starting note, a range size of 13 (one octave), no sharps and flats, and no random music. After a little practice, it shouldn't be that difficult to identify notes.

Interval Recognition

This drill helps to develop your ability to recognize intervals. There are three levels, each of which adds more intervals to those included in the drill. For instance, if you choose Level 1, the examples are composed of major thirds, fourths, and fifths. Level 2 adds half steps, whole steps, and minor thirds; and Level 3 sixths, sevenths, and octaves. You can choose to have the intervals presented in ascending or descending order. For an added difficulty, you can also choose to have the lower note be random; it is otherwise c each time. You can also choose to have random music play between exercises. Up to ten examples are provided, and you receive 10 points for each correct answer.

Chord Recognition

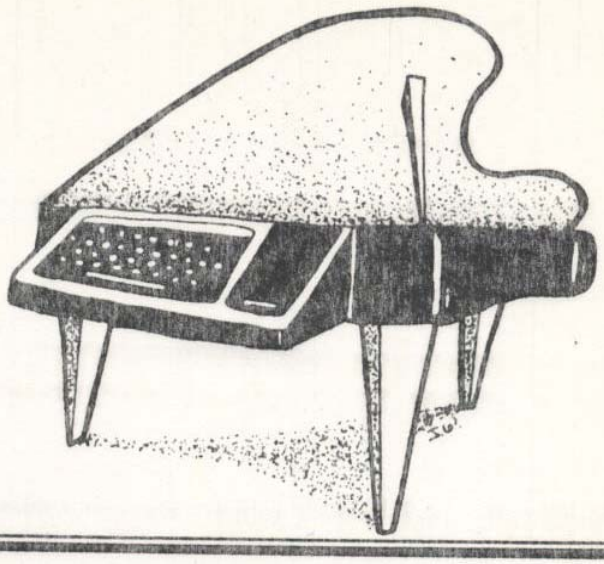
This drill provides practice in recognizing chords. Again there are three levels, with Level 1 consisting of major and minor chords, Level 2 adding seventh and diminished, and Level 3 adding augmented. If you choose the Random Bass option, the root can be any note; otherwise it is a c. If you choose the Random Inversions option, inverted chords will be played; otherwise, a root-position chord is always played. If you choose the Chord Only option, the three notes will be played simultaneously. If you don't choose it, the notes comprising the chord are first played individually and then together. As in the previous drills, you can select the Random Music option. You receive 10 points for each, up to 10 problems.

Phrase Recall

This drill develops your ability to remember a sequence of as many as nine random notes. A blank keyboard overlay, provided with the program, is used to label the keys with their corresponding pitch, covering two octaves much like the layout of a piano keyboard. You can select the starting note and range size, and determine whether sharps and flats are to be included in the examples. You can also specify the number of notes which constitute the phrase (1-9). After a phrase is played, you respond by entering notes from the keyboard as if it were a piano. As you play the notes, you hear them and they are displayed as well; if you make a mistake, you can use SHIFT T to start over again without penalty. When you have entered the notes that you think correctly represent the phrase, you press ENTER. The correct notes are then displayed below your response, and you are awarded points based on the number of correct notes and the number of notes included in the phrase. Up to ten examples are given with a possible total score of up to 100 points. As in the previous drills, the Random Music option can be chosen to make this drill even more difficult.

We feel that TI's *Music Skills Trainer* will be useful even for experienced musicians who want to keep their auditory skills sharp. We would also recommend it for novices interested in further developing their knowledge and abilities in areas of music theory covered by the program.

Let's Learn Notes

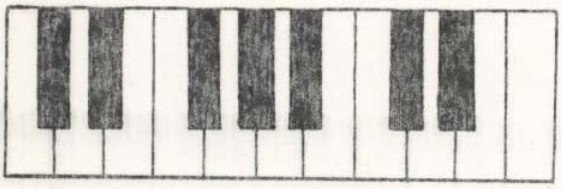


Let's Learn Notes was designed for beginning music students. A piano or organ teacher can use the program during a lesson to give the student a different approach to learning musical notes, or a student can run the program before or after the regular lesson. Students can also use the program at home for additional practice in learning musical notes. Even preschool children can begin learning the notes with this program.

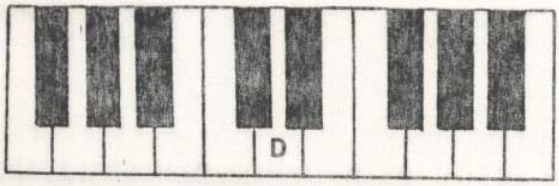
The program is written in TI BASIC and uses color graphics to draw piano keyboards, musical staves, and notes. In addition, the program generates musical tones.

This program provides three options: Keyboard Learning, Treble Clef Learning and Bass Clef Learning. Each option asks for ten responses. An incorrect response is recognized by a slight non-musical noise; the correct response must be entered before the program will continue.

Keyboard Learning randomly selects and displays one of two piano keyboards (starting at the left with either two black keys or three black keys). It then randomly selects one of the 11 displayed piano keys and flashes a question mark on the key. The student responds by pressing the letter on the computer keyboard that corresponds to the letter name of the piano key shown. If the response is correct, the corresponding musical tone is played and the letter name is printed on the piano key. The program randomly chooses Keyboard 1 or Keyboard 2 for each question. If the keyboard chosen is the same as for the previous question, the keyboard is not redrawn.

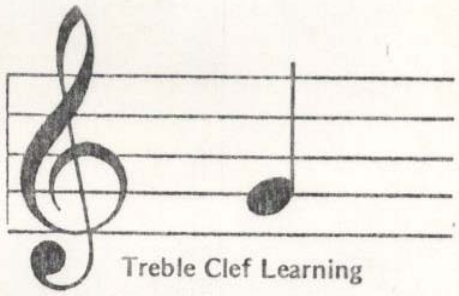


Keyboard 1



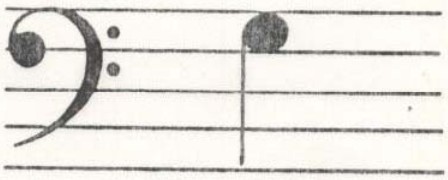
Keyboard 2

Treble Clef Learning displays a staff and treble clef. A note is selected randomly from Middle C to high F (top line of the staff) and displayed as a red quarter note. The student presses the letter on the computer keyboard that corresponds to the letter name of the note. If the response is correct, the corresponding musical tone is played and the letter name is printed on the note.



Treble Clef Learning

Bass Clef Learning displays a staff and bass clef. A note is selected randomly from low G (bottom line of the staff) to Middle C and displayed as a red quarter note. The student presses the letter on the computer keyboard that corresponds to the letter name of the note. If the response is correct, a five-note scale is played and the letter name is printed on the note.



Bass Clef Learning

This program is very easy to use and "student-friendly"—even for the youngest piano learners. A student can select the three learning options either at the beginning of the program or after each option has finished, simply by pressing 1, 2, or 3 on the computer keyboard. If a number greater than 3 is pressed, the program ends.

This program makes repetitious drill much more fun for the piano student and much less boring for the teacher. TI's color graphics and sound in this program greatly enhance the student's motivation to learn the letter names of piano keys and notes.

EXPLANATION OF THE PROGRAM
Let's Learn Notes

Line Nos.

150 T=1500 for the CALL SOUND(T,---,-) statements.

170-450 Defines colors and characters for the title screen.

460-1220 Displays the characters for musical notes and a treble clef for the title screen. Musical tones of the C Major scale and arpeggio are played while the title screen is displayed.

1230 Asks which option the student wants and branches to that option.

1240-1340 Option 1, Keyboard. Defines color and characters for drawing the keyboard.

1360 COUNT set to zero and incremented for each question. There are 10 questions in each option.

1370 Keyboard number is randomly chosen, 1 or 2.

1380-1440 Prints "NAME THE KEY".

1450-1550 Draws the white keys.

1570-1730 Draws the black keys for one of the two piano keyboards.

1740-1750 Chooses one of the 11 keys randomly.

1760-1810 Blinks a red question mark on the key.

1820-1830 Reads the student's response.

1840-2680 Tests the response. If it is incorrect, there is a nonmusical sound and another response is required. If it is correct, the corresponding musical tone is played and the letter name of the key is displayed on the key.

2690-2710 Delays, then erases the letter name.

2720-2730 Increments COUNT and determines if there have been 10 questions.

2740-2750 Chooses keyboard pattern randomly. If it is the same as the previous question, only a new key is chosen; if it is different, a new keyboard is drawn before the key is chosen.

2760 Treble Clef and Bass Clef option.

2770-2800 Resets colors for this screen.

2810-3270 Defines special characters for staff, treble clef, and note.

3280 Draws staff.

3290-3340 Prints "NAME THE NOTE".

3360-3450 Treble Clef option. Draws the treble clef.

3460 Sets COUNT for number of problems.

3470-3530 Chooses note and draws it.

3540-3550 Reads the student's response.

3560-4010 Tests the response. If it is incorrect, there is a nonmusical sound and another response is required. If it is correct, the corresponding musical tone is played and the letter name is displayed on the note.

4020-4030 After a delay, erases the note and chooses a new note. If there have been 10 notes, the options are listed again.

4040-4160 Bass Clef option. Defines special characters for the bass clef.

4170-4230 Prints bass clef.

4240 Sets COUNT = 0 for the number of problems.

4250-4310 Chooses one of 11 notes randomly and draws it.

4320-4330 Reads the student's response.

4340-4900 Tests the response. If it is incorrect, there is a nonmusical sound and another response is required. If it is correct, a five-note scale is played starting at frequency J, and the letter name is displayed on the note.

4910-4920 After a delay, erases the note and chooses a new note. If there have been 10 notes, the options are listed again.

4930-4970 Subroutine for playing the 5-note scale.

4980-5060 Subroutine for drawing the staff.

5070-5210 Subroutine for drawing the note.

5130-5210 Draws the stem of the note up or down from the note, depending on where the note is.

5220-5390 Subroutine for procedure after each note.

5230-5240 Delays

5250-5260 Increments and tests COUNT.

5260-5390 If COUNT < 10, erases the note and returns.

5400-5430 If COUNT = 10, prints menu screen of options.

5540-5490 Branches to Option 1, 2, or 3.

5500 If 4 is pressed, the program ends.

```
100 REM *********************
110 REM *LET'S LEARN NOTES*
120 REM *********************
130 REM
140 REM
150 T=1500
160 CALL CLEAR
170 CALL COLOR(11,13,1)
180 CALL COLOR(12,13,1)
190 CALL COLOR(13,14,1)
200 CALL COLOR(14,5,1)
210 CALL COLOR(15,5,1)
220 CALL CHAR(112,"0000020705050505")
230 CALL CHAR(113,"0505050303030202")
240 CALL CHAR(114,"050D0910302060C0")
250 CALL CHAR(115,"0000000808080809CE3")
260 CALL CHAR(116,"40C1C2C2868686C2")
270 CALL CHAR(117,"C0404040202020C0")
280 CALL CHAR(118,"8080404040404040")
290 CALL CHAR(119,"C1804020100C07")
300 CALL CHAR(120,"02010111121CF010")
310 CALL CHAR(121,"8080808")
320 CALL CHAR(122,"1099990E0")
330 CALL SOUND(T,262,2)
340 CALL CHAR(136,"0000030F1F1F3F3F")
350 CALL CHAR(137,"0202E2FAFAFEFEFE")
360 CALL CHAR(138,"1F1F0F0F03")
370 CALL CHAR(139,"FCFCFBE0")
380 CALL CHAR(140,"0202020202020202")
390 CALL CHAR(141,"0000000303030202")
400 CALL CHAR(142,"0000000000808040")
410 CALL CHAR(143,"2020101010080808")
420 CALL CHAR(144,"00011EE")
430 CALL CHAR(145,"0FF0")
440 CALL CHAR(146,"00000000000031CE0")
450 CALL CHAR(147,"0000000003EC20202")
460 CALL SOUND(T,294,2)
470 DATA 76,69,84,39,83,32,76,69,65,82
    ,78,32
480 RESTORE 470
490 FOR Y=12 TO 23
500 READ L
510 CALL HCHAR(6,Y,L)
520 NEXT Y
530 CALL SOUND(T,330,2)
540 DATA 78,79,84,69,83,32
550 RESTORE 540
560 FOR Y=13 TO 23 STEP 2
570 READ L
580 CALL HCHAR(10,Y,L)
590 NEXT Y
600 CALL SOUND(T,349,2)
670 DATA 8,4,138,8,5,139,7,4,138,7,8,1
    37,6,5,140,5,5,140,4,5,141,4,6,144
    ,4,7,145
620 RESTORE 610
630 FOR I=1 TO 9
640 READ X,Y,GR
650 CALL HCHAR(X,Y,GR)
660 NEXT I
```

```
 670 CALL SOUND(T,392,2)
 680 DATA 3,8,146,3,9,147,4,9,140,5,9,1
     40,6,9,137,6,8,136,7,8,138,7,9,139
 690 RESTORE 680
 700 FOR I=1 TO 8
 710 READ X,Y,GR
 720 CALL HCHAR(X,Y,GR)
 730 NEXT I
 740 CALL SOUND(T,440,2)
 750 CALL VCHAR(16,12,49)
 760 DATA 75,69,89,66,79,65,82,68
 770 RESTORE 760
 780 FOR Y=14 TO 21
 790 READ L
 800 CALL HCHAR(16,Y,L)
 810 NEXT Y
 820 CALL SOUND(T,494,2)
 830 DATA 4,26,112,5,26,113,6,26,114,6,
     27,115,7,26,116,7,27,117
 840 RESTORE 830
 850 FOR I=1 TO 6
 860 READ X,Y,GR
 870 CALL HCHAR(X,Y,GR)
 880 NEXT I
 890 CALL SOUND(T,523,2)
 900 DATA 7,28,118,8,26,119,8,27,120,8,
     28,121,9,27,122
 910 RESTORE 900
 920 CALL SOUND(T,262,2)
 930 FOR I=1 TO 5
 940 READ X,Y,GR
 950 CALL HCHAR(X,Y,GR)
 960 NEXT I
 970 CALL SOUND(T,330,2)
 980 CALL HCHAR(18,12,50)
 990 DATA 84,82,69,66,76,69,32,67,76,69
     ,70
1000 RESTORE 990
1010 FOR Y=14 TO 24
1020 READ L
1030 CALL HCHAR(18,Y,L)
1040 NEXT Y
1050 CALL SOUND(T,392,2)
1060 CALL SOUND(T,523,2)
1070 CALL HCHAR(20,12,51)
1080 DATA 66,65,83,83,32,67,76,69,70
1090 RESTORE 1080
1100 FOR Y=14 TO 22
1110 READ L
1120 CALL HCHAR(20,Y,L)
1130 NEXT Y
1140 CALL SOUND(T,392,2)
1150 DATA 18,27,138,18,28,139,17,27,136
     ,17,28,137,16,28,140,15,28,140,14,
     28,141,14,29,142,15,29,143
1160 RESTORE 1150
1170 FOR I=1 TO 9
1180 READ X,Y,GR
1190 CALL HCHAR(X,Y,GR)
1200 NEXT I
1210 CALL SOUND(T,330,2)
1220 CALL SOUND(T,262,2)
1230 GOTO 5430
1240 CALL CLEAR
1250 CALL COLOR(4,2,7)
1260 CALL COLOR(9,2,1)
1270 CALL COLOR(10,2,16)
1280 CALL CHAR(96,"FF")
1290 CALL CHAR(104,"FF")
1300 CALL CHAR(105,"7F7F7F7F7F7F7F7F")
1310 CALL CHAR(106,"FEFEFEFEFEFEFEFE")
1320 CALL CHAR(107,"0101010101010101")
1330 CALL CHAR(108,"8080808080808080")
1340 CALL CHAR(109,"0")
1350 RANDOMIZE
1360 COUNT=0
1370 KB=INT(2*RND)+1
1380 CALL CLEAR
1390 DATA 78,65,77,69,32,84,72,69,32,75
     ,69,89
1400 RESTORE 1390
1410 FOR Y=11 TO 22
1420 READ L
1430 CALL HCHAR(4,Y,L)
1440 NEXT Y
1450 PAT=KB
1460 REM DRAW KEYBOARD
1470 CALL HCHAR(8,1,104,32)
1480 CALL HCHAR(18,1,96,32)
1490 CALL VCHAR(8,1,109,10)
1500 CALL VCHAR(8,32,109,10)
1510 FOR Y=3 TO 30 STEP 3
1520 CALL VCHAR(8,Y-1,109,10)
1530 CALL VCHAR(8,Y,107,10)
1540 CALL VCHAR(8,Y+1,108,10)
1550 NEXT Y
1560 IF KB=2 THEN 1660
1570 REM KEYBOARD 1
1580 DATA 3,6,12,15,18,24,27,27
1590 RESTORE 1580
1600 FOR I=1 TO 8
1610 READ Y
1620 CALL VCHAR(8,Y,105,7)
1630 CALL VCHAR(8,Y+1,106,7)
1640 NEXT I
1650 GOTO 1740
1660 REM KEYBOARD 2
1670 DATA 3,6,9,15,18,24,27,30,30
1680 RESTORE 1670
1690 FOR I=1 TO 9
1700 READ Y
1710 CALL VCHAR(8,Y,105,7)
1720 CALL VCHAR(8,Y+1,106,7)
1730 NEXT I
1740 REM PICK KEY
1750 NN=INT(11*RND)+1
1760 J=3*NN-1
1770 CALL HCHAR(16,J,63)
1780 FOR I=1 TO 10
1790 CALL COLOR(4,2,7)
1800 CALL COLOR(4,16,7)
1810 NEXT I
1820 CALL KEY(0,NOTE,STATUS)
1830 IF STATUS<>1 THEN 1820
1840 IF KB=2 THEN 1860
1850 ON NN GOTO 2050,2120,2190,2260,233
     0,2400,2470,2540,2580,2620,2660
1860 ON NN GOTO 1890,1930,1970,2010,205
     0,2120,2190,2260,2330,2400,2470
1870 CALL SOUND(500,-4,2)
1880 GOTO 1820
1890 IF NOTE<>70 THEN 1870
1900 CALL SOUND(T,175,2)
1910 CALL HCHAR(16,2,70)
1920 GOTO 2690
1930 IF NOTE<>71 THEN 1870
1940 CALL SOUND(T,196,2)
1950 CALL HCHAR(16,5,71)
1960 GOTO 2690
1970 IF NOTE<>65 THEN 1870
1980 CALL SOUND(T,220,2)
1990 CALL HCHAR(16,8,65)
2000 GOTO 2690
2010 IF NOTE<>66 THEN 1870
2020 CALL SOUND(T,247,2)
2030 CALL HCHAR(16,11,66)
2040 GOTO 2690
2050 IF NOTE<>67 THEN 1870
2060 CALL SOUND(T,262,2)
2070 IF KB=2 THEN 2100
2080 CALL HCHAR(16,2,67)
2090 GOTO 2690
2100 CALL HCHAR(16,14,67)
2110 GOTO 2690
2120 IF NOTE<>68 THEN 1870
2130 CALL SOUND(T,294,2)
```

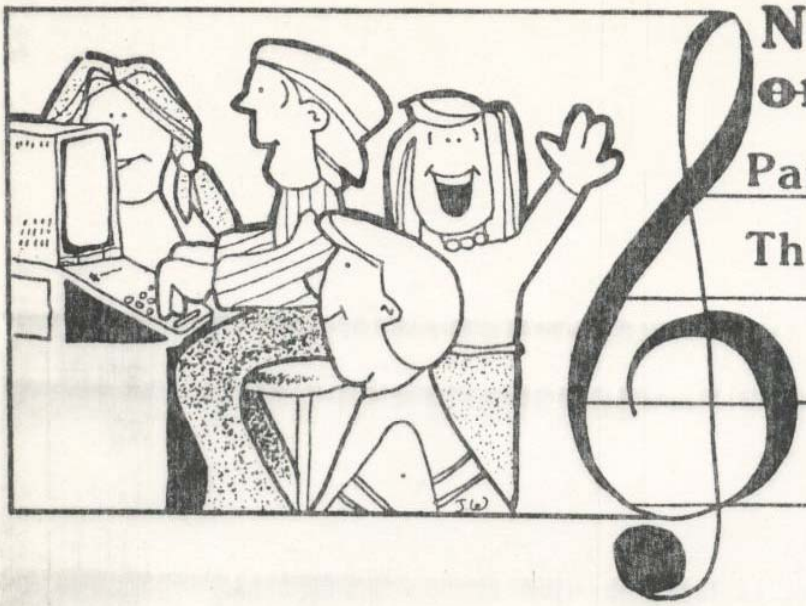
```
2140 IF KB=2 THEN 2170
2150 CALL HCHAR(16,5,68)
2160 GOTO 2690
2170 CALL HCHAR(16,17,68)
2180 GOTO 2690
2190 IF NOTE<>69 THEN 1870
2200 CALL SOUND(T,330,2)
2210 IF KB=2 THEN 2240
2220 CALL HCHAR(16,8,69)
2230 GOTO 2690
2240 CALL HCHAR(16,20,69)
2250 GOTO 2690
2260 IF NOTE<>70 THEN 1870
2270 CALL SOUND(T,349,2)
2280 IF KB=2 THEN 2310
2290 CALL HCHAR(16,11,70)
2300 GOTO 2690
2310 CALL HCHAR(16,23,70)
2320 GOTO 2690
2330 IF NOTE<>71 THEN 1870
2340 CALL SOUND(T,392,2)
2350 IF KB=2 THEN 2380
2360 CALL HCHAR(16,14,71)
2370 GOTO 2690
2380 CALL HCHAR(16,26,71)
2390 GOTO 2690
2400 IF NOTE<>65 THEN 1870
2410 CALL SOUND(T,440,2)
2420 IF KB=2 THEN 2450
2430 CALL HCHAR(16,17,65)
2440 GOTO 2690
2450 CALL HCHAR(16,29,65)
2460 GOTO 2690
2470 IF NOTE<>66 THEN 1870
2480 CALL SOUND(T,494,2)
2490 IF KB=2 THEN 2520
2500 CALL HCHAR(16,20,66)
2510 GOTO 2690
2520 CALL HCHAR(16,32,66)
2530 GOTO 2690
2540 IF NOTE<>67 THEN 1870
2550 CALL SOUND(T,523,2)
2560 CALL HCHAR(16,23,67)
2570 GOTO 2690
2580 IF NOTE<>68 THEN 1870
2590 CALL SOUND(T,587,2)
2600 CALL HCHAR(16,26,68)
2610 GOTO 2690
2620 IF NOTE<>69 THEN 1870
2630 CALL SOUND(T,659,2)
2640 CALL HCHAR(16,29,69)
2650 GOTO 2690
2660 IF NOTE<>70 THEN 1870
2670 CALL SOUND(T,698,2)
2680 CALL HCHAR(16,32,70)
2690 FOR DELAY=1 TO 1000
2700 NEXT DELAY
2710 CALL HCHAR(16,J,109)
2720 COUNT=COUNT+1
2730 IF COUNT=10 THEN 5400
2740 KB=INT(2*RND)+1
2750 IF KB=PAT THEN 1740 ELSE 1380
2760 REM TREBLE CLEF NOTES
2770 CALL CLEAR
2780 FOR I=9 TO 15
2790 CALL COLOR(I,2,1)
2800 NEXT I
2810 CALL CHAR(96,"FF")
2820 CALL CHAR(104,"0000007F3F78F0E0")
2830 CALL CHAR(105,"00000C0F0F0783C")
2840 CALL CHAR(106,"C0808080808080808080")
2850 CALL CHAR(107,"1C0C0C0C0C0C0C040404")
2860 CALL CHAR(108,"8080808080808080C040")
2870 CALL CHAR(109,"0404040404080808181")
2880 CALL CHAR(110,"FF4040404040202020")
2890 CALL CHAR(111,"FF3040606040404C080")
2900 CALL CHAR(112,"2121331216161C1C18")
2910 CALL CHAR(113,"FF0000001030707F1E")
2920 CALL CHAR(114,"FF78E8C8848404040")
2930 CALL CHAR(115,"0000000010103070F")
2940 CALL CHAR(116,"3C78F0E0C0808080")
2950 CALL CHAR(117,"0404020202020202020")
2960 CALL CHAR(118,"FF0E1C1838383070706")
2970 CALL CHAR(119,"FF0107F1F3D3971")
2980 CALL CHAR(120,"FFFFFFFFC0")
2990 CALL CHAR(121,"FFFEFF0F03081")
3000 CALL CHAR(122,"FF000080C0C0E0606")
3010 CALL CHAR(123,"606060E0E0606060606")
3020 CALL CHAR(125,"60E0C0C0C0C0C0E06")
3030 CALL CHAR(126,"8080808080808080806")
3040 CALL CHAR(127,"3030101010101010106")
3050 CALL CHAR(128,"FF707033833C1C1E")
3060 CALL CHAR(129,"FF6060303106")
3070 CALL CHAR(130,"FF4040404040404040406")
3080 CALL CHAR(131,"FF1020202040C086")
3090 CALL CHAR(132,"0F0703081")
3100 CALL CHAR(133,"0080C0F0F833C0F03")
3110 CALL CHAR(134,"0000000000000000C0")
3120 CALL CHAR(135,"2020202020202020212E")
3130 CALL CHAR(136,"0103060810606086")
3140 CALL CHAR(137,"FF2020202020202020206")
3150 CALL CHAR(138,"1010101010101010106")
3160 CALL CHAR(139,"3C7EFEFEFC7C391F")
3170 CALL CHAR(140,"1010103020406086")
3180 CALL CHAR(141,"FF8080808080808080806")
3190 CALL CHAR(152,"030F1F3F7FFFFFFFF")
3200 CALL CHAR(153,"FCFFFFFFFFFFFFFFF")
3210 CALL CHAR(154,"FFFFFFF7F3F1F0F03")
3220 CALL CHAR(155,"FFFFFFFFFFFFFFFC")
3230 CALL CHAR(156,"F0F0F0E0C0806")
3240 CALL CHAR(157,"1010909D9F0F0F0F0F06")
3250 CALL CHAR(158,"00000080C0E0F0F0F06")
3260 CALL CHAR(159,"0101010101010101016")
3270 CALL COLOR(16,7,1)
3280 GOSUB 4980
3290 DATA 78,65,77,69,32,84,72,69,32,78
     ,79,84,69
3300 RESTORE 3290
3310 FOR Y=10 TO 22
3320 READ L
3330 CALL HCHAR(6,Y,L)
3340 NEXT Y
3350 IF GAME=3 THEN 4940
3360 REM TREBLE CLEF
3370 DATA 9,5,104,9,6,105,10,5,106,10,6
     ,107,11,5,108,11,6,109,12,5,110,12
     ,6,111,13,5,112
3380 RESTORE 3370
3390 DATA 14,4,113,14,5,114,15,3,115,15
     ,4,116,15,5,117,16,3,118,16,5,119,
     16,6,120,16,7,121,16,8,122
3400 DATA 17,3,123,17,5,125,17,6,126,17
     ,8,127,18,3,128,18,5,129,18,6,130,
     18,8,131,19,3,132
3410 DATA 19,4,133,19,5,134,19,6,135,19
     ,7,136,20,6,137,21,6,138,22,5,139,
     22,6,140,1,1,32
3420 FOR I=1 TO 37
3430 READ X,Y,GR
3440 CALL HCHAR(X,Y,GR)
3450 NEXT I
3460 COUNT=0
3470 RANDOMIZE
3480 NN=INT(11*RND)+1
3490 M=22-NN
3500 GOSUB 5070
3510 IF NN<>1 THEN 3540
3520 CALL HCHAR(M+1,14,96)
3530 CALL HCHAR(M+1,18,96)
3540 CALL KEY(0,NOTE,STATUS)
3550 IF STATUS<>1 THEN 3540
3560 ON NN GOTO 3590,3630,3670,3710,375
     0,3790,3830,3870,3910,3950,3990
3570 CALL SOUND(600,-8,2)
3580 GOTO 3540
3590 IF NOTE<>67 THEN 3570
```

```
3600 CALL SOUND(T,262,2)
3610 CALL HCHAR(M+1,16,67)
3620 GOTO 4020
3630 IF NOTE<>68 THEN 3570
3640 CALL SOUND(T,294,2)
3650 CALL HCHAR(M+1,16,68)
3660 GOTO 4020
3670 IF NOTE<>69 THEN 3570
3680 CALL SOUND(T,330,2)
3690 CALL HCHAR(M+1,16,69)
3700 GOTO 4020
3710 IF NOTE<>70 THEN 3570
3720 CALL SOUND(T,349,2)
3730 CALL HCHAR(M+1,16,70)
3740 GOTO 4020
3750 IF NOTE<>71 THEN 3570
3760 CALL SOUND(T,392,2)
3770 CALL HCHAR(M+1,16,71)
3780 GOTO 4020
3790 IF NOTE<>65 THEN 3570
3800 CALL SOUND(T,440,2)
3810 CALL HCHAR(M+1,16,65)
3820 GOTO 4020
3830 IF NOTE<>66 THEN 3570
3840 CALL SOUND(T,494,2)
3850 CALL HCHAR(M+1,16,66)
3860 GOTO 4020
3870 IF NOTE<>67 THEN 3570
3880 CALL SOUND(T,523,2)
3890 CALL HCHAR(M+1,16,67)
3900 GOTO 4020
3910 IF NOTE<>68 THEN 3570
3920 CALL SOUND(T,587,2)
3930 CALL HCHAR(M+1,16,68)
3940 GOTO 4020
3950 IF NOTE<>69 THEN 3570
3960 CALL SOUND(T,659,2)
3970 CALL HCHAR(M+1,16,69)
3980 GOTO 4020
3990 IF NOTE<>70 THEN 3570
4000 CALL SOUND(T,698,2)
4010 CALL HCHAR(M+1,16,70)
4020 GOSUB 5220
4030 GOTO 3480
4040 CALL CHAR(97,"FF0000000000307878FC")
4050 CALL CHAR(98,"FC7830")
4060 CALL CHAR(99,"FFFF7E3C")
4070 CALL CHAR(100,"1810206060607C7EFF")
4080 CALL CHAR(101,"FF0000001030606040")
4090 CALL CHAR(102,"FF0F70C0")
4100 CALL CHAR(103,"FFC0381C0E070303")
4110 CALL CHAR(124,"C0C0606030383838383838")
4120 CALL CHAR(142,"FF1C1C1C1C1C1C1C1C1C")
4130 CALL CHAR(143,"1C1C1C1C1C1C1C1C1C1C")
4140 CALL CHAR(144,"FF383838387070C0C0")
4150 CALL CHAR(145,"0303030604040C1870C0")
4160 CALL CHAR(146,"FF03060C10608080")
4170 DATA 14,3,99,13,3,100,12,3,101,12,
     4,102,12,6,103,13,7,124,14,7,142
4180 RESTORE 4170
4190 DATA 15,7,143,16,7,144,17,6,145,18
     ,5,146,12,9,97,13,9,98,14,9,97,15,
     9,98,1,1,32
4200 FOR I=1 TO 16
4210 READ X,Y,GR
4220 CALL HCHAR(X,Y,GR)
4230 NEXT I
4240 COUNT=0
4250 RANDOMIZE
4260 NN=INT(11*RND)+1
4270 M=20-NN
4280 GOSUB 5070
4290 IF NN<>11 THEN 4320
4300 CALL HCHAR(M+1,14,96)
4310 CALL HCHAR(M+1,18,96)
4320 CALL KEY(0,NOTE,STATUS)
4330 IF STATUS<>1 THEN 4320
4340 ON NN GOTO 4370,4420,4470,4520,457
     0,4620,4670,4720,4770,4820,4870
```

```
4350 CALL SOUND(600,-8,2)
4360 GOTO 4320
4370 IF NOTE<>71 THEN 4350
4380 J=110
4390 GOSUB 4930
4400 CALL HCHAR(20,16,71)
4410 GOTO 4910
4420 IF NOTE<>65 THEN 4350
4430 J=116
4440 GOSUB 4930
4450 CALL HCHAR(19,16,65)
4460 GOTO 4910
4470 IF NOTE<>66 THEN 4350
4480 J=123
4490 GOSUB 4930
4500 CALL HCHAR(18,16,66)
4510 GOTO 4910
4520 IF NOTE<>67 THEN 4350
4530 J=131
4540 GOSUB 4930
4550 CALL HCHAR(17,16,67)
4560 GOTO 4910
4570 IF NOTE<>68 THEN 4350
4580 J=147
4590 GOSUB 4930
4600 CALL HCHAR(16,16,68)
4610 GOTO 4910
4620 IF NOTE<>69 THEN 4350
4630 J=165
4640 GOSUB 4930
4650 CALL HCHAR(15,16,69)
4660 GOTO 4910
4670 IF NOTE<>70 THEN 4350
4680 J=175
4690 GOSUB 4930
4700 CALL HCHAR(14,16,70)
4710 GOTO 4910
4720 IF NOTE<>71 THEN 4350
4730 J=196
4740 GOSUB 4930
4750 CALL HCHAR(13,16,71)
4760 GOTO 4910
4770 IF NOTE<>65 THEN 4350
4780 J=220
4790 GOSUB 4930
4800 CALL HCHAR(12,16,65)
4810 GOTO 4910
4820 IF NOTE<>66 THEN 4350
4830 J=247
4840 GOSUB 4930
4850 CALL HCHAR(11,16,66)
4860 GOTO 4910
4870 IF NOTE<>67 THEN 4350
4880 J=262
4890 GOSUB 4930
4900 CALL HCHAR(10,16,67)
4910 GOSUB 5220
4920 GOTO 4260
4930 FOR I=1 TO 5
4940 CALL SOUND(200,J,2)
4950 J=J+15
4960 NEXT I
4970 RETURN
4980 REM STAFF
4990 FOR X=12 TO 20 STEP 2
5000 CALL HCHAR(X,1,96,31)
5010 NEXT X
5020 FOR X=12 TO 18 STEP 2
5030 CALL HCHAR(X,1,141)
5040 CALL HCHAR(X+1,1,126)
5050 NEXT X
5060 RETURN
5070 REM NOTE
5080 CALL HCHAR(M,15,152)
5090 CALL HCHAR(M,16,153)
5100 CALL HCHAR(M+1,15,154)
5110 CALL HCHAR(M+1,16,155)
5120 CALL HCHAR(M+1,17,156)
```

```
5130 IF GAME=3 THEN 5210
5140 IF NN<8 THEN 5180
5150 CALL HCHAR(M,17,158)
5160 CALL VCHAR(M+1,14,159,6)
5170 RETURN
5180 CALL HCHAR(M,17,157)
5190 CALL VCHAR(M-5,17,138,5)
5200 RETURN
5210 IF NN<5 THEN 5180 ELSE 5150
5220 REM CHANGE NOTES
5230 FOR DELAY=1 TO 1000
5240 NEXT DELAY
5250 COUNT=COUNT+1
5260 IF COUNT=10 THEN 5400
5270 CALL HCHAR(M,14,32,5)
5280 CALL HCHAR(M+1,14,32,5)
5290 IF GAME=3 THEN 5350
5300 IF NN<8 THEN 5330
5310 CALL VCHAR(M+1,14,32,6)
```

```
5320 GOTO 5360
5330 CALL VCHAR(M-5,17,32,5)
5340 GOTO 5360
5350 IF NN<5 THEN 5330 ELSE 5310
5360 FOR K=12 TO 20 STEP 2
5370 CALL HCHAR(K,14,96,5)
5380 NEXT K
5390 RETURN
5400 PRINT "PRESS 1 FOR KEYBOARD QUIZ"
5410 PRINT TAB(7);"2 FOR TREBLE NOTES"
5420 PRINT TAB(7);"3 FOR BASS NOTES"
5430 PRINT TAB(7);"4 TO END PROGRAM"
5440 CALL KEY(0,K,S)
5450 IF K=52 THEN 5500
5460 IF K>51 THEN 5440
5470 IF K<49 THEN 5440
5480 GAME=K-48
5490 IF GAME=1 THEN 1240 ELSE 2770
5500 END
```



Notes on a Computer Score:

Part 1 - The TI-99/4 Conducts Music Theory Drill in a Traditional Classroom Setting.

Recently I returned to my job as elementary music teacher is the Rossford (Ohio) School District after an exciting and rewarding summer. When reading students' responses to the question, "What did you most enjoy about music last year?" on a first-day questionnaire, I was pleased to see the number of students responding, "The computer." At New Horizons Academy for the Gifted, a Computer-Assisted Music Program held in the summer of 1981 elicited a similar response from students. In both of these very different educational contexts, computer usage has proved to be a strong motivational force in students' acquisition of music theory and skills.

Last year my husband and I purchased a TI-99/4 with some rather nebulous ideas about potential applications in the general music curriculum. Together we worked on the development of programs—trying to incorporate motivational strategies which apply in virtually any teaching situation—and experimented with various uses of commercially available software (e.g., the *Music Maker* Command Cartridge and *Music Skills Trainer*). We tried many techniques in the classroom to determine the children's responses. The result has been the continuing evolution of a computer-assisted music curriculum tailored to the needs of my teaching situation. It has been a stimulating year—one which progressed from ignorance about using the computer and apprehensions concerning its effectiveness to one of the most exciting experiences of my teaching career.

Whether you are a teacher planning a Computer-Assisted Instruction (CAI) project or a parent considering the potential educational value of a computer in your home, it may not be necessary to know exactly where you are going before you take the first step. Our experience has shown that the element of discovery inherent in developing a curriculum interactively with children can be as rewarding and exciting a learning process for the educator as is the use of the final product for the student.

Glenwood School in the Rossford (Ohio) district is a typical elementary school with an enrollment of 400 students in grades 1-6. I incorporated the computer into my general music curriculum for grades 4-6 during the fall semester of the 1980-81 school year (my first year in this system). Classes were intact groups which met for two 35-minute periods each week in the "music room"—a corner of the cafeteria. Average class size was 25 students, with pupils from the Adjusted Curriculum (learning difficulties) as well as Project Horizons (gifted) programs mainstreamed into the regular classes

My classes are organized around the belief that music should be fun and provide students with an outlet for their creativity. Although music class can be a break from routine academics, children must be equipped with basic knowledge of the fundamentals of music reading and theory upon completion of a general music course. A variety of experiences —singing, movement, listening and playing instruments— should be provided. The computer was employed as an additional enrichment activity—one which turned out to be unusually effective for the students, as well as challenging for the teacher.

The general approach I employ involves an initial experiential emphasis (e.g., singing and movement) followed by instruction in the basic theory required to read music. In addition to providing knowledge of theory, this approach readies students for potential participation in band and choir.

Two computer programs, *Rhythm* and *Mystery Words*, were used to reinforce two aspects of the curriculum: (1) aural recognition of rhythmic patterns, and (2) knowledge of musical notation for note names in both treble and bass clef.

In teaching students how to discriminate between various note values and rests, I first used "Echo Clapping" in which I clapped a rhythmic pattern and the students tried to reproduce it. Next, students were taught to associate appropriate music terms with relative durations (whole, half, quarter, eighth and their corresponding rests).

Then we progressed to an activity called "Rhythmic Dictation" in which students wrote in musical notation the rhythms they heard clapped (e.g., ♩♩♩♩, ♩𝅘𝅥𝅯𝅘𝅥𝅯𝅘𝅥𝅯𝅘𝅥𝅯, ♩♩𝅘𝅥𝅯𝅘𝅥𝅯♩). So as not to get too complex at the beginning, only the ♩, 𝅘𝅥𝅯𝅘𝅥𝅯 and 𝄽 were taught.

Following presentation of the concepts and introduction to initial rhythmic dictation exercises, many students reach

a plateau when acquiring the skill of describing rhythmic patterns in musical notation. Representation of rhythmic patterns by clapping is to some extent an abstraction because it requires analysis of the relationship between a steady beat and intervals between claps. Although most children are highly motivated to acquire this skill, many encounter difficulties which result, in part, from its abstract nature.

To address this problem, we tried to incorporate two principles into the *Rhythm* program: (1) concreteness, and (2) immediacy of feedback. At the same time, in order to optimize the effect of one computer on 25 children, we decided to use a game format—allowing for the possibility of up to ten teams within a class. We also put a lot of effort into the introductory portion of the program to catch students' attention with the color and sound capabilities of the TI-99/4.

I initally introduced this program in my sixth grade classes after considerable time had been spent practicing rhythmic dictation. Periodic quizzes showed that a fair number of students in all three classes had not grasped the concept. After three sessions with the *Rhythm* program, almost every student had become competent in clapped rhythmic dictation.

I believe several factors contributed to this remarkable improvement: The activity was conceptually more concrete than rhythmic dictation, and it provided students with immediate feedback which included the correct response when mistakes were made. Concentration and motivation were improved, in part, simply by the uniqueness of the computer activity. It was not uncommon for several students to show up in the music room shortly after their bus arrived, in the hope of spending ten minutes working with a computer music game before school started. When teachers arrived after a general music class to take their students back to their classrooms, we invariably had to pry some of them away; classes sometimes had to be actually extended 10 to 15 minutes!

The group dynamics involved also played a significant role. During the first session, the sixth graders asked to have team scores added together for comparison with the other sixth grade classes. Because each class represented an intact group with some history and cohesiveness, a positive atmosphere prevailed in which the student working at the computer was supported and encouraged by the cheers and comments of fellow classmates. All students had several opportunities to make a contribution to the class total score.

Next, I used the *Rhythm* program in the fifth grade. Most students were aware through word-of-mouth that the sixth graders had been using a computer and naturally were interested in the top score the sixth graders had achieved. The typical score for the first day in the sixth grade had been 15 to 20. By comparison, the first day scores in the fifth grade were as high as 45! Similar enthusiam was observed in the fourth grade.

I anticipated these outcomes, although the actual impressive results far exceeded my expectations. There were also some genuine surprises: First, using the computer allowed me to observe and diagnose the problems of individual students and, where necessary, to take them aside and give special attention to their needs while the class was occupied with the computer. Learning the concept was important to them in order to make points for their class.

Another gratifying result was that the students with learning difficulties found it easier to master this more concrete activity and took a great deal of pride in their contributions to the class score. It was indeed rewarding to see their beaming faces as classmates cheered and patted them on the back after their correct responses.

Following the computer activity, nearly every student had achieved competency in the basic rhythms which had been presented, and they were able to apply this knowledge in the playing of rhythm instruments. I wrote several lines of rhythmic patterns on the board in musical notation and asked individuals or small groups to play them. Subsequently, the patterns were played to accompany class singing or listening to records.

The rhythm unit was followed by the study of musical notation for pitch. Students were introduced to this concept through a discussion of the importance of learning the note names on the staff in order to read music when singing or playing instruments. I compared note reading to reading a foreign language, using symbols and notes instead of words to create a musical story.

Initial instruction presented the familiar phrases "Every Good Boy Does Fine" and "F A C E" to facilitate learning the position of notes in the treble clef, and this was followed by drills to further reinforce note name recognition. Students were promised that the computer would be brought back to class when they learned these note names well enough to play a computer game. Thereafter, the *Mystery Words* game listed at the end of this article was introduced.

Mystery Words is a game that is based upon the use of note name letters to spell a variety of words, for example,"cabbage," "bead," and "facade." The program randomly chooses one of these words and represents it in music notation graphics in the treble clef, the bass clef, or both. The teacher has the option of excluding words with more difficult meanings (such as "facade" or "accede") when using the game with younger students.

The screen is divided in half with a red side and a blue side corresponding to the red and blue teams into which the class is divided. One member of each team is seated at the console. Before the presentation of a Mystery Word each player must signal he is ready by pressing the "1" or "0" key. As each team member signals readiness, a traffic light on each side of the screen changes from red to yellow to green, the Mystery Theme is played, and the graphic representation of the Mystery Word appears. The first student to decipher the word presses the 1 or 0 key, and the graphic representation disappears. He is then instructed to enter the answer using the keys 3 through 9 which have been labeled A through G on a blank keyboard overlay. As each letter is pressed, it appears in the graphics window. If the entire word is entered correctly, the graphic representation reappears with notes above the letters entered by the student, and the team's score is incremented. In the event an incorrect letter is entered, the opposing team member is instructed to try.

When the game was introduced in class, only the treble clef option was selected since previous instruction did not include the bass clef. Prior experience suggested that the presentation of both treble and bass clefs was too confusing for the average elementary age student. But using the

computer, students quickly mastered treble clef note names and requested that they be allowed to try working with the bass clef as well. Their ability to learn bass clef note names rapidly with minimal prior classroom work and to work with both clefs simultaneously was truly amazing.

Use of *Mystery Words* was accompanied by the same intense interest and motivation as *Rhythm*. Although this game was also constructed for intra-class competition, students again asked that team scores be added together for comparison with other classes. Some students began showing up before school with younger brothers and sisters to explain the computer games to them, and I became a popular figure among students in the cafeteria at lunch time—the main topic of conversation being the computer.

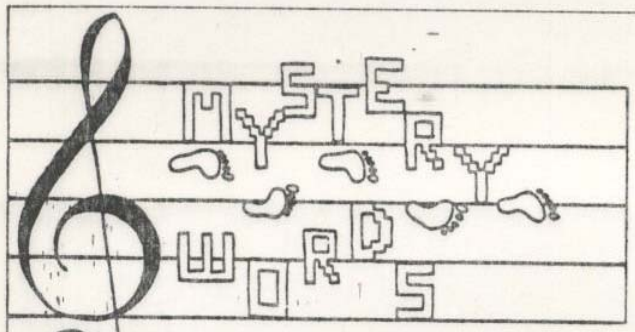
Seymour Papert defines three components for learning mathematics: the *Continuity Principle*, the *Power Principle* and the *Principle of Cultural Resonance*.* These principles, of course, may be applied to the acquisition of any content domain—not just mathematics. This is to say that a concept or skill may be acquired with the least effort if it (1) is continuous with what the learner already knows, (2) empowers him to achieve personal objectives which could not be achieved otherwise, and (3) makes sense within a larger social context. Construing the computer-assisted units on these principles may help to elucidate some of the elements which I believe are critical to the success of this application (and for that matter, of any learning environment).

All children are intimately familar with music in their everyday environment. The initial emphasis on those aspects of music already familar to them, i.e., singing and movement, helped to give a sense of continuity with respect to subsequent course material. Second, the computer activity was integrated into a larger social context by the students themselves when they asked that team scores be added together for comparison with other classes. This phenomenon was also apparent when students in lower grades expressed interest in the scores obtained by the upper grades and used this information in the setting of personal goals.

But perhaps the most important element is the *Power Principle*. Students perceived that the acquisiton of music skills would enable them to make contributions toward group achievement in the computer music game. Later they found that they were able to play rhythm instruments and read musical notation, and at the same time, they realized that these skills may be useful to them in the future when participating in band and chorus activities, which are themselves part of a meaningful social context.

In summary, use of the computer increased student motivation, and at the same time allowed abstract material to be represented in a concrete way, leading to more rapid acquisition of skills and concepts. The computer also made it possible to diagnose individual weaknesses and provide individualized remedial work. Discipline problems arising from boredom and lack of interest were nonexistent when the computer was in the classroom. In general, a positive environment, cooperation, and mutual support predominated.

*Papert, S. *Mindstorms: Children, Computers, and Powerful Ideas*. New York: Basic Books, 1980.



EXPLANATION OF THE PROGRAM
Mystery Words

Line Nos.	
130-200	Clears screen; sets screen color.
210-270	Accepts user input for word list.
280-350	Accepts user choice of type of drill.
360-420	Instruction for keyboard labeling.
430-490	Interrupts program for explanation to class.
500-700	Displays notes of treble clef for review.
710-840	Displays notes of bass clef for review.
850-870	Restores data pointer and dimensions arrays.
888-980	Defines character patterns and sets colors for initial graphic screen.
990-1100	Displays initial graphics screen; plays associated sounds.
1110-1330	Displays *Mystery Words* title screen.
1340-1390	Assigns word codes to SET$ array; numeric digits correspond to letters (1 = A, 2 = B, etc.).
1400-1470	Assigns coded print locations (potentially 2 for each staff) for each of the 7 letter codes to array LINE.
1480-1920	Defines character patterns and sets colors used in the game screen.
1930-2090	Displays basic elements of game screen.
2100-2110	Selects a Mystery Word randomly.
2120-2530	Determines a representation for word on staff(s); when more than 1 possibility for a location exists, choice is random. The word is played in invisible characters (white on white).
2540-3000	Displays READY message, changes traffic light colors as player signal ready by pressing keys 1 and 0; plays Mystery Theme.
3010-3030	Makes representation of Mystery Word visible.
3040-3100	Accepts answer signals from players.
3110-3180	Flashes screen of red team.
3190-3250	Flashes screen of blue team.
3260-3280	Removes Mystery Word from view pending player's response.
3290-3300	Instructs team to answer.
3310-3450	Accepts key 3-9 (A-G) input, checking each letter against correct spelling.
3460-3690	If answer is correct, displays Mystery Word again with spelling; increments team score; returns to line 1930.
3700-3820	If answer is incorrect, allows opposing team to respond.
Subroutines:	
3830-4330	Displays game screen.
4340-4470	Draws treble staff.
4480-4570	Draws bass staff.
4580-4630	Displays messages, MSG$.
4640-4700	Displays letters of answer.
4710-4830	Displays initial graphic screen.
4840-4900	Plays Mystery Theme.
4910-5140	Displays treble and/or bass clef notes for initial review.
5160-5180	Erases screens after initial user input.

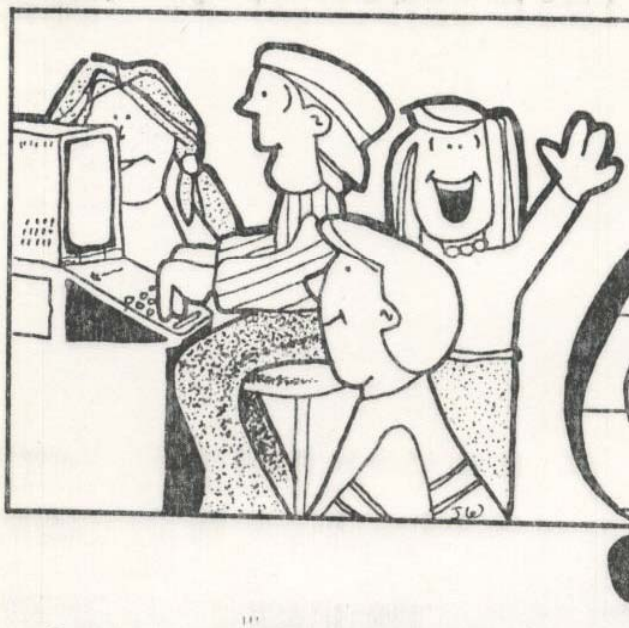
```
100 REM      MYSTERY WORDS
110 REM
120 REM
130 CALL CLEAR
140 NSET=52
150 RANDOMIZE
160 CALL SCREEN(5)
170 CALL VCHAR(1,31,1,96)
180 FOR I=1 TO 8
190 CALL COLOR(I,2,8)
200 NEXT I
210 PRINT "SOME OF THE WORDS IN THE
    WORD LIST ARE NOT READILY UNDE
    RSTOOD BY YOUNGER CHIL-DREN.":::
220 PRINT "WOULD YOU LIKE TO EXCLUDE
    THESE WORDS (Y/N)?":::::::
    :
230 CALL KEY(0,K,S)
240 IF K=89 THEN 270
250 IF K<>78 THEN 230
260 NSET=59
270 CALL HCHAR(16,23,K)
280 GOSUB 5150
290 PRINT "THREE DRILLS ARE AVAILABLE
    ":::::TAB(5);"1) TREBLE CLEF":::
    TAB(5);"2) BASS CLEF":::
300 PRINT TAB(5);"3) TREBLE AND BASS":
    :::::"YOUR CHOICE (1,2,OR 3)?"
    :::::
310 CALL KEY(0,K,S)
320 IF K<49 THEN 310
330 IF K>51 THEN 310
340 CLEF=K-48
350 CALL HCHAR(21,28,K)
360 GOSUB 5150
370 PRINT TAB(9);"INSTRUCTIONS":::"LAB
    EL 3-9 AS A-G, 1 RED AND 0 BLUE WI
    TH BLANK OVERLAY."
380 PRINT :"BOTH 1 AND 0 MUST BE PRESS
    EDAFTER TRAFFIC LIGHT IS RED."
390 PRINT :"FIRST PLAYER TO DECODE WOR
    D PRESSES 1/0 AND USES 3-9 TO ENTE
    R THE ANSWER."
400 PRINT :::::::TAB(8);"PRESS ANY K
    EY."
410 CALL KEY(0,K,S)
420 IF S=0 THEN 410
430 GOSUB 5150
440 MSG$="PRESS ANY KEY TO BEGIN"
450 ROW=12
460 COL=6
470 GOSUB 4600
480 CALL KEY(0,K,S)
490 IF S=0 THEN 480
500 CALL CLEAR
510 FOR I=1 TO 8
520 CALL COLOR(I,2,16)
530 NEXT I
540 RESTORE 5110
550 GOTO 1480
560 CALL CHAR(119,"000000FFFF")
570 ON CLEF GOTO 580,710,580
580 GOSUB 4340
590 ROW=4
600 COL=12
610 MSG$="TREBLE CLEF"
620 GOSUB 4600
630 ROW=23
640 COL=10
650 F1=4
660 S1=24
670 F2=20
680 S2=39
690 GOSUB 4910
700 IF CLEF<>3 THEN 850
710 CALL CLEAR
720 R=2
730 GOSUB 4480
740 ROW=4
750 COL=13
760 MSG$="BASS CLEF"
770 GOSUB 4600
780 ROW=23
790 COL=10
800 F1=1
810 S1=21
820 F2=14
830 S2=33
835 RESTORE 5130
840 GOSUB 4910
850 RESTORE
860 OPTION BASE 1
870 DIM SET$(59),NLINE(3),LINE(7,2,2)
880 CALL CLEAR
890 FLAG=1
900 CALL CHAR(35,"FFFFFFFFFFFFFFFF")
910 CALL CHAR(40,"")
920 CALL CHAR(96,"3C7E7E7E7E7E7E7E")
930 CALL CHAR(97,"3C3C0000003C3C3C")
940 CALL CLEAR
950 CALL COLOR(1,6,1)
960 CALL SCREEN(2)
970 CALL COLOR(2,7,7)
980 CALL COLOR(9,11,2)
990 R=24
1000 M=2
1010 MM=500
1020 FOR I=1 TO 1000
1030 NEXT I
1040 GOSUB 4710
1050 GOSUB 4840
1060 M=4
1070 MM=275
1080 GOSUB 4710
1090 GOSUB 4840
1100 CALL SOUND(1200,262,1,311,1,392,1)
1110 CALL CLEAR
1120 FOR I=1 TO 500
1130 NEXT I
1140 FOR I=1 TO 99
1150 READ D$
1160 CALL HCHAR(VAL(SEG$(D$,1,2)),VAL(S
     EG$(D$,3,2)),40)
1170 NEXT I
1180 FOR I=100 TO 188
1190 READ D$
1200 CALL HCHAR(VAL(SEG$(D$,1,2)),VAL(S
     EG$(D$,3,2)),35)
1210 NEXT I
1220 DATA 0503,0603,0703,0803,0903,1003
     ,1103,0604,0705,0606,0507,0607,070
     7,0807,0907,1007
1230 DATA 1107,0107,0207,0308,0111,0211
     ,0310,0409,0509,0609,0709,0809,071
     4,0713,0712,0711
1240 DATA 0811,0911,1011,1012,1013,1014
     ,1114,1214,1314,1313,1312,1311,051
     4,0515,0516,0517
1250 DATA 0518,0616,0716,0816,0916,1016
     ,1116,0721,0720,0719,0718,0818,091
     8,1018,1118,1218
1260 DATA 1318,1319,1320,1321,1019,1020
     ,0523,0623,0723,0823,0923,1023,112
     3,0524,0525,0526
1270 DATA 0626,0726,0826,0825,0824,0924
     ,1025,1126,0126,0226,0327,0130,023
     0,0329,0428,0528,0628
1280 DATA 0728,0828,1504,1604,1704,1804
     ,1904,2004,2104,2005,1906,2007,210
     8,2008,1908,1808
1290 DATA 1708,1608,1508,1813,1812,1811
     ,1810,1910,2010,2110,2210,2310,241
     0,2411,2412,2413
1300 DATA 2313,2213,2113,2013,1913,1615
     ,1715,1815,1915,2015,2115,2215,161
     6,1617,1618,1718
```

```
1310 DATA 1818,1918,1917,1916,2016,2117
     ,2218,1921,2021,2121,2221,2321,182
     0,1821,1822,1823
1320 DATA 1824,1924,2024,2124,2224,2324
     ,2424,2423,2422,2421,2420,1729,172
     8,1727,1726,1826
1330 DATA 1926,2026,2027,2028,2029,2129
     ,2229,2329,2328,2327,2326
1340 FOR I=1 TO 59
1350 READ SET$(I)
1360 NEXT I
1370 DATA 175,135,144,125,214,211,217,2
     55,257,254,251,312,412,414,452,455
     ,522,577,612,614,654,717,755,712,1
     754
1380 DATA 2125,2514,2556,3165,3175,4514
     ,4516,4554,5475,6135,6145,6554,717
     5,655,31754,456135,453145,3122175
1390 DATA 2177175,21475,14454,4122254,54
     754,2177754,257754,52254,61354,2145
     ,133545,14175,2174l4,45311,613145,5
     66135
1400 FOR I=1 TO 2
1410 FOR I=1 TO 7
1420 FOR K=1 TO 2
1430 READ LINE(J,I,K)
1440 NEXT K
1450 NEXT J
1460 NEXT I
1470 DATA 5,0,-5,0,4,0,-4,0,3,-7,-3,6,-
     6,0,-3,6,-6,0,5,0,-5,0,4,0,-4,0,3,
     -7
1480 CALL CHAR(96,"0406050404040404")
1490 CALL CHAR(40,"")
1500 CALL CHAR(97,"0000000804020101")
1510 CALL CHAR(98,"0101010101010101")
1520 CALL CHAR(99,"FF8040404040404040")
1530 CALL CHAR(100,"FF0808101020408")
1540 CALL CHAR(101,"FF0000000010204080")
1550 CALL CHAR(102,"FF061C6484040404")
1560 CALL CHAR(103,"FF")
1570 CALL CHAR(104,"FF1010101010101")
1580 CALL CHAR(105,"FF0F34448484844")
1590 CALL CHAR(106,"FF00C0201010080808")
1600 CALL CHAR(107,"FF0808040402010")
1610 CALL CHAR(108,"FF8040404040484847F")
1620 CALL CHAR(109,"FF0808081020408")
1630 CALL CHAR(110,"FF0000003040808010")
1640 CALL CHAR(111,"FF00FC03")
1650 CALL CHAR(112,"FF0000003834020201")
1660 CALL CHAR(113,"FF1008040")
1670 CALL CHAR(114,"FF1010101313101")
1680 CALL CHAR(115,"FF0000000000000102")
1690 CALL CHAR(116,"FF2020404080")
1700 CALL CHAR(117,"FF0000000000010608")
1710 CALL CHAR(118,"FF041820C")
1720 CALL CHAR(120,"FF0000000003844822")
1730 CALL CHAR(121,"FF824438")
1740 CALL CHAR(119,"FF3844828282824438")
1750 CALL CHAR(122,"00384482828282443822")
1760 CALL CHAR(123,"0000000000003844822")
1770 CALL CHAR(136,"")
1780 CALL CHAR(137,"3C7EFFFFFFFFFF7E3C")
1790 CALL CHAR(138,"FFFFFFFFFFFFFFFF")
1800 CALL CHAR(128,"3C7EFFFFFFFFFF7E3C")
1810 CALL CHAR(95,"")
1820 FOR I=5 TO 12
1830 CALL COLOR(I,2,16)
1840 NEXT I
1850 IF FLAG=0 THEN 560 ELSE 1860
1860 CALL COLOR(13,5,5)
1870 CALL COLOR(2,7,7)
1880 CALL COLOR(3,2,16)
1890 CALL COLOR(4,2,16)
1900 CALL COLOR(13,15,1)
1910 CALL COLOR(14,15,1)
1920 GOSUB 3830
1930 C=13
1940 TRY=1
1950 FOR I=5 TO 12
1960 CALL COLOR(I,16,16)
1970 NEXT I
1980 CALL HCHAR(2,13,95,13)
1990 FOR I=3 TO 7
2000 CALL HCHAR(I,13,103,13)
2010 NEXT I
2020 CALL HCHAR(9,13,95,13)
2030 IF CLEF<>3 THEN 2080
2040 CALL HCHAR(10,13,95,13)
2050 FOR I=11 TO 15
2060 CALL HCHAR(I,13,103,13)
2070 NEXT I
2080 CALL HCHAR(20,3,95,12)
2090 CALL HCHAR(20,19,95,12)
2100 N=INT(NSET*RND)+1
2110 WORD$=SET$(N)
2120 FOR I=1 TO LEN(WORD$)
2130 N=1
2140 FOR K=1 TO 3
2150 NLINE(K)=0
2160 NEXT K
2170 IF CLEF=2 THEN 2250
2180 FOR J=1 TO 2
2190 L=LINE(VAL(SEG$(WORD$,I,1)),1,J)
2200 IF L=0 THEN 2230
2210 NLINE(N)=L
2220 N=N+1
2230 NEXT J
2240 IF CLEF=1 THEN 2330
2250 FOR J=1 TO 2
2260 L=LINE(VAL(SEG$(WORD$,I,1)),2,J)
2270 IF L=0 THEN 2320
2280 IF CLEF=2 THEN 2300
2290 L=SGN(L)*(ABS(L)+8)
2300 NLINE(N)=L
2310 N=N+1
2320 NEXT J
2330 N=N-1
2340 IF N=1 THEN 2360
2350 N=INT(N*RND)+1
2360 L=NLINE(N)
2370 IF L<0 THEN 2450
2380 IF L=2 THEN 2420
2390 IF L=10 THEN 2420
2400 CALL HCHAR(L,C,119)
2410 GOTO 2430
2420 CALL HCHAR(L,C,122)
2430 C=C+2
2440 GOTO 2530
2450 L=ABS(L)
2460 IF L=3 THEN 2500
2470 IF L=11 THEN 2500
2480 CALL HCHAR(L-1,C,120)
2490 GOTO 2510
2500 CALL HCHAR(L-1,C,123)
2510 CALL HCHAR(L,C,121)
2520 C=C+2
2530 NEXT I
2540 ROW=20
2550 MSG$="READY"
2560 COL=3
2570 GOSUB 4600
2580 COL=19
2590 GOSUB 4600
2600 CALL COLOR(13,15,1)
2610 CALL HCHAR(12,3,137)
2620 CALL HCHAR(12,30,137)
2630 CALL HCHAR(8,3,128)
2640 CALL HCHAR(8,30,128)
2650 CALL COLOR(13,7,1)
2660 CALL SOUND(100,880,1)
2670 FOR I=5 TO 8
2680 CALL COLOR(I,2,16)
2690 NEXT I
2700 CALL KEY(1,KEY1,S)
2710 CALL KEY(2,KEY2,S)
```

```
2720 IF KEY1=19 THEN 2750
2730 IF KEY2=10 THEN 2850
2740 GOTO 2700
2750 CALL COLOR(13,15,1)
2760 CALL HCHAR(8,3,137)
2770 CALL HCHAR(8,30,137)
2780 CALL HCHAR(10,3,128)
2790 CALL HCHAR(10,30,128)
2800 CALL COLOR(13,11,1)
2810 CALL SOUND(100,880,1)
2820 CALL KEY(2,KEY2,S)
2830 IF KEY2<>10 THEN 2820
2840 GOTO 2940
2850 CALL COLOR(13,15,1)
2860 CALL HCHAR(8,3,137)
2870 CALL HCHAR(8,30,137)
2880 CALL HCHAR(10,3,128)
2890 CALL HCHAR(10,30,128)
2900 CALL COLOR(13,11,1)
2910 CALL SOUND(100,880,1)
2920 CALL KEY(1,KEY1,S)
2930 IF KEY1<>19 THEN 2920
2940 CALL COLOR(13,15,1)
2950 CALL HCHAR(10,3,137)
2960 CALL HCHAR(10,30,137)
2970 CALL HCHAR(12,3,128)
2980 CALL HCHAR(12,30,128)
2990 CALL COLOR(13,13,1)
3000 GOSUB 4840
3010 FOR I=9 TO 12
3020 CALL COLOR(I,2,16)
3030 NEXT I
3040 CALL KEY(1,KEY1,S1)
3050 CALL KEY(2,KEY2,S2)
3060 IF S1=-1 THEN 3040
3070 IF S2=-1 THEN 3040
3080 IF KEY1=19 THEN 3110
3090 IF KEY2=10 THEN 3190
3100 GOTO 3040
3110 CALL SOUND(300,-3,1)
3120 FOR M=1 TO 3
3130 CALL COLOR(2,11,11)
3140 CALL COLOR(2,7,7)
3150 NEXT M
3160 COL=3
3170 TEAM$="RED"
3180 GOTO 3260
3190 CALL SOUND(300,-2,1)
3200 FOR M=1 TO 3
3210 CALL COLOR(1,11,11)
3220 CALL COLOR(1,5,1)
3230 NEXT M
3240 COL=19
3250 TEAM$="BLUE"
3260 FOR I=9 TO 12
3270 CALL COLOR(I,16,16)
3280 NEXT I
3290 MSG$="OK_"&TEAM$&"_TEAM"
3300 GOSUB 4580
3310 C=13
3320 N=1
3330 CALL KEY(0,KEY,S)
3340 IF S<>1 THEN 3330
3350 IF KEY<51 THEN 3330
3360 IF KEY>57 THEN 3330
3370 KEY=KEY-50
3380 IF KEY<>VAL(SEG$(WORD$,N,1))THEN 3
700
3390 CALL SOUND(100,880,1)
3400 KEY=KEY+64
3410 CALL HCHAR(9,C,KEY)
3420 IF N=LEN(WORD$)THEN 3460
3430 N=N+1
3440 C=C+2
3450 GOTO 3330
3460 FOR I=9 TO 12
3470 CALL COLOR(I,2,16)
3480 NEXT I
3490 CALL SOUND(100,330,1,392,1,524,1)
3500 CALL SOUND(500,330,1,392,1,524,1)
3510 MSG$="RIGHT"
3520 IF TEAM$="BLUE" THEN 3550
3530 COL=3
3540 GOTO 3560
3550 COL=19
3560 GOSUB 4580
3570 IF TEAM$<>"RED" THEN 3620
3580 RED=RED+1
3590 C=3
3600 SCORE=RED
3610 GOTO 3650
3620 BLUE=BLUE+1
3630 C=29
3640 SCORE=BLUE
3650 MSG$=STR$(SCORE)
3660 GOSUB 4640
3670 FOR I=1 TO 500
3680 NEXT I
3690 GO TO 1930
3700 CALL SOUND(100,110,1)
3710 IF TRY=2 THEN 1930
3720 TRY=TRY+1
3730 CALL HCHAR(9,13,95,13)
3740 IF TEAM$="RED" THEN 3780
3750 TEAM$="RED"
3760 COL=3
3770 GOTO 3800
3780 TEAM$="BLUE"
3790 COL=19
3800 MSG$="YOU_TRY_"&TEAM$
3810 GOSUB 4580
3820 GOTO 3310
3830 CALL SCREEN(2)
3840 FOR I=1 TO 16
3850 CALL VCHAR(1,I,40,24)
3860 NEXT I
3870 FOR I=17 TO 32
3880 CALL VCHAR(1,I,35,24)
3890 NEXT I
3900 FOR I=2 TO 4
3910 CALL HCHAR(I,2,95,4)
3920 CALL HCHAR(I,28,95,4)
3930 NEXT I
3940 FOR I=19 TO 21
3950 CALL HCHAR(I,2,95,14)
3960 CALL HCHAR(I,18,95,14)
3970 NEXT I
3980 CALL HCHAR(1,7,95,20)
3990 CALL HCHAR(2,7,95,20)
4000 FOR I=3 TO 7
4010 CALL HCHAR(I,8,103,19)
4020 NEXT I
4030 FOR I=8 TO 10
4040 CALL HCHAR(I,7,95,20)
4050 NEXT I
4060 CALL HCHAR(7,7,95)
4070 CALL VCHAR(3,7,98,4)
4080 IF CLEF=2 THEN 4110
4090 GOSUB 4340
4100 GOTO 4140
4110 R=3
4120 GOSUB 4480
4130 GOTO 4230
4140 IF CLEF<>3 THEN 4230
4150 FOR I=11 TO 15
4160 CALL HCHAR(I,8,103,19)
4170 NEXT I
4180 CALL HCHAR(16,7,95,20)
4190 CALL VCHAR(7,7,98,8)
4200 CALL HCHAR(15,7,95)
4210 R=11
4220 GOSUB 4480
4230 FOR I=7 TO 13
4240 CALL HCHAR(I,2,136,3)
4250 CALL HCHAR(I,29,136,3)
4260 NEXT I
4270 CALL HCHAR(8,3,128)
4280 CALL HCHAR(8,30,128)
```

```
4290 CALL HCHAR(10,3,137)
4300 CALL HCHAR(10,30,137)
4310 CALL HCHAR(12,3,137)
4320 CALL HCHAR(12,30,137)
4330 RETURN
4340 CALL HCHAR(2,9,96)
4350 CALL HCHAR(2,10,97)
4360 CALL HCHAR(3,9,99)
4370 CALL HCHAR(3,10,100)
4380 CALL HCHAR(4,8,101)
4390 CALL HCHAR(4,9,102)
4400 CALL HCHAR(5,8,104)
4410 CALL HCHAR(5,9,105)
4420 CALL HCHAR(5,10,106)
4430 CALL HCHAR(6,8,107)
4440 CALL HCHAR(6,9,108)
4450 CALL HCHAR(6,10,109)
4460 CALL HCHAR(7,9,99)
4470 RETURN
4480 CALL HCHAR(R,8,110)
4490 CALL HCHAR(R,9,111)
4500 CALL HCHAR(R,10,112)
4510 CALL HCHAR(R+1,8,113)
4520 CALL HCHAR(R+1,10,114)
4530 CALL HCHAR(R+2,9,115)
4540 CALL HCHAR(R+2,10,116)
4550 CALL HCHAR(R+3,8,117)
4560 CALL HCHAR(R+3,9,118)
4570 RETURN
4580 CALL HCHAR(20,3,95,12)
4590 CALL HCHAR(20,19,95,12)
4600 FOR I=0 TO LEN(MSG$)-1
4610 CALL HCHAR(ROW,COL+I,ASC(SEG$(MSG$
     ,I+1,1)))
4620 NEXT I
4630 RETURN
4640 IF LEN(MSG$)=2 THEN 4660
4650 MSG$="_"&MSG$
4660 FOR I=1 TO 2
4670 CALL HCHAR(3,C,ASC(SEG$(MSG$,I,1))
     )
4680 C=C+1
4690 NEXT I
4700 RETURN
4710 FOR MMM=1 TO M
4720 CALL HCHAR(R,6,97)
4730 CALL HCHAR(R-1,6,96)
4740 CALL SOUND(50,-7,5)
4750 CALL SOUND(MM-100,110,30)
4760 CALL HCHAR(R-2,8,97)
```

```
4770 CALL HCHAR(R-3,8,96)
4780 CALL SOUND(50,-7,5)
4790 CALL SOUND(MM-100,110,30)
4800 R=R-4
4810 NEXT MMM
4820 CALL SOUND(200,110,30)
4830 RETURN
4840 CALL SOUND(675,262,1)
4850 CALL SOUND(225,294,1)
4860 CALL SOUND(225,311,1)
4870 CALL SOUND(225,110,30)
4880 CALL SOUND(225,262,1)
4890 CALL SOUND(225,110,30)
4900 RETURN
4910 FOR J=12 TO 20 STEP 2
4920 CALL HCHAR(J,2,119,30)
4930 NEXT J
4940 FOR I=F1 TO F1+8 STEP 2
4950 READ MSG$
4960 FOR J=0 TO LEN(MSG$)-1
4970 CALL HCHAR(S1-I,I+J,ASC(SEG$(MSG$,
     J+1,1)))
4980 NEXT J
4990 NEXT I
5000 FOR I=F2 TO F2+6 STEP 2
5010 READ MSG$
5020 FOR J=0 TO LEN(MSG$)-1
5030 CALL HCHAR(S2-I,I+J,ASC(SEG$(MSG$,
     J+1,1)))
5040 NEXT J
5050 NEXT I
5060 MSG$="PRESS ANY KEY"
5070 GOSUB 4600
5080 CALL KEY(0,K,S)
5090 IF S=0 THEN 5080
5100 RETURN
5110 DATA ".E-EVERY..G-GOOD..
     B-BOY..D-DOES..F-FINE
     "
5120 DATA F,A,C,E
5130 DATA "G-GOOD..B-BOYS..D
     -DO..F-FINE..A-ALWAYS
     "
5140 DATA "A-ALL..C-COWS..E
     -EAT..G-GRASS"
5150 FOR I=3 TO 30
5160 CALL VCHAR(1,I,32,24)
5170 NEXT I
5180 RETURN
```



Notes on a Computer Score: Part 2

- The TI-99/4 Assists Gifted Children in the Learning Process

Although the TI-99/4 proved itself a valuable enrichment tool in my traditional music classes, I began to realize its full potential during a summer program for gifted children at New Horizons Academy. It was exciting to let a curriculum evolve as children enthusiastically identified their own interests and pursued ways of expressing them creatively through use of the computer.

The Educational Setting

New Horizons Academy is a private school that was founded by Nanci Lucas in the belief that children are naturally excited about learning and capable of handling academic pursuits beyond their years. When an individualized curriculum is designed to allow for advancement through basic skills and extensive opportunity for enrichment and acceleration, children find learning to be exciting and meaningful.

In addition to the regular academic curriculum, the Academy periodically provides workshops that are open to any interested children. Since 1978, "Summer Spectacular" has offered courses in computers, creative dramatics, archaeology and photography. I became involved with New Horizons last summer when I taught two sessions in "Computer Music" with our TI-99/4.

The Computer Music classes were intended to familiarize students with basic music concepts and provide for individualized and accelerated learning in a manner consistent with the philosophy of the Academy. A typical group consisted of eight students ranging in age from 7-13. The group was scheduled to meet eight times in a two week period with each session lasting one hour. After the first day, however, the students "demanded" that I arrive at 8:45 A.M. and not leave until 2:15 P.M. Several times it was [illegible] students asking to cut other classes, skip lunch, and having their parents pick them up late—all just so they could work on their projects—was tremendously rewarding.

The Educational Mode

I employed Renzulli's theoretical framework, *The Enrichment Triad Model*.[1] The model contains three constructs which convey the types of learning activity believed to be best for gifted children.

In *General Exploratory Activities* (Type I) students are exposed to a broad range of possibilities. None of these is presented in detail. The purpose is merely to introduce the students to the range of possible alternatives open to them. *Group Training Activities* (Type II) follow, providing the students with fundamental information of potential use in subsequent development of their interest areas. These activities are oriented toward content.

During the two preceding phases, students begin to identify their interests and develop the skills to create a final product. In *Individual and Small Group Investigations of Real Problems* (Type III), each student determines a problem or project of particular interest that is based on the information obtained in the previous activities, and then pursues that choice in greater depth.

This final element of the Enrichment Triad is perhaps the most important. Ideally, the students will exemplify the "turned-on professional" and pursue their objectives with intense motivation and commitment.

In many respects, Renzulli's model parallels Seymour Papert's principles of continuity, power, and cultural resonance.[2]

Implementation of the Model

In the first few days, I exposed the students to a variety of musical activities including a TI-99/4 concert of familiar children's songs such as "Happy Birthday," "Yankee Doodle," and "Pop Goes the Weasel"—all complete with graphics. We also played rhythm instruments, the autoharp, resonator bells and recorders, drew on impressions of music while listening to recordings, identified environmental sounds, and discussed the commonalities and differences of all sounds. [illegible] of music notation and theory. In this mode the Texas Instruments *Music Maker* Command Cartridge contains two options with which children can write music. In the exploratory activities, they utilized the Sound Graphs option, in which "the composer" need not have any prior knowledge of music notation and theory. In this mode the students experimented with duration of notes by con-

trolling the length of the line in whatever voice they were composing (3 voices or 3-part harmony is possible). Frequency is determined by the height of the line on the screen, and there is a volume choice. From all of our exploratory activities, students came to the conclusion that all sounds have duration, frequency, and volume in common. These concepts were effectively and concretely exemplified by the *Music Maker*'s Sound Graphs Mode.

In summary, I believe that the exploratory activities altered the students' *experience* of music. They began to see music in a new way, as part of the continuum of sound and noise; the "freshnesss" of this new perspective contributed to their desire to move toward the next phases of the model.

Group Training Activities were concerned with content-oriented learning. The objectives were to provide students with a basic knowledge of music theory and an understanding of how a computer program is written—information which they could use as tools in developing their interest areas. Several computer music games and drills were used by the entire group, but as a child's interest waned, he was allowed to break away from the group activity and pursue individual work in his primary interest area.

The computer games included (1) *Mystery Words*, in which the players learned the names of treble and bass clef notes; (2) *Rhythm*, which provided ear training in the recognition of quarter notes, eighth notes, and quarter rest patterns; and (3) TI's *Music Skills Trainer*, which contains four games to improve the player's skills in recognition and recall of pitches, intervals, chords, and phrases played by the computer.

We took a look at the program listings for our favorite songs and games and "brainstormed" about what all those commands could possibly mean. Discovering how changing the duration, frequency, and/or volume in a CALL SOUND statement affects the tones produced by the computer was a popular Group Training Activity. The children soon started drawing conclusions and generalizations about how to program. At this point it was necessary to hand out information from the User's Manual for students to take home and study over the weekend—homework at their request!

Additional content-oriented learning took place when students experimented with the Traditional Mode of the computer's *Music Maker* Command Cartridge. I used the traditional mode to help children discover information about key signatures, time signatures, tempo, and music notation—including how various notes such as whole, half, quarter, eighth and sixteenth and their corresponding rests relate to each other and can be organized into a composition that is musically correct.

In moving into Type III of Renzulli's model, almost all students elected to write a computer program to play a musical composition; some students selected pieces with which they were already familar, and others wrote original compositions. Many investigated how to use graphics and color to enhance their creations, and designed title screens to be displayed during the computer's performances of their works. Compositions included "The Entertainer," "Mr. Tambourine Man," "Amazing Grace," Beethoven's Ninth Symphony (*Ode to Joy*), and "Jingle Bells." Three students, Byron, Allan, and Peggy, exhibited a competitive spirit when comparing the number of lines and difficulty of their programs. Steve wrote his program to play Beethoven's Ninth Symphony in three-part harmony; Bryan wrote his original composition to flash a change of screen color to emphasize musical contrasts at appropriate points in the music; Peggy reworked her original composition many times until she was satisfied with the rhythmic structure .

It is important to note that not every student was equally enthusiastic about programming. For example, Adrienne seemed to prefer taking Computer Music for enjoyment and the personal satisfaction of becoming familiar with it but did not have a genuine interest in becoming a creative producer.

"Seeing your students eagerly asking to cut other classes . .,. and having their parents pick them up late — all just so they could work on their projects — was tremendously rewarding."

The satisfaction children feel from communicating the results of their work to an audience was obvious when three ladies from Springfield (Ohio) School District and a banker visited our classroom one day. Byron stopped his work to take over for Mrs. Lucas and me as we were explaining the Computer Music Class. He and two others enthusiastically gave a presentation of their music and proudly explained what had gone into its composition. In addition, the already high level of enthusiam increased when the class found out they could present their finished products to their parents and others on Visitation Day and possibly have them published in *99'er Magazine*.

With this group, I served mainly as a resource person and passed the responsibility for learning and investigating on to the students. Students were introduced to concepts of programming and music theory as they explored and used this knowledge to make their programs more complex. It was a good example of making material relevant. The Computer Music course allowed for freedom of choice in that no course requirements were established ahead of time; instead,

study

the class members were allowed to devolop their own "courses of study" as their interests developed. Likewise, the time allocations were flexible because the entire staff allowed students to skip their classes and come to Computer Music all day if they wanted.

By requiring students to play the games described in Type II activities only until they no longer were interested, mastery of competencies became more streamlined and exciting—as Renzulli suggests. After playing *Mystery Words* about ten minutes, seven-year-old Michael put it this way, "Do we have to play it anymore? We know this now!"

It was interesting to observe how the gifted children mastered the basics much more rapidly and efficiently than my regular general music students. The need for individualization and enrichment for the gifted is obvious when one realizes that playing the same games which intrigued my regular classes for several of their thirty minute music periods tired the gifted in ten to fifteen minutes; they then requested permission to return to programming their *own* creations. This is an example of Renzulli's differentiation of "real investigative activities" for the gifted from "training exercises." This differentiation prompted the development of his Enrichment Triad Model.

It is important to point out that Renzulli's model is not a fixed, rigid framework; the three activity types often overlap. For example, while the actual composing of music is Type III (Individual and Small Group Investigations), the trial-and-error initial discoveries of the computer's musical capabilities might be considered Type I (Exploratory). Likewise, there is overlap in some of the students' other Type II activities. Since brainstorming provides children with the skills needed to explore alternative solutions to problems, our discussions about environmental sounds and computer language were exercises in developing the processes that enable a learner to deal more effectively with content, yet took place during an exploratory activity.

Furthermore, process training (Type II) occurred when students wrote their own programs. As Byron observed, "The computer really programs you; you don't program it." He was referring to the fact that the best programmers learn to think like the computer, in that they think out the process of writing their programs instead of memorizing what to write. Essentially, they must consider how the computer thinks, then determine what to say to get the computer to accomplish their goals. Obviously, the programming required to achieve the final product is a Type III activity, yet the development of thinking processes involved in Type II is also present.

The experience of teaching at New Horizons has given me new insights into how children think and how different their learning styles can be. It was exciting to allow a curriculum to evolve as children enthusiastically identified their own interests and pursued ways of expressing these interests creatively.

Postscript

Although the Academy already had four CBM computers and a TRS-80, the magical attraction of students to our TI-99/4 did not go unrecognized. Soon the Academy purchased a TI-99/4A, and subsequently a second one, together with a variety of the high quality educational software offered by Texas Instruments.

My husband and I have conducted several other enrichment sessions at the Academy and have become increasingly excited about the profound potential of computer-facilitated learning.

References

1 Renzulli, Joseph S. *The Enrichment Triad Model: A Guide for Developing Defensible Programs for the Gifted and Talented.* Connecticut: Creative Learning Press, Inc., 1977.

2 Papert, Seymour. *Mindstorms: Children, Computers, and Powerful Ideas.* New York: Basic Books, Inc., 1980.

A Music Text Editor & File Player for the TI-99/4A

TI BASIC

For those readers who do not as yet have a *Music Maker* but would like to experiment with music writing anyway, we are offering a primitive music text editor, which has two of the three voices of the TI99/4A, as well as a file player program and the input required to play.

The *Music Text Editor* program creates a tape file which is read and played by the *Music File Player* that follows it. Although the file can be played by the editor, the tempo will be somewhat slower than when performed by the separate player program.

Use the following symbols to write note values:

W-Whole	H-Half
Q-Quarter	E-Eighth
S-Sixteenth	

For a dotted note value, put a period after the symbols, for example, S.,Q., etc.

Use the following symbols for pitches:

A A#	E
B	F F#
C C#	G G#
D D#	R-Rest (a pitch with value 0)

After each pitch, give the octave (1-4):

Octave	Begins
1	Bass clef, bottom space (A = 110 cps)
2	Bass clef, top line (A = 220 cps)
3	Treble clef, 2nd space from bottom (A = 440 cps)
4	Treble clef, 1st ledger line above (A = 880 cps)

[**NOTE:** *SAVE your input periodically! An input of any note plus an accidental but* ***without*** *an octave number (e.g., A#) may cause your program to crash.]*

The *Music Text Editor* will first ask you for the composition's and the composer's names. Then the prompt M.M will ask you for a tempo in quarter notes per minute—corresponding approximately to a metronome beat—between 56 and 126 per minute. (NOTE: The program won't accept any value larger than 126; the computer will reject an overly-funereal beat because the value put in the duration parameter of CALL SOUND will be too large.)

The program then begins requesting input for the composition, line by line for the two voices. After the program displays the line number, you enter the duration (W, Q., S, . . .), followed by the pitch values and octave ranges for each of the two voices. You must separate these values by a slash (/), and end the line with a slash. For instance, to play simultaneously the dotted eighth notes **F♯** in octave 2 and **C** in octave 1, followed by a sixteenth rest in the first voice and a **B♭** sixteenth note in the second, after the first program line number prompts (1 =) you would enter:

```
1 = E./F#2/C1/
2 = S/R/A#1/
3 =
```

When the program prompts you for the next line of notes (in this instance line 3), you may instead enter one of the following commands:

CHANGE, REDO, LIST, PLAY, or SAVE.

CHANGE, REDO and LIST will ask for a range of lines. The SAVE command merges new data with data already stored in the tape file. Unless you answer the question, "FINISHED?" with a "YES", no end-of-file mark will be written on the tape file. Until the file has an end-of-file mark, it can't be read. The *Music File Player* can read and play a file consisting of up to 550 lines.

```
120 REM  *  MUSIC TEXT EDITOR  *
160 REM
170 REM
180 CALL CLEAR
190 DIM N$(11,4),V$(8,1),P(349,2)
200 INPUT "TITLE          ":T$
210 INPUT "COMPOSER       ":C$
220 FOR I=0 TO 11
230 READ N$(I,0)
240 NEXT I
250 DATA A,A#,B,C,C#,D,D#,E,F,F#,G,G#
260 FOR I=1 TO 4
270 FOR J=0 TO 11
280 X=110*2^(S/12)
290 GOSUB 1420
300 N$(J,I)=STR$(X)
310 S=S+1
320 NEXT J
330 NEXT I
340 CALL CLEAR
350 PRINT "ENTER TEMPO (M.M.)":::"QUARTE
    R NOTES/MINUTE":::
360 INPUT "M.M = ":T
370 IF T<127 THEN 410
380 PRINT :"MAXIMUM IS 126":::
390 CALL SOUND(150,220,1)
400 GOTO 360
410 CALL CLEAR
420 T=6E+4/T
430 FOR I=0 TO 8
440 FOR J=0 TO 1
450 READ V$(I,J)
460 NEXT J
```

```
470 NEXT I
480 DATA W,4,H,2,H.,3,Q,1.1,Q..1.5,E,.
    5,E...75,S,.25,S..,.375
490 FOR I=0 TO 8
500 X=VAL(V$(I,1))
510 X=T*X
520 GOSUB 1420
530 V$(I,1)=STR$(X)
540 NEXT I
550 CALL CLEAR
560 KP=0
570 GOSUB 1330
580 PO=1
590 X$="."
600 FOR N=1 TO 4-LEN(STR$(K+KP+1))
610 X$=X$&"."
620 NEXT N
630 INPUT STR$(K+KP+1)&X$&" = ":X$
640 IF K=350 THEN 670
650 P(K,1)=111
660 P(K,2)=111
670 IF X$<>"CHANGE" THEN 700
680 GOSUB 1570
690 GOTO 580
700 IF X$<>"PLAY" THEN 730
710 GOSUB 1200
720 GOTO 580
730 IF X$<>"REDO" THEN 760
740 GOSUB 1470
750 GOTO 580
760 IF X$<>"LIST" THEN 790
770 GOSUB 1710
780 GOTO 580
790 IF X$<>"SAVE" THEN 810
800 GOSUB 2160
810 IF K<>350 THEN 840
820 PRINT "SAVE FILE BEFORE PROCEEDING
    ":::
830 GOTO 580
840 GOSUB 1370
850 IF S$="" THEN 870
860 IF PN<>0 THEN 890
870 CALL SOUND(150,220,1)
880 GOTO 580
890 FOR I=0 TO 8
900 IF S$=V$(I,0)THEN 950
910 NEXT I
920 PRINT "ERR- VALUE"
930 CALL SOUND(150,220,1)
940 GOTO 580
950 P(K,0)=VAL(V$(I,1))
960 FOR I=1 TO 2
970 GOSUB 1370
980 IF S$="" THEN 1000
990 IF PN<>0 THEN 1020
1000 CALL SOUND(150,220,1)
1010 GOTO 580
1020 IF S$="R" THEN 1150
1030 IF SEG$(S$,LEN(S$),1)<"5" THEN 106
     0
1040 CALL SOUND(150,220,1)
1050 GOTO 580
1060 OC=VAL(SEG$(S$,LEN(S$),1))
1070 S$=SEG$(S$,1,LEN(S$)-1)
1080 FOR J=0 TO 11
1090 IF S$=N$(J,0)THEN 1140
1100 NEXT J
1110 PRINT "ERR- VOICE";I
1120 CALL SOUND(150,220,1)
1130 GOTO 580
1140 P(K,I)=VAL(N$(J,OC))
1150 NEXT I
1160 IF SR=0 THEN 1180
1170 RETURN
1180 K=K+1
1190 GOTO 580
1200 FOR L=0 TO K-1
1210 IF P(L,1)<>111 THEN 1270
1220 IF P(L,2)<>111 THEN 1250
1230 CALL SOUND(P(L,0),110,30,110,30)
1240 GOTO 1310
1250 CALL SOUND(P(L,0),110,30,P(L,2),1)
1260 GOTO 1310
1270 IF P(L,2)<>111 THEN 1300
1280 CALL SOUND(P(L,0),P(L,1),1,110,30)
1290 GOTO 1310
1300 CALL SOUND(P(L,0),P(L,1),1,P(L,2),
     1)
1310 NEXT L
1320 RETURN
1330 FOR I=0 TO 349
1340 P(I,0)=0
1350 NEXT I
1360 RETURN
1370 PN=POS(X$,"/",PO)
1380 IF PN=0 THEN 1410
1390 S$=SEG$(X$,PO,PN-PO)
1400 PO=PN+1
1410 RETURN
1420 Y=X-INT(X)
1430 IF Y<.5 THEN 1450
1440 X=X+1
1450 X=INT(X)
1460 RETURN
1470 M=K
1480 INPUT "START AT LINE - ":K
1490 IF K>KP THEN 1520
1500 CALL SOUND(150,220,1)
1510 GOTO 1480
1520 K=K-KP-1
1530 IF K<M+1 THEN 1560
1540 CALL SOUND(150,220,1)
1550 GOTO 1480
1560 RETURN
1570 SR=1
1580 M=K
1590 INPUT "CHANGE LINE - ":K
1600 IF K>KP THEN 1630
1610 CALL SOUND(150,220,1)
1620 GOTO 1590
1630 K=K-KP-1
1640 IF K<M THEN 1670
1650 CALL SOUND(150,220,1)
1660 GOTO 1590
1670 GOSUB 580
1680 K=M
1690 SR=0
1700 RETURN
1710 INPUT "FIRST LINE - ":O
1720 IF O>KP THEN 1750
1730 CALL SOUND(150,220,1)
1740 GOTO 1710
1750 INPUT "LAST LINE - ":Q
1760 IF Q>KP THEN 1790
1770 CALL SOUND(150,220,1)
1780 GOTO 1750
1790 IF O<=Q THEN 1820
1800 CALL SOUND(150,220,1)
1810 GOTO 1710
1820 IF Q-KP<K+1 THEN 1840
1830 Q=K+KP
1840 PRINT ::
1850 O=O-KP-1
1860 Q=Q-KP-1
1870 FOR R=O TO Q
1880 X$=""
1890 FOR I=1 TO 4-LEN(STR$(R+1))
1900 X$=X$&"."
1910 NEXT I
1920 PRINT STR$(R+1)&X$&" - ";
1930 FOR S=0 TO 8
1940 IF P(R,0)=VAL(V$(S,1))THEN 1960
1950 NEXT S
1960 PRINT V$(S,0)&"/";
1970 FOR I=1 TO 2
1980 IF P(R,I)<>111 THEN 2040
1990 IF I<>2 THEN 2020
2000 PRINT "R/"
```

```
2010 GOTO 2140
2020 PRINT "R/";
2030 GOTO 2120
2040 FOR I1=0 TO 11
2050 FOR I2=1 TO 4
2060 IF P(R,I)<VAL(N$(I1,I2))THEN 2090
2070 IF P(R,I)=VAL(N$(I1,I2))THEN 2100
2080 NEXT I2
2090 NEXT I1
2100 IF I=2 THEN 2130
2110 PRINT N$(I1,0)&STR$(I2)&"/";
2120 NEXT I
2130 PRINT N$(I1,0)&STR$(I2)&"/"
2140 NEXT R
2150 RETURN
2160 IF F$="1" THEN 2200
2170 F$="1"
2180 OPEN #1:"CS1",SEQUENTIAL,INTERNAL,
     OUTPUT,FIXED 192
2190 PRINT #1:T$;C$
2200 I1=0
2210 K=K-1
2220 FOR I=1 TO 12
```

```
2230 FOR J=0 TO 2
2240 PRINT #1:STR$(P(I1,I)),
2250 NEXT J
2260 I1=I1+1
2270 IF I1>K THEN 2330
2280 NEXT I
2290 PRINT #1:STR$(P(I1,0)),STR$(P(I1,1
     )),STR$(P(I1,2))
2300 I1=I1+1
2310 IF I1>K THEN 2340
2320 GOTO 2220
2330 PRINT #1:"-"
2340 PRINT "FINISHED? (Y/N)"
2350 CALL KEY(0,KEY,STATUS)
2360 IF KEY=89 THEN 2410
2370 IF KEY<>78 THEN 2350
2380 KP=KP+K+1
2390 K=0
2400 GOTO 570
2410 PRINT #1:"@@@@@"
2420 CLOSE #1
2430 PRINT T$&" SAVED"
2440 END
```

99'er

```
 120 REM  * MUSIC FILE PLAYER *
 160 REM
 170 REM
 180 CALL CLEAR
 190 PRINT "THIS PROGRAM PLAYS A MUSIC
     FILE CREATED BY THE 2-VOICE EDITO
     R."
 200 PRINT :"* PLACE DATA IN CASSETTE C
     S1":"THEN PRESS ENTER"
 210 CALL SOUND(150,1400,1)
 220 CALL KEY(0,KEY,STATUS)
 230 IF KEY<>13 THEN 220
 240 DIM P(530,2)
 250 OPEN #1:"CS1",SEQUENTIAL,INTERNAL,
     INPUT ,FIXED 192
 260 PRINT :"* READING"
 270 CALL SOUND(150,1400,1)
 280 INPUT #1:N$,C$
 290 FOR I=0 TO 530
 300 FOR J=0 TO 2
 310 INPUT #1:X$,
 320 IF X$="" THEN 310
 330 IF X$="@@@@@" THEN 370
 340 P(I,J)=VAL(X$)
 350 NEXT J
 360 NEXT I
 370 CALL CLEAR
```

```
 380 PRINT TAB((28-LEN(N$))/2);N$::TAB(
     13);"BY"::TAB((28-LEN(C$))/2);C$::
     :::::::::
 390 FOR K=0 TO I-1
 400 IF P(K,1)<>111 THEN 440
 410 IF P(K,2)<>111 THEN 470
 420 CALL SOUND(P(K,0),110,30,110,30)
 430 GOTO 500
 440 IF P(K,2)<>111 THEN 490
 450 CALL SOUND(P(K,0),P(K,1),1,110,30)
 460 GOTO 500
 470 CALL SOUND(P(K,0),110,30,P(K,2),1)
 480 GOTO 500
 490 CALL SOUND(P(K,0),P(K,1),1,P(K,2),
     1)
 500 NEXT K
 510 CALL CLEAR
 520 PRINT "SELECT FROM...":::"(1) PLAY
     IT AGAIN":::"(2) NEXT PIECE":::"(3)
     STOP"
 530 CALL KEY(0,KEY,STATUS)
 540 IF KEY<49 THEN 530
 550 IF KEY>51 THEN 530
 560 KEY=KEY-48
 570 ON KEY GOTO 370,280,580
 580 CLOSE #1
 590 STOP
```

99'er

Fugue in G minor

MM = 100 **by Bach**

1 - Q/G2/R/
2 - Q/D3/R/
3 - Q/A#3/R/
4 - E/A3/R/
5 - E/G2/R/
6 - E/A#3/R/
7 - E/A3/R/
8 - E/G2/R/
9 - E/F#2/R/
10 - E/A3/R/
11 - Q/D2/R/
12 - E/G2/R/
13 - E/D2/R/
14 - E/A3/R/
15 - E/D2/R/
16 - E/A#3/R/
17 - S/A3/R/
18 - S/G2/R/
19 - E/A3/R/
20 - E/D2/R/
21 - E/G2/R/
22 - S/D2/R/
23 - S/G2/R/
24 - E/A3/R/
25 - S/D2/R/
26 - S/A3/R/
27 - E/A#3/R/
28 - S/A3/R/
29 - S/G2/R/
30 - S/A3/R/
31 - S/D2/R/
32 - S/D3/R/
33 - S/C3/R/
34 - S/A#3/R/
35 - S/A3/R/
36 - S/G2/R/
37 - S/A#3/R/
38 - S/A3/R/
39 - S/G2/R/
40 - S/F#2/R/
41 - S/A3/R/
42 - S/G2/R/
43 - S/D2/R/
44 - S/G2/R/
45 - S/A3/R/
46 - S/A#3/R/
47 - S/C3/R/
48 - S/D3/R/
49 - S/E3/R/
50 - S/F3/D2/
51 - S/E3/D2/
52 - S/D3/R/
53 - S/F3/R/
54 - S/E3/A3/
55 - S/D3/A3/
56 - S/C#3/R/
57 - S/E3/R/
58 - S/D3/F2/
59 - E/A3/F2/
60 - E/D3/F2/
61 - S/E3/E2/
62 - S/E3/E2/
63 - S/F3/D2/
64 - S/G3/D2/
65 - S/F3/F2/
66 - S/G3/F2/
67 - E/G3/E2/
68 - E/G3/E2/
69 - E/G3/D2/
70 - S/F3/D2/
71 - S/G3/D2/
72 - S/A4/C#2/
73 - S/G3/C#2/
74 - S/A4/C#2/
75 - S/A#4/E2/
76 - S/A4/E2/
77 - S/G3/E2/
78 - S/F3/A2/
79 - S/E3/A2/
80 - S/F3/D2/
81 - S/A4/A2/
82 - S/G3/A2/
83 - S/A4/A2/
84 - S/C#3/E2/
85 - S/A4/E2/
86 - S/G3/A2/
87 - S/A4/A2/
88 - S/D3/F2/
89 - S/A4/F2/
90 - S/G3/E2/
91 - S/A4/D2/
92 - S/C#3/E2/
93 - S/A4/E2/
94 - S/G3/E2/
95 - S/A4/A2/
96 - S/F3/D2/
97 - S/D3/D2/
98 - S/C#3/A2/
99 - S/D3/D2/
100 - S/G3/E2/
101 - S/D3/A2/
102 - S/C#3/A2/
103 - S/D3/E2/
104 - S/A4/F2/
105 - S/D3/F2/
106 - S/C#3/E2/
107 - S/D3/D2/
108 - S/G3/E2/
109 - S/D3/A2/
110 - S/C#3/A3/
111 - S/D3/G2/
112 - E/A3/F2/
113 - E/F3/D2/
114 - E/G2/E2/
115 - E/E3/C#2/
116 - E/F2/D2/
117 - E/A3/D2/
118 - E/D3/F2/
119 - S/F3/A3/
120 - S/F3/B3/
121 - E/D#3/C3/
122 - E/A4/C3/
123 - E/A4/D3/
124 - E/D#3/A#3/
125 - S/D3/C3/
126 - E/D3/A#3/
127 - E/G3/A#3/
128 - E/G3/A#3/
129 - S/D3/A3/
130 - E/D3/G2/
131 - S/R/A#3/
132 - S/C3/A3/
133 - S/A#3/A3/
134 - S/C3/G2/
135 - S/D3/G2/
136 - S/C3/F#2/
137 - S/A4/F#2/
138 - S/G3/D2/
139 - S/A4/D2/
140 - S/A#3/G2/
141 - S/G3/R/
142 - S/F#3/R/
143 - S/G3/R/
144 - S/A3/D2/
145 - S/F3/R/
146 - S/E3/R/
147 - S/F3/R/
148 - E/G1/A#2/
149 - E/D#3/R/
150 - E/G2/R/
151 - E/A2/R/
152 - E/A#2/E1/
153 - E/R/D1/
154 - E/R/F1
155 - E/R/G1
156 - Q/D1/R/
157 - Q/A2/R/
158 - Q/F1/R/
159 - E/E1/R/
160 - E/D1/R/
161 - E/F1/R/
162 - E/E1/R/
163 - E/D1/R/
164 - E/C#1/A2/
165 - E/E1/A#2/
166 - Q/A1/R/
167 - S/R/F2/
168 - S/R/E2/
169 - E/D1/F2/
170 - E/A1/R/
171 - E/E1/C#2/
172 - E/F1/D2/
173 - Q/A1/D2/

99'er

Music Maker

A TI Command Cartridge

To paraphrase Shakespeare, "The computer that hath no music in its chips, nor is not programm'd with concord of sweet sound is fit but for business, mathematics, and sorts." The TI-99/4A is definitely not one of these.

Outstanding music and sound effects capabilities are among the many features which set the TI-99/4A apart from other personal computers. A user can generate three simultaneous tones and a noise, and can specify their duration, pitch, and loudness—all with a single TI BASIC statement. The sound is played through the speaker of the color monitor or TV display.

Of course, an assortment of beeps, "ta-daas" and outer space sounds can greatly enhance a graphics presentation and provide useful auditory feedback during the program execution. But when the sophisticated sound capabilities of the TI-99/4A become the focus of the programmer's attention, the Texas Instruments machine becomes a musical instrument in its own right. Whether playing a Bach sonata or your own composition, a successful TI-99/4A performance is worth the programming effort.

With the introduction of TI's *Music Maker* Command Cartridge, you can take full advantage of the TI-99/4A's sound capabilities without having to write a complex BASIC program. The *Music Maker* allows you to write a composition using either of two methods—Traditional Mode or Sound Graphs. While Traditional Mode requires some knowledge of fundamental music theory, Sound Graphs does not. Both methods are graphics-based, in contrast to other music editor formats which require entry of notes using ASCII characters. Both also make superb use of the TI-99/4A's outstanding color graphics capabilities. Notes are entered by manipulating the cursor with either the joysticks or the arrow keys. A composer can then print out the bass and treble clefs of each measure—complete with all notes, sharps, flats, and rests—with TI's thermal printer (using its special graphic character set). It's also possible to save the completed musical score on cassette tape or diskette.

Traditional Mode

In Traditional Mode, notes are entered directly on the music staff using standard notation. The first step involves defining the key, meter, and tempo. All possible key signatures (0-7 sharps or flats) are allowed. The meter or time signature options for the denominator are 1, 2, 4, 8, and 16—corresponding to the unit of measure receiving one beat (i.e., whole, half, quarter, eighth or sixteenth note). The numerator of the time signature indicates the number of such units which comprise a measure. Your options here are restricted to values equal to, or less than, the denominator. Examples of allowable time signatures are:

$$\frac{4}{4}, \frac{6}{8}, \frac{2}{2}, \text{ and } \frac{3}{4};$$

on the other hand,

$$\frac{3}{2}, \frac{12}{8}, \text{ and } \frac{5}{4}$$

are not allowable, because the numerator exceeds the

denominator. This limitation is significant, because there is a natural accent which falls on the first beat of every measure when music is accurately interpreted by a performer. This regular impulse, together with phrasing and secondary accents in compound meter, gives a composition its underlying rhythmic structure. The *Music Maker* does not automatically provide for this natural rhythm. The implementation of accent is entirely up to you. For example, a composition written in 4/4 time may be made to sound like 3/2 time with proper phrasing and specification of accent. Therefore, the time signature limitation does not actually limit the music you can write with the cartridge. Finally tempo is specified as a number from 1 to 30, corresponding approximately to metronomic indications from 25 to 128 quarter notes per minute—sufficient range for nearly all compositions.

After these parameters are defined, the graphics representation for the first measure appears. Some music editors for other machines do not use graphics at all. It is a great advantage to see your composition displayed in standard notation as you are writing it.

Up to three voices may be "drawn" using whole, half, quarter, eighth and sixteenth notes and their corresponding rests. Single dotting can be used with notes, but not with rests. The notes for each voice are represented in a different color, which facilitates identification of voices when editing.

The pitch range is three octaves, extending from the second **A** below middle **c** (bottom space of bass clef) to the second **a** above middle **c** (first ledger line above treble clef). This may seem like a wide range. In arranging several piano pieces for the TI-99/4A, however, we found that it was frequently necessary to make octave transpositions for notes extending beyond the *Music Maker's* pitch range in Traditional Mode. On the other hand, the *Music Maker* is not really intended for the transcription of existing music for other instruments, but rather to facilitate original composition. Like all instruments, it too has limitations which must be taken into account when preparing an original composition.

Accidentals (sharps, flats, or naturals different from the key signature) must be written for each note; once written, they do not carry over through the entire measure as they do in standard notation. For someone who is accustomed to standard notation, this may take a little getting used to. Additionally, the large and legible graphic symbols that the cursor picks up from the menu become too small to be easily read when placed beside a note.

Graphics

Graphics characters for the notes themselves resemble square notation, but we do not feel this detracts from their readability (especially when compared with the legibility of many manuscripts). However, in drawing clusters of two or more notes, we encountered a peculiar graphics-related difficulty. This is a function of the position (up or down) of an existing note stem. You will find that a note for one voice can not be placed at a pitch immediately above or below an existing note if that pitch is occupied by the stem of the existing note. The stems for voices one and two go upward unless they are placed immediately below a note in another voice which has its stem going upward. The opposite is true of voice-three notes. This means that while it is possible to represent any two-note cluster, the process can be more involved than would seem necessary. For instance, suppose you have already written a voice-one quarter note at middle **c**, and you want to write a voice-two note at **d** immediately above it. Finding that you cannot do this simply, you would have to do the following: Change voices, erase the **c**, change voices, draw the **d** (voice-two), change voices, redraw the **c** (stem down), and finally change back to voice-two to continue. A cluster of three notes with adjacent pitches cannot be written at all. These problems will be troublesome only in the event that the composer wants to write dissonant chords in the form of clusters.

At the bottom of the display is a double row of squares; the upper row is used to specify volume for each note. There are eight levels of volume which allow a very smooth crescendo or diminuendo without abrupt transitions from one level to the next. Some other music editors do not allow this degree of versatility in dynamics. This default value for loudness is the maximum level of eight. If you want to accent selected notes, say the first note of every measure, you must drop the volume of all other notes. A default loudness of six or seven might have been a little easier to use in this regard.

The bottom row of squares is used to indicate the width of each note; this is very helpful in positioning them. It also allows one to create rests without using rest graphics by simply leaving a gap between one note and the next. Two adjacent notes of the same pitch are automatically tied. The only way to articulate them is to leave a gap in between. For instance, one might write a dotted quarter rather than a quarter note, and the resulting gap would then prevent a tie with the next note.

At any point during the writing of a measure, you can play an individual voice or all voices. If you decide to make a change, this is easily accomplished by erasing an individual note or the entire voice. You cannot, however, insert or delete notes without making necessary adjustments to other notes in the measure.

Repetition is easily handled by copying an individual voice or all voices from a previous measure, and this can save a great deal of time. A given voice cannot, however, be copied as another voice. So if you want to use the copy feature to write rounds, they have to be scored differently than they would be in traditional composition. Any two voices can be copied by copying all three and then erasing the one which is not wanted.

When you are finished with a measure, you can either go on to the next measure or back to a menu which allows you to edit, play, save, or print your composition. If you choose to edit, you will be shown the number of measures completed and the percentage of file space used, and you will be given the option of changing the tempo. To play the composition, you specify which voices are to be played, and you are given the option of hearing the music transposed up or down by as many as eleven half steps (twelve half steps are an octave). If you transpose a note so that it falls *below* the *Music Maker's* range, it will not be played. You can interrupt the playing of a composition and view the graphic representation of the measure being played at that point, but graphics are not used when the piece is actually being played.

There are a few features present in some music editors for other machines which are not present in *Music Maker*. For example, the only way to initiate repeats is by manually pressing "SHIFT R" during the playing of a piece; no form of looping can be structured into a compostition.

Given the relatively vast storage space available (compared with music compositions written in TI BASIC), together with the copying feature, the lack of repeat capability is less significant than it might otherwise be. With 16K of RAM, you will be able to write about 900 notes for each of the three voices. For example, writing all sixteenth notes for three voices, the file could be 57 measures long; with all quarter notes, it could be 224 measures. Additionally, there is no capability to write phrases and then arrange them in different voices. This capability could be useful when employing the device of imitation, such as writing canons and fugues. Even so, the same effect can be achieved with *Music Maker*—it just takes a little more effort.

In summary, despite the few shortcomings mentioned, the Traditional Mode provides a beautiful graphics-based editor which makes the process of writing music as enjoyable as listening to the finished product. Even if this were the extent of the *Music Maker*'s capabilities, we feel it would be an excellent investment at the suggested retail price of $39.95.

Sound Graphs

While some knowledge of music theory is essential for effective use of the Traditional Mode, the Sound Graphs method may be used without any prior understanding of music terminology. As the name implies, music is entered in a Cartesian coordinate graph format. The frequency graph can have a resolution of one-hundred-twenty vertical positions (frequency) by twenty horizontal positions (duration) per "measure." A Sound Graphs music file may contain up to 46 measures. A color-coded line is plotted on the graph with the cursor, and as in Traditional Mode, a different color is used for each voice.

The volume graph has a resolution of eight vertical positions (volume) by twenty horizontal positions (duration), and appears below the frequency graph. A separate volume graph may be plotted for each voice appearing in the frequency graph (default is the highest volume). In addition to the three voices, a Sound Graph may also include a noise which is plotted on the volume graph.

The user has the option of either Discrete or Continuous tones. Under the Discrete option, the vertical axis is divided into thirty frequencies, consisting of C Major diatonic pitches from the second A below middle c to the third b above middle c. You can, however change any or all of these pitches with the List Tones option. Although any frequency from 110 Hz to 20,000 Hz can be used, tables are provided in TI's excellent documentation, giving the frequencies for chromatic, pentatonic, and gypsy scales. The frequencies can be changed at any time, even during or after the plotting of a Sound Graph.

Under the Continuous option you specify the upper and lower limits of the frequency range. These can be changed as often as you wish. The frequency axis is divided into 120 steps within this range, giving a frequency "slide" which sounds continuous and can be used to create sound effects such as whistles and sirens as well as interesting experimental music sounds. When you take into consideration the fact that a noise can be used in addition to three voices and that the composition can be played as fast as twenty characters per second, the range of possibilities is quite extensive.

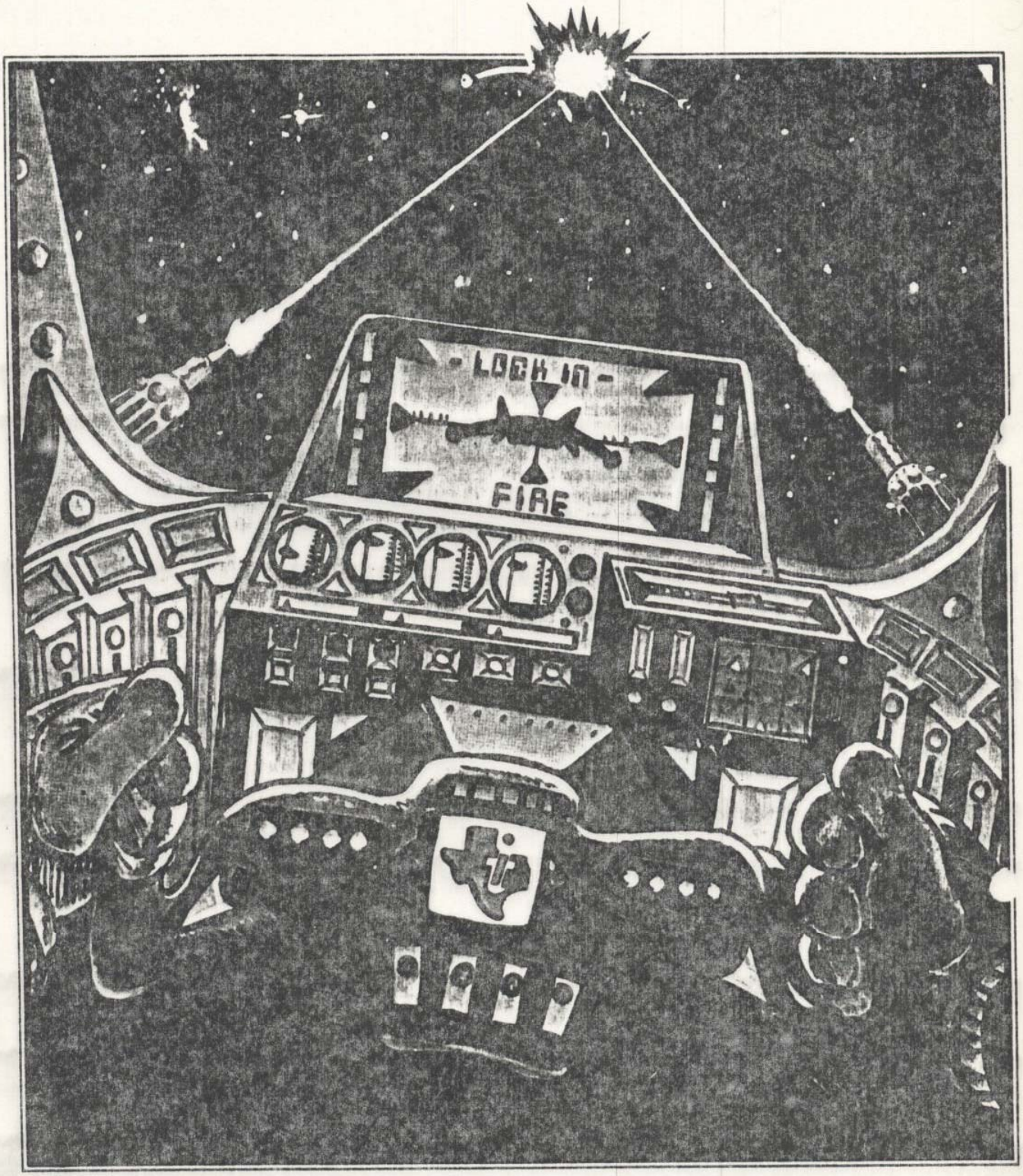
In evaluating the noise, we were surprised to find that we could not distinguish any difference between the periodic and "white noise" groups—i.e., noises 1-4 and 5-8, respectively. Noise 1 appears to be the same as noise 5; noise 2 the same as 6, and so on. If you are familiar with the difference between periodic noise and white noise in TI BASIC, do not expect to find the same distinction in the *Music Maker*.

Other aspects of using Sound Graphs are identical with the corresponding procedures used in Traditional Mode (namely editing, playing, saving, and printing).

If you have no knowledge of music theory, using Sound Graphs is a great way to begin exploring the TI-99/4A's music capabilities. Even if you are familiar with music fundamentals, you will be amazed at the versatility of the Sound Graphs method, and you will find that your TI-99/4A has potential you would not have thought possible.

In conclusion, the *Music Maker* Command Cartridge will greatly enhance one of the already outstanding features of your computer—its capacity for sound and music. We believe it is an accessory you will not want to be without.

7
Computer Gaming



7

Computer Gaming

Action-packed games to delight arcade fans and mind-stretching challenges for strategists.

The JOY of COMPUTER GAMING

It's A Dirty Job, But Somebody's Gotta Do It . . .

Over the last couple of months, I've had virtually no rest. First it was those pesky aliens: They hurled bombs, missiles, mines, and laser blasts at me around the clock. Some even tried to gobble me up on sight! No respect at all. . . These hordes of menacing foes must have come from nearly a dozen different hostile worlds. (Why is it I've never seen a "friendly" alien?) Each of these worlds evidently has its own individual concept of combat strategy, weapon design, ethics and morality because the modes and severity of attacks differed widely. One thing, however, that all these dastardly devils had in common was their quarry—me!

Some of the attacking hordes were accompanied by a malevolent thumping as their precise marching formation advanced hypnotically toward my flimsy barricade. Others stayed stationary but hurled down torrents of lethal missiles that I had to alternately duck and target my lasers against. (My neighbors must have been really surprised when they noticed all that debris strewn across their yards. . .) What? Was I nervous? Not too—that is, not until I had to pilot all those strange land and space craft—everything from X-wing fighters to futuristic prairie schooners. Just when I'd feel comfortable at the controls of one, Ka-boom!—I'd be under vicious attack, or c-r-r-unch!—smack in the middle of a deadly asteroid belt. Nothing like huge chunks of space rocks whizzing around your head to keep you on your toes. . .

But it didn't end with all those downright nasty aliens and slimy, vile space creatures. Oh no, not by a long shot. There were still the Empire forces to contend with. Here, however, the battles were more scattered and slower paced; I had time to launch torpedoes and probes, as well as assess casualty reports and plan long-term strategy. Just when everything was going so well, a Red Alert brought me back down to earth. It seems the Cold War was no longer so cold. . .and my country needed me to command a SAM (surface-to-air missile) site. With all that sophisticated RADAR equipment, it shouldn't have taken too long to finish "locking in" on the enemy missiles and blasting them to smithereens, but as fate would have it, another series of emergencies sent me packing—first to rescue a downed helicopter crew from shark-infested waters, and then on a hazardous journey to the moon and Mars, where I had to jockey my landing craft over some pretty rough terrain.

You'd think by then I should have received a few words of thanks, wouldn't you? But no, the moment I landed on Mars, some terrorists decided to have a field day. . .and there I was right back in the thick of things—commanding a bomb squad. Now defusing a time bomb is no Sunday picnic! If my nerves could withstand that heart-thumping activity, you'd think I'd be in pretty good shape for a few more adventures yet to come.

But nothing prepared me for what I was up against next. Certainly, no one told me when I took this job that hundreds of horrible deaths awaited just around the corner. All they talked about was the treasure and glory! But when I reached the edge of the high cliff, it was already too late to turn back: On my left, a hungry python slithered toward me; to the right, a quicksand bog surrounded by bleached bones awaited; and behind me, a large grizzly bear blocked my only path into the forest.

Well, luckily I got out of that one with my skin, but one adventure led to another. . . And before I could take some well-earned rest, I somehow had gotten involved in a pirate treasure hunt, an escape from an ancient pyramid, the ferreting out of an awesome secret on a savage island, saving a Count trapped by a fiendish curse, preventing a nuclear reactor from blowing its top, and alternately exploring a ghost town, mysterious fun house, ancient alien civilization, enchanted treasure world, and a dark kingdom populated by orcs, dwarves, an old dungeon master, a beautiful princess in distress, and an evil ringwraith.

Whew—I never thought that one mortal could get so tired. What I really needed was a chance to relax and unwind. . . So, handing my ticket to adventure over to Indiana Jones, I planned on doing nothing but sitting back in my favorite easy chair and listening to some good music. But They had other plans for me. It was no use complaining; I'd heard the argument a hundred times before: "It comes with the territory. . ." For some reason or other I was needed to run up a bankroll by betting on the ponies, to lead a championship baseball team to victory, to bring back a

shiny bowling trophy, to don my skis to better the old slalom record, to outrace a suicide car, and then to take part in a grueling decathlon.

After somehow getting through that long, long decathlon, I sat down to a nice big bowl of Wheaties and planned my R&R. Nothing too strenuous. . .nothing too mentally demanding. . .just some good clean fun. So off to the casinos for some baccarat, blackjack, craps, poker, and the slots. It was fun while it lasted, but They needed me back on the job again.

I knew something really BIG must have been in the works because of the way my training for the forthcoming mission was being carried out: plenty of practice with challenging word games, concentration exercises and contortions with cantankerous colored cubes. They obviously wanted my wits sharp for the BIG assignment coming up. But before I'd find out what it was, there was an obstacle course to negotiate, and then the final test of my state of mental readiness—passage through a series of simply complex, complexly simple 3-D mazes. I almost didn't make it through that one. . .

Now I was ready. The BIG assignment finally came in: Someone was needed to guide a dumb chicken safely across a 20-lane highway. . . What? Enough is enough! Tell 'em I'm not here. Guide a chicken across a road like that? Instant chicken salad—with me probably ending up being accused of fowl play. . . Let 'em get somebody else for that one. I wouldn't do it now even if They awarded me the Pullet-ser Prize!

Micro Motivation Comes Full Circle

I've been sitting here now several hours thinking and wondering—thinking about those psychedelic-sounding escapades of mine, and wondering about the fantastic powers of imagination that we all must have—letting us see what we want or expect to see. In my case, it was easy because I had a partner—one who was, incidentally, a lot more patient than that dumb chicken I eventually got teamed up with. Who was this patient partner? Some gaunt guru? Or sinewy sorcerer? No, none of these. My partner in all this was a friendly Texan—a TI Home Computer equipped with the latest in games software.

I have never really cared very much for games. And even when it became obvious that microcomputers were rapidly becoming the ultimate "games machines," I still felt that all the excitement of video games was just a passing fancy. It was my belief that the popularity of computer games was simply due to people just trying to find additional uses for machines that they bought primarily for other purposes. Now I know better. . .

What we're now just starting to see happen is actually the reverse: This year, several million consumers will be considering interactive video games—as opposed to passive TV watching—that they can play at home in the company of friends and family, instead of plunking their quarters down coin slots at crowded arcades or all-night grocery stores. When they start to shop around and compare prices and features, increasing numbers of them will start to find that the stand-alone, cartridge-based, dedicated games machines can be almost as much money as the new breed of lower-cost microcomputers.

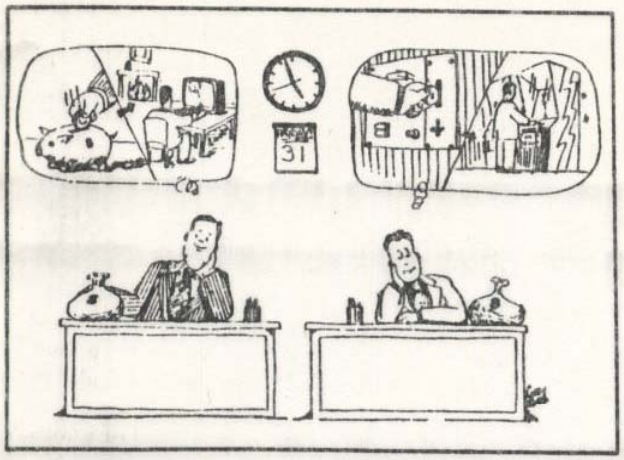
The handwriting is already on the wall: As the price of microcomputers falls even lower, many, many more consumers who were initially looking for video games machines will be able to justify the slightly higher cost of a full computer on the basis of potentially being able to do so much more than "just play games." Ironic, isn't it. . .

A few years ago, the great topic of speculation and cause for disagreement in the microcomputer industry was how to best increase public awareness and acceptance of these miracle machines so that a mass market with its lofty goal of "a computer in every home" could be eventually realized. Everything from electronic mail and banking, to education, home management, and tax/financial record keeping was nominated as being a likely candidate for the magic catalyst. Sure, entertainment was mentioned, but it was usually lumped together somewhat amorphously with home management and education. Nowhere do I remember anyone coming out and stating that it would be computer gaming that would ultimately be this catalyst and pave the way.

The Seriousness of Playing Games

But regardless of whether video games are a primary or secondary motivation for getting a microcomputer, it's rapidly becoming obvious that electronic game playing isn't all just a game. Psychiatrists, psychologists, therapists and educators are discovering how video games can dramatically benefit their players. We hear reports of how the games are speeding eye-hand coordination, sharpening driving and math skills (since the intricate strategies and geometric patterns of many video games provide painless instruction in logic, trigonometry and physics), preventing youth from being stricken by technological "future shock," and providing an emotional rescue (by dissipating anger and frustration, assuaging loneliness and allowing both the recapture of lost athletic prowess as well as the prowess that never was).

Application of video game playing as a form of therapy is definitely on the rise. We're now seeing this technique used in treating brain-damaged victims of strokes, accidents and senile dementia. The most impressive results, however,



have come from work with retarded or emotionally disturbed children. Here, video games often break through where other methods fail. Psychologists have credited this to the "mastery experience" that is now possible for children who formerly were not able to be good in anything else. Until their exposure to games, they have never had a refuge of accomplishment from which to deal with the outside world. Once children become good at something (and, as a result, proud of their achievements), their attitudes and performance in other activities also dramatically improve.

The Hardware Sets the Stage

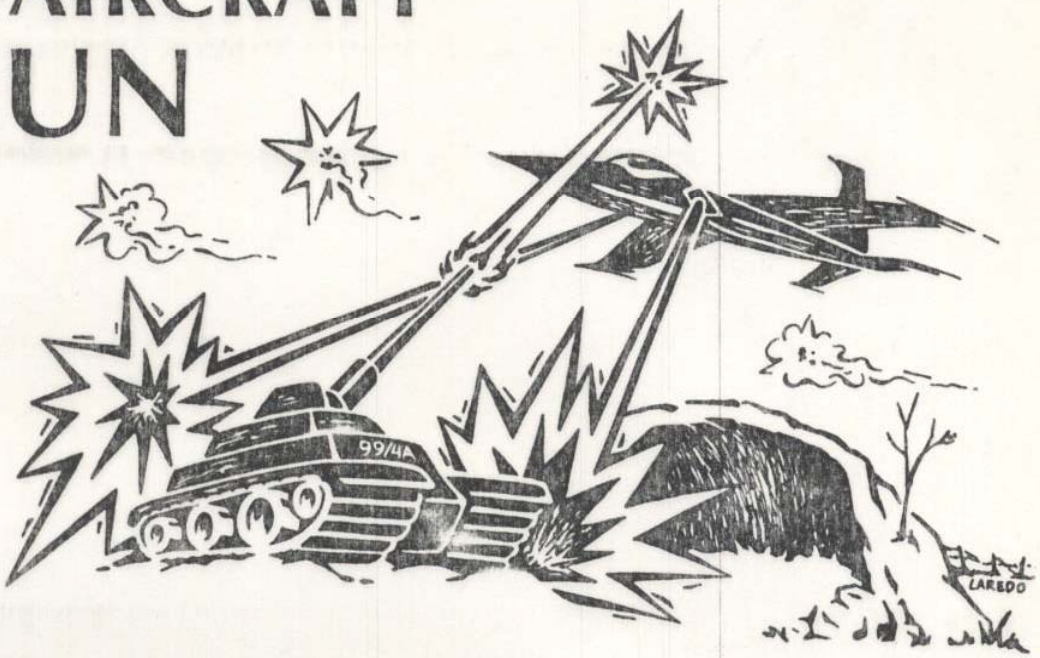
As opposed to video games machines that are designed to be just "machines that play games," microcomputers are usually designed to perform many types of jobs, or handle certain types of work more efficiently. This architectural design determines the gaming environment that a particular computer will present to its users. Any limitations or constraints will be very obvious. For example, if a computer was designed without color graphics capability, then the games software compatible with this particular machine could not utilize color. Likewise, sound effects, music, 3-D animation and synthetic speech are other game-enrichment capabilities that a microcomputer may or may not possess.

A comparison of all presently available microcomputers yields the surprising conclusion that only the Texas Instruments TI-99/4A personal computer has all the above named capabilities and permits programmers to use them all in the same program. This represents an abundance of "raw materials" from which to construct games. When you combine this with TI's fast and powerful 16-bit microprocessor (TMS9900), it becomes apparent that these Texas Instruments personal computers offer one of the best gaming environments available.

The separate Video Display Processor chip (TMS9918A) inside the TI-99/4A is a good example of TI implementing internally in hardware what other computers require programmers to do in software (if indeed it can be done at all). This very sophisticated device gives the games programmer the ability to simply access and set in motion (independently of the program logic) 32 smoothly moving colored objects called "sprites"—objects whose shapes can very easily be defined, magnified, colored, given a 3-D overlapping appearance and checked for collisions. These are the modular components from which many more exciting arcade-type games will be constructed in the future.

ANTI-AIRCRAFT GUN

TI BASIC



Despite the fact that action games programmed in a high-level language such as TI BASIC run much slower than in low-level languages such as 9900 Assembly Language or GPL (the programming language of TI's Command Cartridges), it is indeed possible to create reasonably fast "real-time" games if you observe a few rules:

1. Keep the number of moving objects to a minimum.
2. Keep all unnecessary statements out of loops used to move objects.
3. Use only one character to define objects you want to move quickly.
4. Increment the positions by two spaces each loop. This makes the movement slightly jerky, but contributes greatly to the illusion of speed.

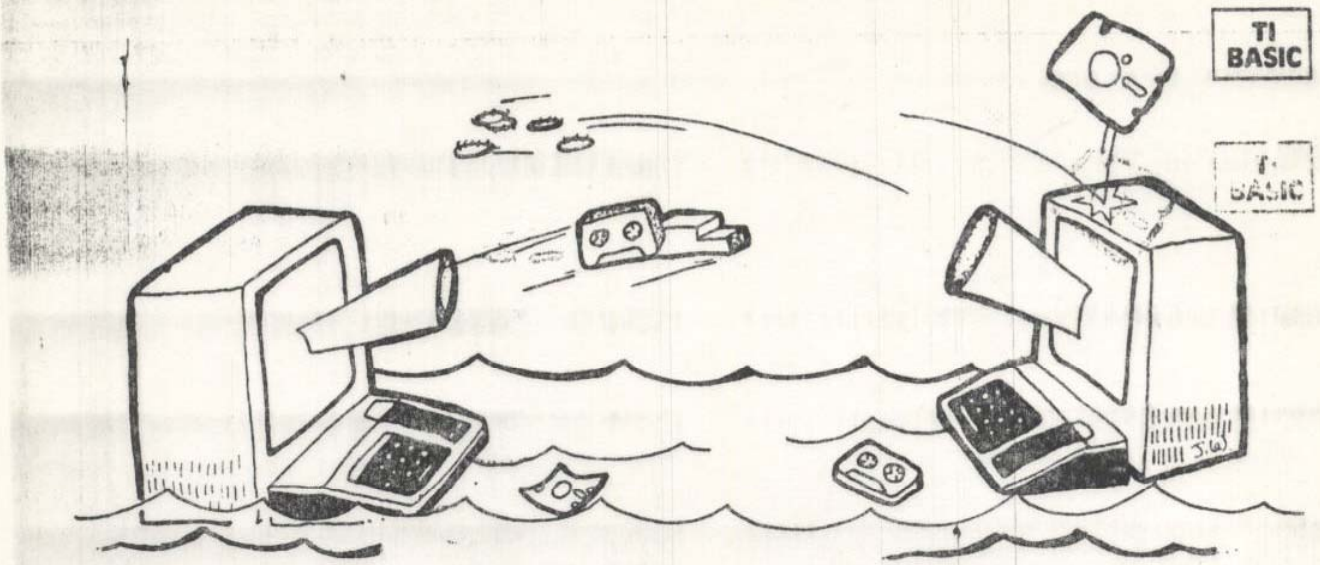
I've followed these rules in writing *Anti-Aircraft Gun*. The basic idea of the game is simple: You must shoot down an attacking plane with your missile launcher before it blasts you twice with its laser. The plane attacks at random heights from both the left and right sides. Its speed and frequency of laser fire are dependent on the skill level you choose. Your missile launcher can move along the ground, and even hide behind a barrier; but when it fires a missile, it is committed to its last position until the missed shot passes off the screen. You'll have to move around as much as possible because the plane "remembers" your last position and there is higher probability it will fire the laser near that position. And don't expect too much protection from the barrier: After five laser blasts (or less, if you launch missiles through it), it will disintegrate and leave you exposed.

EXPLANATION OF THE PROGRAM
Anti-Aircraft Gun

Line Nos.	
100-670	Instructions.
680-810	Sets up levels of difficulty.
820-870	Sets up variables to make plane fire more as difficulty increases.
880-1110	Character definitions and color assignments.
1120-1170	Initial displaying of tank.
1180-1220	Displays ground.
1230-1260	Calculates plane's height.
1270-1380	Determines the direction of the plane.
1390-1450	Reads keyboard, and branches to subroutines.
1460-1530	Fires tank's rocket.
1540-1610	Moves plane.
1620-1650	Decides whether plane will fire or not.
1660-1760	Fires plane's laser; checks for hits.
1770-1780	Checks for a hit on the plane by the tank's rocket.
1790-1890	Checks for plane at the edge of the screen.
1900-2030	Determines new direction for the plane.
2040-2090	If plane and tank hit simultaneously, the tank wins.
2100-2180	Determines new direction for the plane.
2190-2330	Calculates score.
2340-2420	Prints score.
2430-2460	Plane is destroyed by the tank.
2470-2620	Calculates if a free game was won; starts over.
2630-2750	Moves tank left.
2760-2880	Moves tank right.
2890	END.

```
100 REM  *********************
110 REM  *                   *
120 REM  *  ANTI-AIRCRAFT GUN *
130 REM  *                   *
140 REM  *********************
150 REM
160 REM
170 CALL CLEAR
180 CALL SCREEN(8)
190 REM  * INTRODUCTION *
200 PRINT TAB(3);"PRESS 1 FOR INSTRUCT
    IONS"
210 PRINT TAB(8);"AND N FOR NONE":::::
    :::::
220 CALL KEY(0,LP,PL)
230 IF PL=0 THEN 220
240 IF LP=78 THEN 680
250 PRINT TAB(5);"* ANTI-AIRCRAFT GUN
    *"
260 PRINT
270 PRINT "THE OBJECT OF ANTI-AIRCRAFT
    "
280 PRINT "IS TO DESTROY AS MANY PLANE
    S"
290 PRINT "AS POSSIBLE. TO FIRE YOUR "
300 PRINT "MISSILE, PRESS Q. TO MOVE "
310 PRINT "LEFT, PRESS S, AND FOR RIGH
    T"
320 PRINT "PRESS D."
330 PRINT
340 PRINT TAB(9);"* PLANES: *"
350 PRINT
360 PRINT "THE PLANE FIRES A NEWLY-"
370 PRINT "DEVELOPED LASER. THE PLANE"
380 PRINT "MAY COME FROM LEFT OR RIGHT
    "
390 PRINT "WARNING! IT HAS A RADAR THA
    T"
400 PRINT "REMEMBERS YOUR LAST FIRING"
410 PRINT "POSITION & TRIES TO GET YOU
    "
420 PRINT "THERE; BETTER MOVE AROUND."
430 PRINT :::
440 PRINT "PRESS ANYTHING TO CONTINUE"
450 PRINT
460 CALL KEY(0,KL,LK)
470 IF LK=0 THEN 460
480 PRINT TAB(8);"* BARRIER: *"
490 PRINT
500 PRINT "THE LASER CAN'T PENETRATE "
510 PRINT "THE BARRIER, BUT THE BARRIE
    R"
520 PRINT "CAN SUSTAIN ONLY 5 DIRECT "
530 PRINT "HITS. YOU CAN FIRE WHEN BE-
    "
540 PRINT "HIND IT, BUT THIS WILL ONLY
    "
550 PRINT "SHORTEN ITS LIFE."
560 PRINT
570 PRINT TAB(8);"* SCORING: *"
580 PRINT
590 PRINT "PLANES ARE SCORED ACCORDING
    "
600 PRINT "TO HEIGHT: 5 FOR LOWEST, 20
    "
610 PRINT "FOR HIGHEST. IF YOUR SCORE
    "
620 PRINT "IS OVER 100 THEN YOU GET A
    "
630 PRINT "FREE BONUS GAME."
640 PRINT :::::::
650 PRINT "PRESS ANYTHING TO CONTINUE"
660 CALL KEY(0,KL,LK)
670 IF LK=0 THEN 660
680 CALL CLEAR
690 CALL COLOR(2,2,8)
700 PRINT "INPUT LEVEL OF DIFFICULTY:"
710 PRINT
720 PRINT "(1) PRO"
730 PRINT
740 PRINT "(2) INTERMEDIATE"
750 PRINT
760 PRINT "(3) NOVICE"
770 PRINT
780 PRINT "(4) BEGINNER"
790 PRINT :::::::
800 CALL KEY(0,DIF,XS)
810 IF (DIF<49)+(DIF>52)=-1 THEN 800
820 REM  PLANE FIRING MORE AS DIFFIC
    ULTY INCREASES
830 DIF=DIF-38
840 MS="0"
850 AA=1
860 ZZ=3
870 CALL CLEAR
880 REM  DEFINE CHARACTERS
890 REM  BAS=BARRIER, LS=  LASER, RS
    =ROCKET, PS & PRS= PLANE, TLS & TR
    S=TANK, DS= DESTRUCTION
900 BAS="FFFFFFFF"
910 DS="2A044122890452200"
920 LS="1818181818181818"
930 RS="1818181818183C7E"
940 PS="6030908FFFF983060"
950 PRS="060C19FFFF190C06"
960 TLS="071F7FFFFF7F3F0F"
970 TRS="E0F8FEFFFFFEFCE0"
980 CALL CHAR(120,DS)
990 CALL CHAR(144,LS)
1000 CALL CHAR(97,RS)
1010 CALL CHAR(112,TLS)
1020 CALL CHAR(113,TRS)
1030 CALL CHAR(101,BAS)
1040 CALL CLEAR
1050 BA=0
1060 CALL COLOR(2,16,2)
1070 CALL COLOR(16,2,2)
1080 CALL COLOR(15,12,8)
1090 CALL COLOR(13,3,3)
1100 CALL COLOR(12,7,8)
1110 T=15
1120 REM  INITIAL PRINTING OF TANK
1130 CALL HCHAR(24,T,112)
1140 CALL HCHAR(24,T+1,42)
1150 CALL HCHAR(24,T+2,113)
1160 CALL HCHAR(23,T+1,152)
1170 CALL HCHAR(22,15,101,3)
1180 REM  PRINT GROUND(GREEN)
1190 CALL HCHAR(23,1,128,15)
1200 CALL HCHAR(23,17,128,15)
1210 CALL HCHAR(24,1,128,14)
1220 CALL HCHAR(24,18,128,14)
1230 REM  A=HEIGHT OF PLANE
1240 RANDOMIZE
1250 A=2*INT(9*RND)+1
1260 IF A=1 THEN 1240
1270 REM  DETERMINE PLANE'S DIRECTION
1280 RANDOMIZE
1290 B=INT(2*RND)+1
1300 ON B GOTO 1310,1350
1310 B=30
1320 CALL CHAR(96,PRS)
1330 DIR=-2
1340 GOTO 1380
1350 DIR=2
1360 CALL CHAR(96,PS)
1370 B=2
1380 Y=21
1390 REM  * BEGIN LOOP *
1400 CALL KEY(0,K,S)
1410 REM  81 FIRES ROCKET, 83 AND 68
     MOVE TANK
1420 IF K=81 THEN 1450
1430 IF K=83 THEN 2640
1440 IF K=68 THEN 2770 ELSE 1550
1450 IF Y=21 THEN 1480 ELSE 1470
1460 REM  FIRE ROCKET
1470 CALL VCHAR(Y+2,T+1,32)
```

```
1480 CALL VCHAR(Y,T+1,97)
1490 T=T+1
1500 Y=Y-2
1510 IF Y=-1 THEN 1520 ELSE 1550
1520 CALL VCHAR(1,T+1,32)
1530 GOTO 1380
1540 REM  MOVE PLANE
1550 IF DIR=-2 THEN 1570
1560 IF B<3 THEN 1590
1570 CALL VCHAR(A,B-DIR,32)
1580 IF B<2 THEN 1900
1590 CALL VCHAR(A,B,96)
1600 RANDOMIZE
1610 Q=INT(15*RND)+1
1620 REM  WILL PLANE FIRE      LASER?
1630 TT=TT-2+INT(Q/2)
1640 IF B=TT THEN 1670
1650 IF Q>DIFF THEN 1670 ELSE 1820
1660 REM  PLANE WILL FIRE      LASER
1670 IF BA=5 THEN 1760 ELSE 1680
1680 IF B=16 THEN 1690 ELSE 1760
1690 BA=BA+1
1700 IF BA<>5 THEN 1720
1710 CALL HCHAR(22,15,32,3)
1720 CALL VCHAR(A+1,B,144,21-A)
1730 CALL VCHAR(A+1,B,32,21-A)
1740 CALL SOUND(500,-5,2)
1750 GOTO 1850
1760 CALL VCHAR(A+1,B,144,23-A)
1770 IF B<>T+1 THEN 1790
1780 IF A<>Y+2 THEN 2440
1790 CALL VCHAR(A+1,B,32,22-A)
1800 CALL VCHAR(23,B,128,2)
1810 CALL SOUND(500,-5,2)
1820 IF B>30 THEN 1840 ELSE 1830
1830 IF B<3 THEN 1840 ELSE 1850
1840 CALL VCHAR(A,B,32)
1850 IF A<>Y+2 THEN 1870
1860 IF B=T+1 THEN 2050
1870 B=B+DIR
1880 IF B>32 THEN 1900 ELSE 1890
1890 IF K=81 THEN 1420 ELSE 1400
1900 RANDOMIZE
1910 B=INT(2*RND)+1
1920 REM  DETERMINE PLANE'S    DIRECTION
1930 ON B GOTO 1940,1980
1940 B=30
1950 CALL CHAR(96,PR$)
1960 DIR=-2
1970 GOTO 2000
1980 DIR=2
1990 CALL CHAR(96,P$)
2000 RANDOMIZE
2010 A=2*INT(9*RND)+1
2020 IF A=1 THEN 2000
2030 GOTO 1890
2040 REM  TANK HITS PLANE
2050 CALL SOUND(1000,-5,2)
2060 REM  A TIE-TANK WINS-     REPRINT T
     ANK
2070 CALL HCHAR(24,T+1,42)
2080 CALL HCHAR(23,T+1,152)
2090 CALL HCHAR(A,B,120)
2100 REM  DETERMINE PLANE'S    DIRECTION
2110 B=INT(2*RND)+1
2120 ON B GOTO 2130,2170
2130 B=30
2140 DIR=-2
2150 CALL CHAR(96,PR$)
2160 GOTO 2200
2170 DIR=2
2180 CALL CHAR(96,P$)
2190 REM  * SCORING *
2200 IF A>13 THEN 2210 ELSE 2230
2210 SC=SC+5
2220 GOTO 2300
2230 IF A>7 THEN 2240 ELSE 2260
2240 SC=SC+10
2250 GOTO 2300
2260 IF A<>5 THEN 2290
2270 SC=SC+15
2280 GOTO 2300
2290 SC=SC+20
2300 M$=STR$(SC)
2310 RANDOMIZE
2320 A=2*INT(9*RND)+1
2330 IF A=1 THEN 2310
2340 REM  PRINT SCORE
2350 FOR I=1 TO LEN(M$)
2360 CV=ASC(SEG$(M$,I,1))
2370 CALL HCHAR(AA,ZZ+I,CV)
2380 NEXT I
2390 GOTO 1380
2400 FOR I=1 TO 1000
2410 NEXT I
2420 GOTO 1040
2430 REM  PLANE HIT TANK
2440 CALL SOUND(500,-6,2)
2450 CALL CLEAR
2460 SX=SX+1
2470 REM  IF 2 GAMES PLAYED, START OVER
2480 IF SX<2 THEN 1040
2490 PRINT TAB(7);"  GAME OVER  "
2500 PRINT TAB(7);"YOUR SCORE IS";SC
2510 FOR I=1 TO 10
2520 PRINT
2530 NEXT I
2540 SX=0
2550 IF SC<100 THEN 2600
2560 PRINT TAB(3);"* YOU GET A FREE GAM
     E *"
2570 REM  * A FREE GAME! *
2580 SC=0
2590 GOTO 2400
2600 SC=0
2610 REM  GO BACK TO START
2620 GOTO 680
2630 REM  MOVE TANK LEFT
2640 IF T<3 THEN 2650 ELSE 2680
2650 CALL SOUND(250,110,2)
2660 T=1
2670 GOTO 2690
2680 T=T-2
2690 CALL HCHAR(24,T,112)
2700 CALL HCHAR(24,T+1,42)
2710 CALL HCHAR(24,T+2,113)
2720 CALL HCHAR(23,T+1,152)
2730 CALL HCHAR(24,T+3,128,2)
2740 CALL HCHAR(23,T+3,128)
2750 GOTO 1550
2760 REM  MOVE TANK RIGHT
2770 IF T>27 THEN 2780 ELSE 2810
2780 CALL SOUND(-250,110,2)
2790 T=29
2800 GOTO 2820
2810 T=T+2
2820 CALL HCHAR(24,T,112)
2830 CALL HCHAR(24,T+1,42)
2840 CALL HCHAR(24,T+2,113)
2850 CALL HCHAR(23,T+1,152)
2860 CALL HCHAR(24,T-2,128,2)
2870 CALL HCHAR(23,T-1,128)
2880 GOTO 1550
2890 END
```



BATTLE AT SEA

"Damn the torpedoes! Full speed ahead . . ." Get ready, all you armchair admirals out there in 99'er-land. You're about to do battle with the most crafty enemy of all—the Imperial TI Fleet. If you're old enough to remember those rainy Saturdays in the pre-TV age, you've probably spent many an hour with pencil and paper playing Battleship. In the intervening years, Battleship has been dressed up as a consumer item in many forms: First it was "cardboardized," then "plasticized," and finally "electronicized."

Well gang, as it happened, one rainy Saturday afternoon a few months ago, I had this mad urge to play Battleship . . . The expensive electronic version looked really enticing in a local toy store display, but I sure wasn't going to spring for it—especially when I had my trusty TI-99/4 personal computer waiting to carry out my every command. So program it I did. The result: Battleship has now been "99'erized" into a 16K TI BASIC version, which I call *Battle at Sea*.

Two 10 x 10 grids are displayed on the screen along with the row and column designations. The computer will ask you to enter coordinates for the placement of each of your ships on the grid at the right. Each coordinate must be entered separately; for example, first A 5 and then A 6 are entered for the destroyer. Since the ships occupy different numbers of grid squares, I've put in a counter for each ship to indicate how many remaining squares must be entered.

After all the coordinates for a ship have been entered, that ship will be displayed on the screen. Once all five ships are set up, the computer will secretly set up its own ships on the grid to the left. You won't be able to see the computer's ships, since the whole idea of the game is to try to find them.

Once the computer has set up its ships, it will ask you for the coordinates of your shot at its grid (on the left). You must enter your shot as a row letter, then a column number. Valid coordinates are from A to J and from 0 to 9. Any other entry will result in having to enter the coordinates again. Your hit or miss will be marked on the grid and displayed at the bottom of the screen as a MISS or **HIT**. The computer will then take a shot at your grid. It cannot see your ships, but it does keep track of where the hits and misses are.

After a hit, any ship that has been sunk will be displayed at the bottom of the screen. The score is also updated at this time: one point for each ship sunk. The first player to sink all five ships will win the game.

Because there are no moving objects in this game, speed was not the most important factor in the game design. The action happens to be fairly fast, but the critical factor was programming the computer to make intelligent decisions. With no limit on available memory, I might have been able to write a program with flawless logic. But here that wasn't the case—I had to stay within the confines of standard 16K TI BASIC.

I started by giving the computer a set of rules and several variables to test for a given situation. First, if a ship has been hit only once, the computer will take random shots around that hit until the direction is determined. It will then continue in that direction until the ship either sinks, misses a shot, or runs up to the edge of the grid. It will then reverse and shoot at the other end if the ship was not sunk.

And now it's you against the Imperial TI Fleet!

EXPLANATION OF THE PROGRAM
Battle At Sea

Line Nos.	
100-630	Initializaton: Set up variables, character definition, and color assignments.
640-870	Instruction page.
880-1010	Display 10 x 10 grids.
1020-1100	Control loop for setting up your ships on the 10 X 10 grid.
1110-1360	Subroutines holding data on each ship.
1370-1380	Branch to subroutine: computer sets up its ships.
1390-1530	Display message for ship coordinates to be entered.
1540-1710	Read keyboard; INPUT coordinates of ships.
1720-1950	Put the coordinates in order
1960-2050	Check that all coordinates are valid.
2060-2220	Display ship on the 10 x 10 grid.
2230-2380	Control loop holding data for computer to set up its ships.
2390-2600	Subroutine to set up computer's ships at random.
2610-2860	Set up variables for messages; subroutines for displaying those messages.
2870-2910	Keep track of which turn it is. Branch to either user's shot, or computer's shot.
2920-3170	computer takes random shot at your grid if no ships are hit.
3180-3340	Read keyboard; INPUT user's shot a computer's grid.
3350-3570	Check for valid INPUT, hit or miss.
3580-3710	Check for direction of hits on your ships.
3720-4150	Take random shot around last hit if only one hit on the ship.
4160-4450	If more than one hit on a ship takes another hit in proper direction.
4460-4620	Adjust varibales when computer gets a hit.
4630-4770	Find out how many hits on each ship; used for both computer and user.
4780-4980	Calculate score, and number of ships hit, but not sunk.
4990-5020	Display any ships that have been destroyed after every hit.
5030-5090	Display scores.
5100-5190	End of game message.
5200-5320	Re-initialize variables for next game.
5330-5340	END of game.
5350-5460	Subroutine to make sure ships are in line.

```
100 REM  *********************
110 REM  *  BATTLE AT SEA  *
120 REM  *********************
130 REM
140 REM
150 REM
160 REM
170 REM
180 RANDOMIZE
190 CALL SCREEN(12)
200 CALL CLEAR
210 PRINT TAB(7);"BATTLE AT SEA"
220 PRINT
230 PRINT
240 PRINT ::::::::::::::::
250 OPTION BASE 1
260 DIM P(10,10),O(10,10),SH(5,5,2)
270 CALL COLOR(14,7,1)
280 CALL COLOR(15,11,1)
290 CALL CHAR(96,"00000000FF7F3F1F")
300 CALL CHAR(97,"0000000FFFFFFFFF")
310 CALL CHAR(98,"3C7EFFFFFFFFFFFF")
320 CALL CHAR(99,"000000FFFEFCF8")
330 CALL CHAR(100,"10307070707070707070")
340 CALL CHAR(101,"70707070707070707070")
350 CALL CHAR(102,"787C7E7E7F7F7C7870")
360 CALL CHAR(103,"70707070707070101010")
370 CALL CHAR(104,"0008040307FF7F3F")
380 CALL CHAR(105,"8C4C3CFEFFFFFFFF")
390 CALL CHAR(106,"01023C3FFFFFFFFF")
400 CALL CHAR(107,"0002040F8FFFFEFE")
410 CALL CHAR(108,"10307274787878787870")
420 CALL CHAR(109,"7C7C707117A7C7C7C7C")
430 CALL CHAR(110,"7F7F787C7C7C7A79")
440 CALL CHAR(111,"70787878C7C727110")
450 CALL CHAR(112,"0010886F7F7F7F3F")
460 CALL CHAR(113,"09C5C3F3FFFFFFFF")
470 CALL CHAR(114,"0002040F8FFFFEFE")
480 CALL CHAR(115,"10307274787978A7C")
490 CALL CHAR(116,"797A7C7C7F7F7F787C")
500 CALL CHAR(117,"7C7C7A7978787C7C1A")
510 CALL CHAR(118,"0000003FFF7F3F")
520 CALL CHAR(119,"E67EFFFFFFFFFFFF")
530 CALL CHAR(120,"0000000E0FCFFFFE")
540 CALL CHAR(121,"10307878787878787870")
550 CALL CHAR(122,"787C7C7E7E7E7F7F")
560 CALL CHAR(123,"7C7C7878787870302020")
570 CALL CHAR(124,"03030F1FFF7F3F")
580 CALL CHAR(125,"0006060F0FFFFEFE")
590 CALL CHAR(126,"10307070787878787F")
600 CALL CHAR(127,"7C78787070707070")
610 CALL CHAR(128,"FF8181818181818181FF")
620 CALL CHAR(136,"815E2C366A3C2442")
630 CALL CHAR(144,"81667E3C3C7E6681")
640 CALL SOUND(-3000,220,30,554,20,104
    7,20,-8,30)
650 PRINT TAB(7);"BATTLE AT SEA"
660 PRINT :"YOU MUST DESTROY THE ENEMY
    "
670 PRINT "SHIPS BEFORE THE COMPUTER"
680 PRINT "DESTROYS YOUR SHIPS."
690 PRINT :"TO SET UP YOUR SHIPS YOU"
700 PRINT "MUST ENTER COORDINATES ON"
710 PRINT "THE 10 X 10 GRID ON THE
    RIGHT."
720 PRINT "ENTER THE ROW, THEN THE
    COLUMN."
730 PRINT "EXAMPLE: A5"
740 PRINT "AFTER YOUR SHIPS ARE SET UP
    ."
750 PRINT "YOU WILL TAKE A SHOT AT THE
    "
760 PRINT "ENEMY SHIPS BY ENTERING ONE
    "
770 PRINT "PAIR OF COORDINATES ON THE"
780 PRINT "ENEMY GRID."
790 PRINT "THE COMPUTER WILL THEN"
800 PRINT "TAKE A SHOT AT YOUR SHIPS."
810 PRINT :"THE COMPUTER CANNOT SEE"
820 PRINT "YOUR SHIPS. YOU CANNOT SEE"
830 PRINT "THE COMPUTER'S SHIPS."
840 PRINT "ENTER ANY KEY TO BEGIN."
850 CALL SOUND(1,-2,30)
860 CALL KEY(0,K,S)
870 IF S=0 THEN 860
880 CALL SCREEN(6)
890 CALL CLEAR
900 PRINT "     COMPUTER          YOU"
910 PRINT :::::::::::::::::::::::::
920 FOR X=5 TO 14
930 CALL VCHAR(X,5,X+60)
940 CALL HCHAR(X,6,128,10)
950 CALL HCHAR(X,18,128,10)
960 CALL VCHAR(X,17,X+60)
970 NEXT X
980 FOR X=6 TO 15
990 CALL VCHAR(15,X+12,X+42)
1000 CALL VCHAR(15,X,X+42)
1010 NEXT X
1020 S1$="CARRIER"
1030 S2$="BATTLESHIP"
1040 S3$="CRUISER"
```

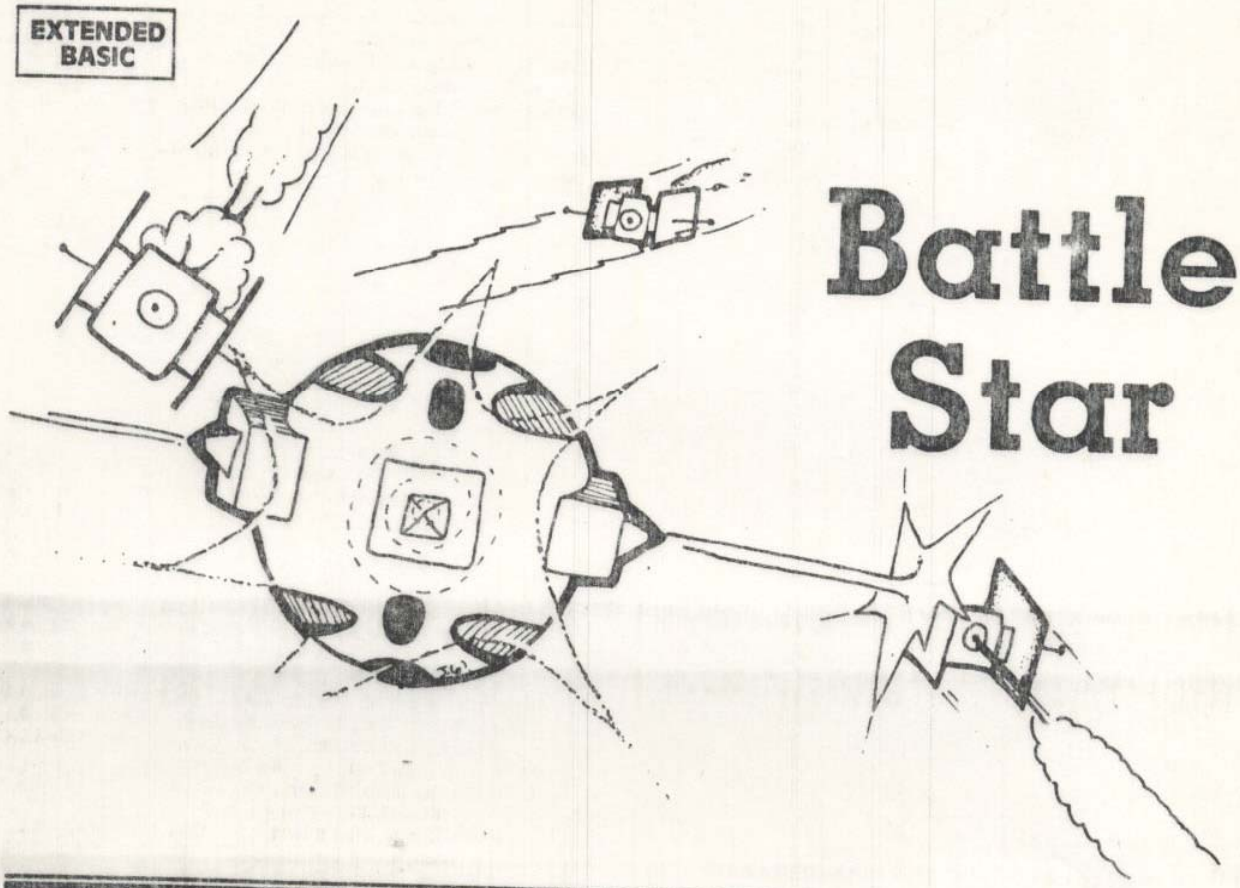
```
1050 S4$="SUBMARINE"
1060 S5$="DESTROYER"
1070 FOR S=1 TO 5
1080 ON S GOSUB 1110,1160,1210,1260,131
     0
1090 GOSUB 1390
1100 GOTO 1360
1110 PR$=S1$
1120 LE=5
1130 S=1
1140 OS=0
1150 RETURN
1160 PR$=S2$
1170 LE=4
1180 S=2
1190 OS=8
1200 RETURN
1210 PR$=S3$
1220 LE=3
1230 S=3
1240 OS=16
1250 RETURN
1260 PR$=S4$
1270 LE=3
1280 S=4
1290 OS=22
1300 RETURN
1310 PR$=S5$
1320 LE=2
1330 S=5
1340 OS=28
1350 RETURN
1360 NEXT S
1370 CALL HCHAR(22,1,32,64)
1380 GOTO 2240
1390 L=LEN(PR$)
1400 SUS$="ENTER ROW,COL. FOR "&STR$(LE
     )&" SPACES"
1410 FOR X=1 TO LEN(SUS$)
1420 SU1$=SEG$(SUS$,X,1)
1430 CALL VCHAR(22,X+2,ASC(SU1$))
1440 NEXT X
1450 PR$=PR$&" SPACE"
1460 CALL HCHAR(23,2,32,30)
1470 FOR X=1 TO LEN(PR$)
1480 SU1$=SEG$(PR$,X,1)
1490 CALL VCHAR(23,X+2,ASC(SU1$))
1500 NEXT X
1510 FOR X=1 TO LE
1520 CALL HCHAR(23,20,35)
1530 CALL VCHAR(23,21,LE-X+49)
1540 CALL KEY(0,K1,ST)
1550 IF ST=0 THEN 1540
1560 IF K1<65 THEN 1590
1570 IF K1>74 THEN 1590
1580 GOTO 1610
1590 CALL SOUND(100,-2,2)
1600 GOTO 1540
1610 CALL VCHAR(23,23,K1)
1620 CALL KEY(0,KE,ST)
1630 IF ST=-1 THEN 1620
1640 CALL KEY(0,K2,ST)
1650 IF ST=0 THEN 1640
1660 IF K2<48 THEN 1690
1670 IF K2>57 THEN 1690
1680 GOTO 1710
1690 CALL SOUND(100,-2,2)
1700 GOTO 1640
1710 CALL VCHAR(23,24,K2)
1720 SH(S,X,1)=K1-64
1730 SH(S,X,2)=K2-47
1740 IF P(K1-64,K2-47)>0 THEN 1590
1750 P(K1-64,K2-47)=S
1760 NEXT X
1770 GOSUB 5370
1780 IF SH(S,1,1)=SH(S,2,1)THEN 1810
1790 X2=1
1800 GOTO 1820
1810 X2=2
1820 FOR X3=1 TO LE
1830 F=0
1840 FOR X1=1 TO LE-X3
1850 IF SH(S,X1,X2)=0 THEN 1910
1860 IF SH(S,X1,X2)<SH(S,X1+1,X2)THEN 1
     910
1870 SW=SH(S,X1,X2)
1880 SH(S,X1,X2)=SH(S,X1+1,X2)
1890 SH(S,X1+1,X2)=SW
1900 F=1
1910 NEXT X1
1920 IF F=0 THEN 1940
1930 NEXT X3
1940 FOR X=1 TO LE-1
1950 IF SH(S,X,1)<>SH(S,X+1,1)-1 THEN 1
     980
1960 NEXT X
1970 GOTO 2070
1980 FOR X=1 TO LE-1
1990 IF SH(S,X,2)<>SH(S,X+1,2)-1 THEN 2
     020
2000 NEXT X
2010 GOTO 2070
2020 CALL SOUND(100,-2,2)
2030 FOR X=1 TO LE
2040 P(SH(S,X,1),SH(S,X,2))=0
2050 NEXT X
2060 GOTO 1460
2070 X=S
2080 FOR X1=1 TO 5
2090 IF SH(X,X1,1)=0 THEN 2190
2100 IF SH(X,1,1)=SH(X,2,1)THEN 2130
2110 OSA=1
2120 GOTO 2140
2130 OSA=0
2140 P(SH(X,X1,1),SH(X,X1,2))=X
2150 IF X>1 THEN 2180
2160 CALL VCHAR(SH(X,X1,1)+4,SH(X,X1,2)
     +17,95+X1+OS+((LE-1)*OSA))
2170 GOTO 2190
2180 CALL HCHAR(SH(X,X1,1)+4,SH(X,X1,2)
     +17,95+X1+OS+(LE*OSA))
2190 NEXT X1
2200 IF X>1 THEN 2230
2210 CALL HCHAR(SH(1,4,1)+4,SH(1,4,2)+1
     7,97+((LE-1)*OSA))
2220 CALL HCHAR(SH(1,5,1)+4,SH(1,5,2)+1
     7,99+((LE-1)*OSA))
2230 RETURN
2240 LE=4
2250 S=1
2260 GOSUB 2400
2270 LE=3
2280 S=2
2290 GOSUB 2400
2300 LE=2
2310 S=3
2320 GOSUB 2400
2330 LE=2
2340 S=4
2350 GOSUB 2400
2360 LE=1
2370 S=5
2380 GOSUB 2400
2390 GOTO 2630
2400 RANDOMIZE
2410 X2=INT(RND*2)+1
2420 IF X2=2 THEN 2460
2430 X=INT(RND*(10-LE))+1
2440 X1=INT(RND*10)+1
2450 GOTO 2480
2460 X=INT(RND*10)+1
2470 X1=INT(RND*(10-LE))+1
2480 ON X2 GOTO 2490,2560
2490 FOR Y=X TO X+LE
2500 IF O(Y,X1)>0 THEN 2400
2510 NEXT Y
2520 FOR Y=X TO X+LE
2530 O(Y,X1)=S
```

```
2540 NEXT Y
2550 RETURN
2560 FOR Y=X1 TO X1+LE
2570 IF O(X,Y)>0 THEN 2400
2580 NEXT Y
2590 FOR Y=X1 TO X1+LE
2600 O(X,Y)=S
2610 NEXT Y
2620 RETURN
2630 M1$="MY SHOT"
2640 M2$="YOUR SHOT"
2650 M3$="SCORE"
2660 M4$="COMPUTER"
2670 M5$="USER"
2680 M6$="YOU MISSED"
2690 M7$="I MISSED"
2700 M8$="**HIT**"
2710 GOTO 2800
2720 FOR V=1 TO 7
2730 CALL HCHAR(18,V+4,ASC(SEG$(M1$,V,1
))))
2740 NEXT V
2750 RETURN
2760 FOR V=1 TO 9
2770 CALL HCHAR(21,V+4,ASC(SEG$(M2$,V,1
))))
2780 NEXT V
2790 RETURN
2800 FOR X=1 TO 5
2810 CALL HCHAR(18,X+22,ASC(SEG$(M3$,X,
1))))
2820 NEXT X
2830 FOR X=1 TO 8
2840 CALL HCHAR(19,X+15,ASC(SEG$(M4$,X,
1))))
2850 NEXT X
2860 FOR X=1 TO 4
2870 CALL HCHAR(19,X+26,ASC(SEG$(M5$,X,
1))))
2880 NEXT X
2890 T=1
2900 IF T=0 THEN 2930
2910 T=0
2920 GOTO 3200
2930 T=1
2940 CALL HCHAR(21,3,32,12)
2950 CALL HCHAR(22,3,32,7)
2960 GOSUB 2720
2970 IF W>0 THEN 3650
2980 RANDOMIZE
2990 X=INT(10*RND)+1
3000 X1=INT(10*RND)+1
3010 H=X
3020 H1=X1
3030 IF P(X,X1)=7 THEN 2980
3040 IF P(X,X1)=6 THEN 2980
3050 CALL HCHAR(19,6,H+64)
3060 CALL HCHAR(19,7,H1+47)
3070 IF P(X,X1)>0 THEN 4480
3080 GOSUB 3120
3090 GOTO 2900
3100 P(X+10,X1)=7
3110 CALL HCHAR(23,1,32,32)
3120 P(X,X1)=6
3130 CALL SOUND(200,-6,2)
3140 CALL HCHAR(23,1,32,32)
3150 CALL VCHAR(X+4,X1+17,144)
3160 FOR Y=1 TO 8
3170 CALL VCHAR(23,12+Y,ASC(SEG$(M7$,Y,
1))))
3180 NEXT Y
3190 RETURN
3200 CALL HCHAR(18,3,32,12)
3210 CALL HCHAR(19,3,32,7)
3220 GOSUB 2760
3230 CALL KEY(0,K1,ST)
3240 IF ST=0 THEN 3230
3250 IF K1<65 THEN 3230
3260 IF K1>74 THEN 3230
3270 CALL VCHAR(22,6,K1)
3280 CALL KEY(0,KE,ST)
3290 IF ST=-1 THEN 3280
3300 CALL KEY(0,K2,ST)
3310 IF ST=0 THEN 3300
3320 IF K2<48 THEN 3300
3330 IF K2>57 THEN 3300
3340 CALL VCHAR(22,7,K2)
3350 K3=K1-64
3360 K4=K2-47
3370 IF O(K3,K4)<6 THEN 3410
3380 CALL SOUND(50,110,2)
3390 CALL HCHAR(22,6,32,7)
3400 GOTO 3200
3410 IF O(K3,K4)=0 THEN 3520
3420 CALL SOUND(200,220,2,330,2,440,2,-
6,2)
3430 CALL SOUND(400,110,2,220,2,330,2,-
8,2)
3440 CALL VCHAR(K3+4,K4+5,136)
3450 SF=O(K3,K4)
3460 O(K3,K4)=7
3470 CALL HCHAR(23,1,32,32)
3480 FOR X2=1 TO 7
3490 CALL HCHAR(23,13+X2,ASC(SEG$(M8$,X
2,1)))
3500 NEXT X2
3510 GOTO 4620
3520 CALL SOUND(200,-6,2)
3530 CALL HCHAR(23,1,32,32)
3540 O(K3,K4)=6
3550 FOR X2=1 TO 10
3560 CALL VCHAR(23,13+X2,ASC(SEG$(M6$,X
2,1)))
3570 NEXT X2
3580 CALL VCHAR(K3+4,K4+5,144)
3590 GOTO 2900
3600 CH=1
3610 GOTO 4670
3620 CH=0
3630 ON SF GOSUB 1110,1160,1210,1260,13
10
3640 IF DS(SF)=LE-1 THEN 3800
3650 IF H=10 THEN 3690
3660 IF P(H+1,H1)<>7 THEN 3680
3670 IF W>1 THEN 4280 ELSE 4080
3680 IF H=1 THEN 3740
3690 IF P(H-1,H1)<>7 THEN 3740
3700 IF W>1 THEN 4280 ELSE 4080
3710 W2=W
3720 W=W1
3730 GOTO 3580
3740 IF H1=10 THEN 3780
3750 IF P(H,H1+1)<>7 THEN 3770
3760 IF W>1 THEN 4080 ELSE 4280
3770 IF H1=1 THEN 3800
3780 IF P(H,H1-1)<>7 THEN 3800
3790 IF W>1 THEN 4080 ELSE 4280
3800 L1=INT(RND*2)+1
3810 ON L1 GOTO 3820,3900
3820 X2=INT(RND*2)+1
3830 ON X2 GOTO 3840,3870
3840 X2=1
3850 X3=0
3860 GOTO 3970
3870 X2=-1
3880 X3=0
3890 GOTO 3970
3900 X3=INT(RND*2)+1
3910 ON X3 GOTO 3920,3950
3920 X3=1
3930 X2=0
3940 GOTO 3970
3950 X3=-1
3960 X2=0
3970 IF H+X2>10 THEN 3800
3980 IF H+X2<1 THEN 3800
3990 IF H1+X3>10 THEN 3800
4000 IF H1+X3<1 THEN 3800
```

```
4010 IF P(H+X2,H1+X3)=6 THEN 3800
4020 IF P(H+X2,H1+X3)=7 THEN 3890
4030 X=H+X2
4040 X1=H1+X3
4050 IF P(X,X1)>0 THEN 4480
4060 GOSUB 3120
4070 GOTO 2900
4080 IF H=10 THEN 4180
4090 H=H+1
4100 IF P(H,H1)=7 THEN 4080
4110 IF P(H,H1)=6 THEN 4180
4120 X=H
4130 X1=H1
4140 IF P(X,X1)>0 THEN 4480
4150 GOSUB 3120
4160 H=H-1
4170 GOTO 2900
4180 IF H=1 THEN 4090
4190 H=H-1
4200 IF P(H,H1)=7 THEN 4180
4210 IF P(H,H1)=6 THEN 4080
4220 X=H
4230 X1=H1
4240 IF P(X,X1)>0 THEN 4480
4250 GOSUB 3120
4260 H=H+1
4270 GOTO 2900
4280 IF H1=10 THEN 4380
4290 H1=H1+1
4300 IF P(H,H1)=7 THEN 4280
4310 IF P(H,H1)=6 THEN 4380
4320 X=H
4330 X1=H1
4340 IF P(X,X1)>0 THEN 4480
4350 GOSUB 3120
4360 H1=H1-1
4370 GOTO 2900
4380 IF H1=1 THEN 4280
4390 H1=H1-1
4400 IF P(H,H1)=7 THEN 4380
4410 IF P(H,H1)=6 THEN 4280
4420 X=H
4430 X1=H1
4440 IF P(X,X1)>0 THEN 4480
4450 GOSUB 3120
4460 H1=H1+1
4470 GOTO 2900
4480 CALL VCHAR(4+X,17+X1,136)
4490 CALL HCHAR(23,1,32,32)
4500 GOSUB 2720
4510 FOR Z=1 TO LEN(M$)
4520 CALL HCHAR(23,14+Z,ASC(SEG$(M$,Z,
     1)))
4530 NEXT Z
4540 CALL SOUND(200,220,2,330,2,440,2,-
     8,2)
4550 CALL SOUND(300,110,0,220,0,330,0,-
     8,0)
4560 SF=P(X,X1)
4570 CALL VCHAR(19,6,X+64)
4580 CALL VCHAR(19,7,X1+47)
4590 P(X,X1)=7
4600 H=X
4610 H1=X1
4620 FOR X2=1 TO 5
4630 DS(X2)=0
4640 NEXT X2
4650 FOR X2=1 TO 10
4660 FOR X3=1 TO 10
4670 IF CH=1 THEN 4690
4680 IF T=0 THEN 4740
4690 IF P(X2,X3)=0 THEN 4780
4700 IF P(X2,X3)=6 THEN 4780
4710 IF P(X2,X3)=7 THEN 4780
4720 DS(P(X2,X3))=DS(P(X2,X3))+1
4730 GOTO 4780
4740 IF O(X2,X3)=0 THEN 4780
4750 IF O(X2,X3)=6 THEN 4780
4760 IF O(X2,X3)=7 THEN 4780
4770 DS(O(X2,X3))=DS(O(X2,X3))+1
4780 NEXT X3
4790 NEXT X2
4800 IF CH=1 THEN 3620
4810 W=0
4820 SCORE=0
4830 FOR Z4=1 TO 5
4840 ON Z4 GOSUB 1110,1160,1210,1260,13
     10
4850 IF DS(Z4)=LE THEN 4940
4860 IF DS(Z4)=0 THEN 4890
4870 W=W+1
4880 GOTO 4940
4890 SCORE=SCORE+1
4900 IF T=0 THEN 4930
4910 GOSUB 5010
4920 GOTO 4940
4930 GOSUB 5000
4940 NEXT Z4
4950 IF T=0 THEN 4980
4960 W1=W
4970 GOTO 2900
4980 W=W1
4990 GOTO 2900
5000 SCP=SCORE
5010 CALL HCHAR(23,1,32,32)
5020 FOR X3=1 TO LEN(PR$)+10
5030 CALL HCHAR(23,X3+6,ASC(SEG$(PR$&"
     DESTROYED",X3,1)))
5040 NEXT X3
5050 IF T=0 THEN 5090
5060 CALL VCHAR(20,20,SCORE+48)
5070 IF SCORE=5 THEN 5120
5080 RETURN
5090 CALL HCHAR(20,27,SCORE+48)
5100 IF SCORE=5 THEN 5140
5110 RETURN
5120 PRINT "THE COMPUTER WINS AGAIN"
5130 GOTO 5150
5140 PRINT "YOU JUST GOT LUCKY THIS TIM
     E."
5150 PRINT "IF YOU WISH TO PLAY AGAIN."
5160 PRINT "ENTER ""Y"", IF NOT ENTER ""
     N"""
5170 INPUT NG$
5180 IF NG$="N" THEN 5350
5190 IF NG$="Y" THEN 5220
5200 CALL SOUND(200,110,0)
5210 GOTO 5150
5220 FOR L=1 TO 10
5230 FOR L1=1 TO 10
5240 P(L,L1)=0
5250 O(L,L1)=0
5260 NEXT L1
5270 NEXT L
5280 FOR L=1 TO 5
5290 FOR L1=1 TO 5
5300 SH(L,L1,1)=0
5310 SH(L,L1,2)=0
5320 NEXT L1
5330 NEXT L
5340 GOTO 880
5350 CALL CLEAR
5360 STOP
5370 NNN=0
5380 AAA=0
5390 FOR X=1 TO LE-1
5400 IF NNN=1 THEN 5430
5410 IF SH(S,X,1)=SH(S,X+1,1)THEN 5460
5420 IF AAA=1 THEN 2020
5430 IF SH(S,X,2)<>SH(S,X+1,2)THEN 2020
5440 NNN=1
5450 GOTO 5470
5460 AAA=1
5470 NEXT X
5480 RETURN
5490 END
```

EXTENDED BASIC

Battle Star



You are the chief security officer in charge of defending the Earth's newest Battle Star from alien attack. At first, the aliens are few—trying only to probe your weak points. Later, they attack in force from all four directions. It's their somewhat ancient nuclear missiles against your laser battery. One hit by a missile, however, and the entire Battle Star is obliterated. The speed at which you can react and move your fingers is the only thing that stands between victory and total destruction . . .

To fire a laser in any one of four directions, press any of the arrow keys. These are the only keys used. You may not move your Battle Star because of your geosynchronous orbit and large size. The entire game is an hand-eye coordination exercise. At one point in the game, the aliens become so fast you may not be able to move fast enough to prevent annihilation. There is, however, an "automatic speed check" put into the program; if you can reach this level and maintain it, the endurance of your fingers will be your only limiting factor. If you wish to make the game even more difficult, you could adjust the limiting speed of the missiles. This is done in lines 730, 760, 790, and 820. The X and Y velocity in the sprite being defined (whichever X or Y is not zero) can be adjusted. For example, in line 730 the X velocity formula is 11 – (L/10). This will allow no speed greater than 10. Change this to 15 – (L/10) and the maximum speed will be 14, with the initial speed being 5. If one line is changed, related lines must all be changed.

The Program

The program is very short and simple—requiring only 3K memory and Extended BASIC. There is plenty of room for a good programmer to experiment and try adding to or improving the features. The action is simple, but can become fairly rapid—thus making the game very challenging. A Battle Star is positioned at the center of the screen, and made up of 9 sprites (3x3). I did this for dramatic reasons: When the Battle Star is hit, each section of it blows up and flies in a different direction. An alien ship will appear to the left, right, above, and below the Battle Star. At first, only the ship will be displayed; later, the ship and a nuclear missile will appear. For every missile knocked out of action, your score will increase by 20 points. For every alien ship destroyed, you will receive 50 points. The trickiest part of the program was to make the laser rays coming from the Battle Star stop after encountering a missile. Since the missile is a sprite, its location is checked using the CALL POSITION statement. Then, calculating the distance from the Battle Star's cannon and dividing by 9 gives me the distance (in number of characters) of the missile. I then use a CALL HCHAR, or CALL VCHAR (first with the ray bolt) and a CHR$(32) which represents a space. Finally, I delete the given sprite. The result is a fast laser bolt and increased program speed.

One problem I encountered when the missiles were traveling at high speed was that they sometimes passed through the base without a hit being detected. This problem was alleviated by checking POSITION instead of COINC, so that if the position was past the edge of the Battle Star, the missile would blow up.

EXPLANATION OF THE PROGRAM

Battle Star

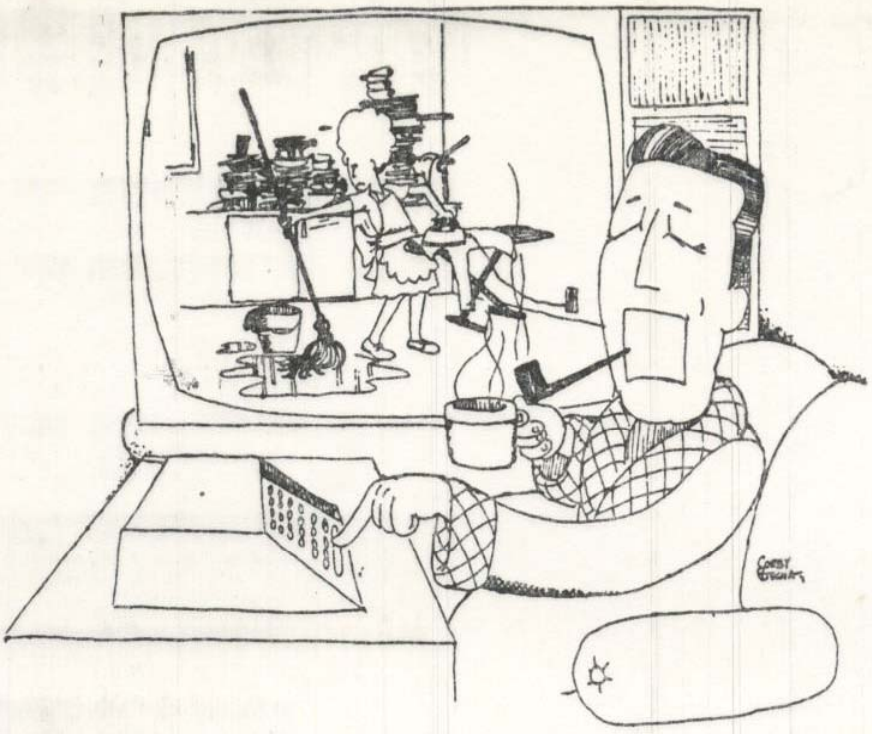
Line Nos.

170-290	Initializes colors and characters.
300	Initializes variables.
310	Jumps to subroutine to create Battle Star.
320-340	Main program loop.
350-380	Sets up sprites to create the Battle Star.
390-440	Reads keyboard; branches to fire laser cannon.
450-490	Fires laser up.
500-540	Fires laser left.
550-590	Fires laser down.
600-640	Fires laser right.
650-690	Checks position of missiles, and branches off if Battle Star hit.
700	Checks the chance of another ship appearing.
710	Decides which ship will appear, and branches to subroutine.
720-740	Places top ship on screen with missile if game progressed.
750-770	Places left ship on screen with missile if game progressed.
780-800	Places bottom ship on screen with missile if game progressed.
810-830	Places right ship on screen with missile if game progressed.
840-870	Battle Star is hit and destroyed.
880-910	Displays score. Play again? Accepts answer.
920-940	Re-initializes variables.
950	End.

```
100 REM *********************
110 REM *                     *
120 REM *  BATTLE STAR        *
130 REM *                     *
140 REM *********************
150 REM
160 REM
170 RANDOMIZE
180 DIR=1 :: CALL CLEAR
190 CALL COLOR(9,7,1):: CALL COLOR(10,
    6,1):: CALL SCREEN(2)
200 CALL CHAR(96,"000000000000070707"):::
    CALL CHAR(97,"181818183C7EFFDB99")
210 CALL CHAR(98,"0000000000E0E0E0"):::
    CALL CHAR(99,"070E1CFFFF1C0E07")
220 CALL CHAR(104,"18423C9999C4218"):
    :: CALL CHAR(101,"E07038FFFF38700E"
    ):: CALL CHAR(102,"070707")
230 CALL CHAR(107,"1046282440A92304")
240 CALL CHAR(103,"99DBFF7E3C181818"):
    :: CALL CHAR(100,"E0E0E0")
250 CALL CHAR(112,"30787C477C7830"):::
    CALL CHAR(113,"1010386CEEEE7C")
260 CALL CHAR(114,"0C1E3EE23E1E0C"):::
    CALL CHAR(115,"007CEEEE6C381010")
270 CALL CHAR(116,"1010387FE3810101"):::
    CALL CHAR(117,"0000183CFF7E2442")
280 CALL CHAR(105,"1818181818181818"):
    :: CALL CHAR(106,"00000FFFF")
290 FOR COL=1 TO 12 :: CALL COLOR(COL,
    16,1):: NEXT COL
300 L=100 :: S=5 :: SC=0 :: SA1,SB1,SA
    2,SB2,SA3,SB3,SA4,SB4=0 :: T=0
310 GOSUB 350
320 GOSUB 390 :: GOSUB 650
330 L=L-.5 :: IF L<1 THEN L=1
340 DISPLAY AT(24,3):SC :: GOTO 320
350 CALL SPRITE(#10,96,16,81,113,0,0,#
    11,97,16,81,121,0,0,#12,98,16,81,1
    29,0,0)
360 CALL SPRITE(#13,99,16,89,113,0,0,#
    14,104,7,89,121,0,0,#15,101,16,89,
    129,0,0)
370 CALL SPRITE(#16,102,16,97,113,0,0,
    #17,103,16,97,121,0,0,#18,100,16,9
    7,129,0,0)
380 RETURN
390 CALL KEY(0,K,S):: IF S=0 THEN RETU
    RN
400 IF K=69 THEN 450
410 IF K=83 THEN 500
420 IF K=88 THEN 550
430 IF K=68 THEN 600
440 RETURN
450 IF SA1=0 AND SB1=0 THEN CALL VCHAR
    (1,16,105,10):: CALL SOUND(10,800,
    0):: CALL VCHAR(1,16,32,10):: SC=S
    C-10 ::: RETURN
460 IF SB1=0 THEN CALL VCHAR(2,16,105,
    9):: CALL SOUND(500,110,2,-5,2):::
    CALL VCHAR(2,16,32,9):: SC=SC+50 :
    : SA1=0 :: RETURN
470 CALL POSITION(#1,P1,P2):: IF P1>76
    THEN 840
480 P1=INT(P1/8)+1 :: CALL VCHAR(P1,16
    ,105,10-P1):: CALL SOUND(200,110,1
    0,-5,8):: CALL VCHAR(P1,16,32,10-P
    1)
490 CALL DELSPRITE(#1):: SC=SC+20 :: S
    B1=0 :: RETURN
500 IF SA2=0 AND SB2=0 THEN CALL HCHAR
    (12,1,106,14):: CALL SOUND(10,800,
    0):: CALL HCHAR(12,1,32,14):: SC=S
    C-10 :: RETURN
510 IF SB2=0 THEN CALL HCHAR(12,3,106,
    12):: CALL SOUND(500,110,2,-5,2):::
    CALL HCHAR(12,3,32,12):: SC=SC+50
    :: SA2=0 :: RETURN
520 CALL POSITION(#2,P1,P2):: IF P2>86
    THEN 840
530 P2=INT(P2/8)+1 :: CALL HCHAR(12,P2
    ,106,15-P2):: CALL SOUND(200,110,1
    0,-5,8):: CALL HCHAR(12,P2,32,15-P
    2)
540 CALL DELSPRITE(#2):: SC=SC+20 :: S
    B2=0 :: RETURN
550 IF SA3=0 AND SB3=0 THEN CALL VCHAR
    (14,16,105,10):: CALL SOUND(10,800
    ,0):: CALL VCHAR(14,16,32,10):: SC
    =SC-10 :: RETURN
560 IF SB3=0 THEN CALL VCHAR(14,16,105
    ,10):: CALL SOUND(500,110,2,-5,2):
    : CALL VCHAR(14,16,32,10):: SC=SC+
    50 :: SA3=0 :: RETURN
570 CALL POSITION(#3,P1,P2):: IF P1<11
    0 AND P1>0 THEN 840
580 P1=INT(P1/8)+1 :: CALL VCHAR(14,16
    ,105,P1-14):: CALL SOUND(200,110,1
    0,-5,8):: CALL VCHAR(14,16,32,P1-1
    4)
590 CALL DELSPRITE(#3):: SC=SC+20 :: S
    B3=0 :: RETURN
600 IF SA4=0 AND SB4=0 THEN CALL HCHAR
    (12,18,106,14):: CALL SOUND(10,800
    ,0):: CALL HCHAR(12,18,32,14):: SC
    =SC-10 :: RETURN
610 IF SB4=0 THEN CALL HCHAR(12,18,106
    ,13):: CALL SOUND(500,110,2,-5,2):
    : CALL HCHAR(12,18,32,13):: SC=SC+
    50 :: SA4=0 :: RETURN
620 CALL POSITION(#4,P1,P2):: IF P8<14
    2 AND P8>0 THEN 840
630 P2=INT(P2/8):: CALL HCHAR(12,18,10
    6,P2-15):: CALL SOUND(200,110,10,-
    5,8):: CALL HCHAR(12,18,32,P2-15)
```

```
640 CALL DELSPRITE(#4)::  SC=SC+20  ::  S
    B4=0  ::  RETURN
650 IF SB1=0 THEN P1,P2=0  ::  GOTO 660
    ELSE CALL POSITION(#1,P1,P2)
660 IF SB2=0 THEN P3,P4=0  ::  GOTO 670
    ELSE CALL POSITION(#2,P3,P4)
670 IF SB3=0 THEN P5,P6=0  ::  GOTO 680
    ELSE CALL POSITION(#3,P5,P6)
680 IF SB4=0 THEN P7,P8=0  ::  GOTO 690
    ELSE CALL POSITION(#4,P7,P8)
690 IF P1>76 OR P4>86 OR(P5<110 AND P5
    >0)OR(P8<142 AND P8>0)THEN 840
700 NS=INT(RND*L)::  IF NS>10 THEN RETU
    RN
710 NS=INT(RND*4)+1  ::  ON NS GOTO 730,
    760,790,820
720 IF SA1=1 AND SB1=1 THEN RETURN
730 CALL HCHAR(2,16,115)::  SA1=1  ::  IF
     L<80 AND SB1=0 THEN CALL SPRITE(#
    1,116,7,17,120,11-(L/10),0)::  SB1=
    1
740 RETURN
750 IF SA2=1 AND SB2=1 THEN RETURN
760 CALL HCHAR(12,3,112)::  SA2=1  ::  IF
     L<80 AND SB2=0 THEN CALL SPRITE(#
    2,116,7,88,17,0,11-(L/10))::  SB2=1
770 RETURN
780 IF SA3=1 AND SB3=1 THEN RETURN
790 CALL HCHAR(23,16,113)::  SA3=1  ::  I
    F L<80 AND SB3=0 THEN CALL SPRITE(
    #3,116,7,175,120,-11+(L/10),0)::  S
    B3=1
800 RETURN
810 IF SA4=1 AND SB4=1 THEN RETURN
820 CALL HCHAR(12,30,114)::  SA4=1  ::  I
    F L<80 AND SB4=0 THEN CALL SPRITE(
    #4,116,7,88,216,0,-11+(L/10))::  SB
    4=1
830 RETURN
840 CALL DELSPRITE(#1,#2,#3,#4)::  CALL
     SOUND(2000,110,2,220,2,1000,30,-4
    ,2)
850 FOR BUB=10 TO 18  ::  CALL MOTION(#B
    UB,INT(RND*40)-20,INT(RND*40)-20)::
     CALL PATTERN(#BUB,107)::  NEXT BU
    B
860 CALL SOUND(1000,110,2,220,2,110,2,
    -5,2)::  CALL SOUND(1,40000,30)
870 CALL DELSPRITE(ALL)::  CALL CLEAR
880 DISPLAY AT(12,7):"YOUR SCORE IS":T
    AB(10);SC
890 CALL DELSPRITE(ALL)
900 DISPLAY AT(22,1):"DO YOU WISH TO P
    LAY AGAIN?  (Y/N)."
910 ACCEPT AT(23,8)VALIDATE("YN"):ANS$
     ::  IF ANS$="N" THEN 950
920 CALL CLEAR  ::  GOSUB 350  ::  SC=0  ::
     L=100
930 SB1,SB2,SB3,SB4,P1,P2,P3,P4,P5,P6,
    P7,P8=0
940 RETURN
950 END
```

TI BASIC



The HARRIED HOUSEWIFE

This matching game is dedicated to tired housewives everywhere who face the daily battle of keeping their houses clean amidst the unrelenting attacks from their kids, husbands, dogs, cats, visiting relatives, unexpected friends, and even home computers—those new family additions that seem to be forever spawning dust, out-of-place furniture, and loose papers.

Harried Housewife uses the color graphics of TI BASIC to depict eight household chores: dusting, sewing, washing clothes, doing dishes, cooking, vacuuming, shopping, and ironing. It is a matching game that even young children will enjoy playing. The rules are simple: An array of 16 squares is displayed on the screen. Each square represents one of the eight chores, and there are two of each chore somewhere in the array. The object of the game is to find each pair. You do this by choosing two squares at a time and entering the corresponding two letters. As a letter is entered, the chore for that square is shown. If a match is made, the chore is considered finished and is listed on the right side of the screen. If a match is not made, the two selections are covered, and two more letters may be chosen.

When all eight pairs are matched, the housework is complete; you have a clean house and the game is over. But you mustn't take too long, because when the kids come home (determined by the counter in line 1420) everything gets scrambled and the harried housewife must start over. . . And as all harried housewives undoubtedly know: it's not easy to get a completely clean house. Often the goal is to become somewhat more attainable—just seeing how can be accomplished before the kids come home.

If you get too harried and want to quit, press S for stop. The arrangement of the current array will be displayed. After you have examined it, SHIFT C (BREAK) to end the program. If you really feel you must win more often — that is, end up with everything matched to signify that elusive "clean house" — you can keep the kids out of the house longer by increasing the number in line 1420. Then enjoy the fantasy of a completely clean house all the time. What? Why can't your home computer make this fantasy actually come true? Be patient. It's just a matter of time. . . Anyway, in the words of a once-popular song: "Such are the dreams of the everyday housewife. . ."

Programming Techniques

This program illustrates the capabilities of TI-99/4A color graphics. Characters are defined in each of the eight user-defined character sets, and each set has a different color scheme. These eight sets are used for the eight chores; and for ease in programming, they are numbered 1 through 8.

Two characters in Set 2 are also redefined with a blue foreground and a red background ("FFFFFFFFFFFFFFFF" and "0") to draw the 16-square checkerboard array. It is drawn with a triple-nested FOR-NEXT loop (statements 2040-2150).

The eight chores to be drawn are called in subroutines (statements 2290 to 3060). The subroutines use x- and y-coordinates to define the placement of the special characters. The coordinates are specified before the subroutine is called. The coordinates of the chores for each of the sixteen squares are listed in subroutines also (statements 5350-5980).

To set up the array of 16 squares, two arrays are actually used: WORK(16) and HH(16). The WORK array is given the numbers of the eight chores: WORK(1) = 1; WORK(2) = 2; . . . WORK(9) = 1; WORK(10) = 2; and so on (statements 3370-3400). For the HH array, a subscript RR is chosen as a random number from 1 to 16. HH(RR) is then set equal to WORK(RR), and then WORK(RR) = 0 so it won't be chosen again. This process continues until all 16 numbers of the HH array have been filled randomly with the numbers from the WORK array (statements 3410-3470). These numbers are the chore numbers for the squares. For example, HH(4) = 7 means that behind the 4th square (D) would be chore number 7 (shopping).

The WORK array is then reset equal to the HH array so the chores can be printed in order on the squares for a "clean house" or for "stop".

As the game is being played, the HH elements are set equal to zero if a match is made, so the match can only be scored once. If a player chooses a square which has previously been part of a matched pair, the word "DONE" appears across the square.

EXPLANATION OF THE PROGRAM
Harried Housewife

Line Nos.

150-180	Prints title screen.
190-260	Defines colors for eight household chores.
270-820	Defines special characters for drawing the chores.
830	Displays the eight chores on title screen.
840-850	Sets counters for the number of trial guesses and the number of successful matches.
860	Dimensions arrays to handle 16 elements.
870-880	Redefines characters for checkerboard.
890-900	Delays for title screen.
910-920	Clears screen and makes it yellow.
930	Defines colors for checkerboard.
940	Draws checkerboard and labels it.
950	Assigns the chores for each square in array.
960-1010	Prints HOUSEWORK.
1020-1130	Prints MATCH 2 LETTERS.
1140-1160	Prints two red lines for the letters chosen.
1170-1200	Waits for letter A-P to be pressed.
1210	Prints the chosen letter.
1220-1230	Finds chore number and coordinates for square chosen.
1240-1260	If the square has been previously matched, prints DONE.
1270	Draws first chore on square.
1280-1330	Waits for second letter to be pressed and prints it.
1340-1350	Finds the chore number and coordinates for that square.
1360-1390	Prints DONE.
1400	Draws second chore on square.
1410	Checks for a match.
1420	Increments the number of trials.
1430	If TIME = 10 prints message to hurry.
1440	If TIME = 12, kids come home.
1450	Branches if TIME is less than 10.
1460	Clears previous message.
1470-1520	Prints OH NO! KIDS ARE HOME!
1530-1550	Reprints checkerboard and scrambles chores for a new game.
1560	Prints PRESS ENTER TO CONTINUE and waits for response, covers squares for next choice.
1570-1620	Prints SPEED-KIDS WILL BE HOME SOON!
1630	Same as 1540
1640-1750	Correct match is made, sounds tone of A, prints finished chore.
1760-1770	Sets elements matched to zero so they can't be scored again.
1780	Returns for next choice.
1790-1840	If all eight matches have been made, prints CLEAN HOUSE!!
1850	Prints S if player wants to stop.
1860	Resets HH array to current arrangement.
1870-1910	Shows all chores in array.
1920-1980	Clears all other printing.
1990-2040	Prints HOUSEWORK NEVER ENDS.
2050	Holds screen until SHIFT C (BREAK) is pressed.

Subroutines:

2060-2170	Prints checkerboard.
2180-2230	Prints letters A to P on squares.
2240-2300	Prints S = STOP and returns.
2310-2410	Draws feather duster.
2420-2490	Draws sewing machine.
2500-2600	Draws T-shirt for washing.
2620-2690	Draws cup and saucer for dishes.
2700-2780	Draws pan for cooking.
2790-2880	Draws vacuum cleaner.
2890-2980	Draws shopping basket.
2990-3080	Draws ironing board.
3090-3300	Places symbols on title screen.
3310-3380	Plays music for title screen.
3390-3420	Puts two sets of chore numbers in WORK array.
3430-3490	Randomly arranges chores in HH array, two of each chore.
3500-3520	Resets WORK array equal to HH array.
3530	Restarts number of matches.
3540-3580	Clears printed list of matches made.
3590-3620	Resets HH array to original WORK array for printing.
3630-4490	When a match is made, blinks the picture and prints the chore in the "Finished" list; prints labels under pictures in the squares.
4500-4570	Prints PRESS ENTER TO CONTINUE and waits for response.
4580-4590	Clears messages.
4600-4630	Covers squares again and relabels them.
4640	Return for next choice.
4650-5280	Subroutines for covering particular square.
5290-5320	Colors blue square.
5330-5360	Colors red square.
5370-6000	Designates the chore number and coordinates for the square chosen.

```
100 REM  *********************
110 REM  *  HARRIED HOUSEWIFE  *
120 REM  *********************
130 REM
140 REM
150 CALL CLEAR
160 PRINT TAB(10);"HARRIED"
170 PRINT ::::TAB(9);"HOUSEWIFE"
180 PRINT ::::::::::::
190 CALL COLOR(9,7,15)
200 CALL COLOR(10,13,12)
210 CALL COLOR(11,14,11)
220 CALL COLOR(12,16,3)
230 CALL COLOR(13,7,12)
240 CALL COLOR(14,5,8)
250 CALL COLOR(15,15,16)
260 CALL COLOR(16,3,16)
270 CALL CHAR(96,"0000004040E0E0E0FFFFFF")
280 CALL CHAR(97,"0000000000020E0C0C")
290 CALL CHAR(98,"02010101070301010101")
300 CALL CHAR(99,"FFFFFFFFFFFFFFFFFF")
310 CALL CHAR(100,"F0E0C0F0F1E080C")
320 CALL CHAR(101,"03070E1C3870E0C")
330 CALL CHAR(102,"FF0E03")
340 CALL CHAR(103,"0")
350 CALL CHAR(104,"FFFFFFFFFFFFFFFFFF")
360 CALL CHAR(105,"FCFCFCFFFFFFFFFFFF")
370 CALL CHAR(106,"FCFCF8F8F0E0C")
380 CALL CHAR(107,"FFFFFFFFFCFCFCFC")
390 CALL CHAR(108,"FFFFFFFFFFF")
400 CALL CHAR(109,"0")
410 CALL CHAR(112,"000000000F0F0F0F")
420 CALL CHAR(113,"0000000081C3FFFF")
430 CALL CHAR(114,"00000000F0F0F0F0")
440 CALL CHAR(115,"0F0F")
450 CALL CHAR(116,"FFFFFFFFFFFFFFFF")
460 CALL CHAR(117,"F0F")
470 CALL CHAR(118,"FFFFFFFFF")
480 CALL CHAR(119,"0")
490 CALL CHAR(120,"00000000000000C0F0F")
500 CALL CHAR(121,"000000000000000FFFF")
510 CALL CHAR(122,"0000000000000C0C0FC")
520 CALL CHAR(123,"0F0F0F0F0F0F0F0F07")
530 CALL CHAR(124,"FFFFFFFFFFFFFFFF")
540 CALL CHAR(125,"FEC6C6C6DCF8E08")
550 CALL CHAR(126,"FFFF")
560 CALL CHAR(128,"1F1F1F1F1F1F1F1F")
570 CALL CHAR(129,"FFFFFCFCFCFCFCFC")
580 CALL CHAR(130,"FFFF")
590 CALL CHAR(131,"1F1F0F")
600 CALL CHAR(132,"FCFCF8")
610 CALL CHAR(133,"0")
620 CALL CHAR(136,"1F0F0101000000000303")
630 CALL CHAR(137,"80C0C0C0C0F0F8F8F8")
640 CALL CHAR(138,"0303030303030301")
650 CALL CHAR(139,"F8F0F0F0F0F0E0E0")
660 CALL CHAR(140,"E0C7CF7FFFFFFFFF")
670 CALL CHAR(141,"008C0C0F0F0E")
680 CALL CHAR(142,"0")
690 CALL CHAR(144,"000000000040C1A19")
700 CALL CHAR(145,"090F09090F09090F")
710 CALL CHAR(146,"FE252424FF2424FF")
720 CALL CHAR(147,"00E09E92FE9292FE")
730 CALL CHAR(148,"06090906")
740 CALL CHAR(149,"9")
750 CALL CHAR(152,"0000000001F1F1F1F")
760 CALL CHAR(153,"000000000F0FFFFFF")
770 CALL CHAR(154,"00000000000F0FEFE")
780 CALL CHAR(155,"1F1F1F0402010")
790 CALL CHAR(156,"FFFFE2040810A04")
800 CALL CHAR(157,"090A0C0808")
810 CALL CHAR(158,"A119070101")
820 CALL CHAR(159,"0")
830 GOSUB 3090
840 TIME=0
850 MATCH=0
860 DIM HH(16),WORE(16)
870 CALL CHAR(43,"FFFFFFFFFFFFFFFF")
880 CALL CHAR(44,"0")
890 CALL SOUND(4225,44000,30)
900 CALL SOUND(4,44000,30)
910 CALL CLEAR
920 CALL SCREEN(12)
930 CALL COLOR(2,6,9)
940 GOSUB 2060
950 GOSUB 3390
960 DATA 72,79,85,83,69,87,79,82,75
970 RESTORE 960
980 FOR Y=23 TO 31
990 READ GR
1000 CALL HCHAR(2,Y,GR)
1010 NEXT Y
1020 DATA 77,65,84,67,72,32,50
1030 RESTORE 1020
1040 FOR Y=23 TO 29
1050 READ GR
1060 CALL HCHAR(5,Y,GR)
1070 NEXT Y
1080 DATA 76,69,84,84,69,82,83
1090 RESTORE 1080
1100 FOR Y=23 TO 29
1110 READ GR
1120 CALL HCHAR(6,Y,GR)
1130 NEXT Y
1140 CALL COLOR(8,7,1)
1150 CALL HCHAR(8,25,95)
1160 CALL HCHAR(8,27,95)
1170 CALL KEY(0,K1,ST)
1180 IF K1=83 THEN 1850
1190 IF K1<65 THEN 1170
1200 IF K1>80 THEN 1170
1210 CALL HCHAR(8,25,K1)
1220 SS=1
1230 ON (K1-64)GOSUB 5370,5410,5450,549
     0,5530,5570,5610,5650,5690,5730,57
     70,5810,5850,5890,5930,5970
1240 IF CH(1)<>0 THEN 1270
1250 GOSUB 4450
1260 GOTO 1280
1270 ON CH(1)GOSUB 2310,2420,2500,2610,
     2700,2790,2890,2990
1280 CALL KEY(0,K2,ST)
1290 IF K2=83 THEN 1850
1300 IF K2<65 THEN 1280
1310 IF K2>80 THEN 1280
1320 IF K2=K1 THEN 1280
1330 CALL HCHAR(8,27,K2)
1340 SS=2
1350 ON (K2-64)GOSUB 5370,5410,5450,549
     0,5530,5570,5610,5650,5690,5730,57
     70,5810,5850,5890,5930,5970
1360 IF CH(2)<>0 THEN 1400
1370 GOSUB 4450
1380 GOTO 1420
1390 IF CH(1)=0 THEN 1420
1400 ON CH(2)GOSUB 2310,2420,2500,2610,
     2700,2790,2890,2990
1410 IF CH(1)=CH(2)THEN 1640
1420 TIME=TIME+1
1430 IF TIME=10 THEN 1570
1440 IF TIME=12 THEN 1460
1450 GOTO 4500
1460 CALL HCHAR(22,2,32,31)
1470 DATA 79,72,32,78,79,33,32,75,73,68
     ,83,32,65,82,69,32,72,79,77,69,33
1480 RESTORE 1470
1490 FOR Y=3 TO 23
1500 READ GR
1510 CALL HCHAR(24,Y,GR)
1520 NEXT Y
1530 GOSUB 2060
1540 GOSUB 3390
1550 TIME=0
1560 GOTO 4500
1570 DATA 83,80,69,69,68,59,32,75,73,68
     ,83,32,87,73,76,76,32,66,69,32,72,
     79,77,69,32,83,79,79,78,33
1580 RESTORE 1570
```

```
1590 FOR Y=2 TO 31
1600 READ GR
1610 CALL HCHAR(22,Y,GR)
1620 NEXT Y
1630 GOTO 4500
1640 CALL SOUND(1000,440,2)
1650 MATCH=MATCH+1
1660 IF MATCH<>1 THEN 1730
1670 DATA 70,73,78,73,83,72,69,68,58
1680 RESTORE 1670
1690 FOR Y=23 TO 31
1700 READ GR
1710 CALL HCHAR(11,Y,GR)
1720 NEXT Y
1730 X=MATCH+9
1740 Y=26
1750 ON CH(1)GOSUB 3630,3720,3800,3890,
     4050,4130,4270,4360
1760 HH(K1-64)=0
1770 HH(K2-64)=0
1780 IF MATCH<>8 THEN 1420
1790 DATA 67,76,69,65,78,32,72,79,85,83
     ,69,33,33
1800 RESTORE 1790
1810 FOR Y=3 TO 27 STEP 2
1820 READ GR
1830 CALL HCHAR(24,Y,GR)
1840 NEXT Y
1850 CALL HCHAR(8,25,83)
1860 GOSUB 3590
1870 FOR S=1 TO 16
1880 SS=3
1890 ON S GOSUB 5370,5410,5450,5490,553
     0,5570,5610,5650,5690,5730,5770,58
     10,5850,5890,5930,5970
1900 ON CH(SS)GOSUB 2310,2420,2500,2610
     ,2700,2790,2890,2990
1910 NEXT S
1920 CALL HCHAR(21,3,32,6)
1930 CALL HCHAR(22,2,32,31)
1940 IF MATCH<>8 THEN 1990
1950 FOR X=2 TO 8
1960 CALL HCHAR(X,23,32,9)
1970 NEXT X
1980 GOTO 2050
1990 DATA 72,79,85,83,69,87,79,82,75,32
     ,78,69,86,69,82,32,69,78,68,83,33
2000 RESTORE 1990
2010 FOR Y=3 TO 23
2020 READ GR
2030 CALL HCHAR(24,Y,GR)
2040 NEXT Y
2050 GOTO 2050
2060 FOR Z=1 TO 11 STEP 10
2070 FOR X=Z TO Z+4
2080 FOR Y=2 TO 12 STEP 10
2090 CALL SOUND(100,1047,2)
2100 CALL HCHAR(X,Y,43,5)
2110 CALL HCHAR(X,Y+5,44,5)
2120 CALL SOUND(100,523,2)
2130 CALL HCHAR(X+5,Y,44,5)
2140 CALL HCHAR(X+5,Y+5,43,5)
2150 NEXT Y
2160 NEXT X
2170 NEXT Z
2180 DATA 3,4,3,9,3,14,3,19,8,4,8,9,8,1
     4,8,19,13,4,13,9,13,14,13,19,18,4,
     18,9,18,14,18,19
2190 RESTORE 2180
2200 FOR CC=65 TO 80
2210 READ X,Y
2220 CALL HCHAR(X,Y,CC)
2230 NEXT CC
2240 CALL HCHAR(21,3,83)
2250 CALL HCHAR(21,4,61)
2260 CALL HCHAR(21,5,83)
2270 CALL HCHAR(21,6,84)
2280 CALL HCHAR(21,7,79)
2290 CALL HCHAR(21,8,80)
2300 RETURN
2310 CALL HCHAR(X-1,Y,96)
2320 CALL HCHAR(X-1,Y-1,103)
2330 CALL HCHAR(X-1,Y+1,97)
2340 CALL HCHAR(X,Y-1,98)
2350 CALL HCHAR(X,Y,99)
2360 CALL HCHAR(X,Y+1,100)
2370 CALL HCHAR(X+1,Y-1,101)
2380 CALL HCHAR(X+1,Y,102)
2390 CALL HCHAR(X+1,Y+1,103)
2400 GOSUB 3670
2410 RETURN
2420 CALL HCHAR(X-1,Y-1,104,2)
2430 CALL HCHAR(X,Y,109)
2440 CALL HCHAR(X-1,Y+1,105)
2450 CALL HCHAR(X,Y-1,106)
2460 CALL HCHAR(X,Y+1,107)
2470 CALL HCHAR(X+1,Y-1,108,3)
2480 GOSUB 3760
2490 RETURN
2500 CALL HCHAR(X-1,Y-1,112)
2510 CALL HCHAR(X-1,Y,113)
2520 CALL HCHAR(X-1,Y+1,114)
2530 CALL HCHAR(X,Y-1,115)
2540 CALL HCHAR(X,Y,116)
2550 CALL HCHAR(X,Y+1,117)
2560 CALL HCHAR(X+1,Y-1,119)
2570 CALL HCHAR(X+1,Y+1,119)
2580 CALL HCHAR(X+1,Y,118)
2590 GOSUB 3840
2600 RETURN
2610 CALL HCHAR(X-1,Y-1,120)
2620 CALL HCHAR(X-1,Y,121)
2630 CALL HCHAR(X-1,Y+1,122)
2640 CALL HCHAR(X,Y-1,123)
2650 CALL HCHAR(X,Y,124)
2660 CALL HCHAR(X,Y+1,125)
2670 CALL HCHAR(X+1,Y-1,126,3)
2680 GOSUB 4000
2690 RETURN
2700 CALL HCHAR(X,Y-1,128)
2710 CALL HCHAR(X-1,Y-1,133,3)
2720 CALL HCHAR(X+1,Y+1,133)
2730 CALL HCHAR(X,Y,129)
2740 CALL HCHAR(X,Y+1,130)
2750 CALL HCHAR(X+1,Y-1,131)
2760 CALL HCHAR(X+1,Y,132)
2770 GOSUB 4090
2780 RETURN
2790 CALL HCHAR(X-1,Y-1,136)
2800 CALL VCHAR(X-1,Y+1,142,2)
2810 CALL HCHAR(X+1,Y-1,142)
2820 CALL HCHAR(X-1,Y,137)
2830 CALL HCHAR(X,Y-1,138)
2840 CALL HCHAR(X,Y,139)
2850 CALL HCHAR(X+1,Y,140)
2860 CALL HCHAR(X+1,Y+1,141)
2870 GOSUB 4230
2880 RETURN
2890 CALL HCHAR(X-1,Y-1,144)
2900 CALL HCHAR(X-1,Y,149,2)
2910 CALL HCHAR(X+1,Y,149)
2920 CALL HCHAR(X,Y-1,145)
2930 CALL HCHAR(X,Y,146)
2940 CALL HCHAR(X,Y+1,147)
2950 CALL HCHAR(X+1,Y-1,148)
2960 CALL HCHAR(X+1,Y+1,148)
2970 GOSUB 4310
2980 RETURN
2990 CALL HCHAR(X-1,Y-1,152)
3000 CALL HCHAR(X-1,Y,153)
3010 CALL HCHAR(X-1,Y+1,154)
3020 CALL HCHAR(X,Y-1,155)
3030 CALL HCHAR(X,Y,156)
3040 CALL HCHAR(X+1,Y-1,157)
3050 CALL HCHAR(X+1,Y,158)
3060 CALL VCHAR(X,Y+1,159,2)
3070 GOSUB 4400
3080 RETURN
```

```
3090 X=3
3100 Y=5
3110 GOSUB 2310
3120 X=4
3130 Y=16
3140 GOSUB 2420
3150 Y=27
3160 GOSUB 2500
3170 X=8
3180 Y=7
3190 GOSUB 2610
3200 X=10
3210 Y=26
3220 GOSUB 2700
3230 X=17
3240 GOSUB 2790
3250 X=16
3260 Y=15
3270 GOSUB 2890
3280 X=15
3290 Y=6
3300 GOSUB 2990
3310 CALL SOUND(300,494,2,196,7)
3320 CALL SOUND(200,440,2)
3330 CALL SOUND(200,392,2)
3340 CALL SOUND(300,440,2,185,8)
3350 CALL SOUND(200,392,3)
3360 CALL SOUND(200,370,3)
3370 CALL SOUND(1000,392,3,165,9)
3380 RETURN
3390 FOR Z=1 TO 8
3400 WORK(Z)=Z
3410 WORK(Z+8)=Z
3420 NEXT Z
3430 RANDOMIZE
3440 FOR R=1 TO 16
3450 RR=INT(16*RND)+1
3460 IF WORK(RR)=0 THEN 3450
3470 HH(R)=WORK(RR)
3480 WORK(RR)=0
3490 NEXT R
3500 FOR R=1 TO 16
3510 WORK(R)=HH(R)
3520 NEXT R
3530 MATCH=0
3540 FOR X=11 TO 18
3550 CALL HCHAR(X,23,32,9)
3560 NEXT X
3570 CALL HCHAR(24,3,32,22)
3580 RETURN
3590 FOR R=1 TO 16
3600 HH(R)=WORK(R)
3610 NEXT R
3620 RETURN
3630 CALL COLOR(9,15,7)
3640 CALL COLOR(9,7,15)
3650 CALL COLOR(9,15,7)
3660 CALL COLOR(9,7,15)
3670 CALL HCHAR(X+2,Y-1,68)
3680 CALL HCHAR(X+2,Y,85)
3690 CALL HCHAR(X+2,Y+1,83)
3700 CALL HCHAR(X+2,Y+2,84)
3710 RETURN
3720 CALL COLOR(10,12,13)
3730 CALL COLOR(10,13,12)
3740 CALL COLOR(10,12,13)
3750 CALL COLOR(10,13,12)
3760 CALL HCHAR(X+2,Y-1,83)
3770 CALL HCHAR(X+2,Y,69)
3780 CALL HCHAR(X+2,Y+1,87)
3790 RETURN
3800 CALL COLOR(11,11,14)
3810 CALL COLOR(11,14,11)
3820 CALL COLOR(11,11,14)
3830 CALL COLOR(11,14,11)
3840 CALL HCHAR(X+2,Y-1,87)
3850 CALL HCHAR(X+2,Y,65)
3860 CALL HCHAR(X+2,Y+1,83)
3870 CALL HCHAR(X+2,Y+2,72)
3880 RETURN
3890 CALL COLOR(12,3,16)
3900 CALL COLOR(12,16,3)
3910 CALL COLOR(12,3,16)
3920 CALL COLOR(12,16,3)
3930 CALL HCHAR(X+2,25,68)
3940 CALL HCHAR(X+2,26,73)
3950 CALL HCHAR(X+2,27,83)
3960 CALL HCHAR(X+2,28,72)
3970 CALL HCHAR(X+2,29,69)
3980 CALL HCHAR(X+2,30,83)
3990 RETURN
4000 CALL HCHAR(X+2,Y-1,68)
4010 CALL HCHAR(X+2,Y,73)
4020 CALL HCHAR(X+2,Y+1,83)
4030 CALL HCHAR(X+2,Y+2,72)
4040 RETURN
4050 CALL COLOR(13,12,7)
4060 CALL COLOR(13,7,12)
4070 CALL COLOR(13,12,7)
4080 CALL COLOR(13,7,12)
4090 CALL HCHAR(X+2,Y-1,67)
4100 CALL HCHAR(X+2,Y,79,2)
4110 CALL HCHAR(X+2,Y+2,75)
4120 RETURN
4130 CALL COLOR(14,8,5)
4140 CALL COLOR(14,5,8)
4150 CALL COLOR(14,8,5)
4160 CALL COLOR(14,5,8)
4170 CALL HCHAR(X+2,25,86)
4180 CALL HCHAR(X+2,26,65)
4190 CALL HCHAR(X+2,27,67)
4200 CALL HCHAR(X+2,28,85,2)
4210 CALL HCHAR(X+2,30,77)
4220 RETURN
4230 CALL HCHAR(X+2,Y-1,86)
4240 CALL HCHAR(X+2,Y,65)
4250 CALL HCHAR(X+2,Y+1,67)
4260 RETURN
4270 CALL COLOR(15,16,15)
4280 CALL COLOR(15,15,16)
4290 CALL COLOR(15,16,15)
4300 CALL COLOR(15,15,16)
4310 CALL HCHAR(X+2,Y-1,83)
4320 CALL HCHAR(X+2,Y,72)
4330 CALL HCHAR(X+2,Y+1,79)
4340 CALL HCHAR(X+2,Y+2,80)
4350 RETURN
4360 CALL COLOR(16,16,3)
4370 CALL COLOR(16,3,16)
4380 CALL COLOR(16,16,3)
4390 CALL COLOR(16,3,16)
4400 CALL HCHAR(X+2,Y-1,73)
4410 CALL HCHAR(X+2,Y,82)
4420 CALL HCHAR(X+2,Y+1,79)
4430 CALL HCHAR(X+2,Y+2,78)
4440 RETURN
4450 CALL HCHAR(X,Y-1,68)
4460 CALL HCHAR(X,Y,79)
4470 CALL HCHAR(X,Y+1,78)
4480 CALL HCHAR(X,Y+2,69)
4490 RETURN
4500 DATA 80,82,69,83,83,32,69,78,84,69
     ,82,32,84,79,32,67,79,78,84,73,78,
     85,69,32
4510 RESTORE 4500
4520 FOR Y=3 TO 26
4530 READ GR
4540 CALL HCHAR(23,Y,GR)
4550 NEXT Y
4560 CALL KEY(0,KEY,ST)
4570 IF KEY<>13 THEN 4560
4580 CALL HCHAR(4,24,32,5)
4590 CALL HCHAR(23,2,32,25)
4600 ON (K1-64)GOSUB 4650,4690,4730,477
     0,4810,4850,4890,4930,4970,5010,50
     50,5090,5130,5170,5210,5250
```

```
4610 CALL HCHAR(X,Y,K1)
4620 ON (K2-64)GOSUB 4650,4690,4730,477
     0,4810,4850,4890,4930,4970,5010,50
     50,5090,5130,5170,5210,5250
4630 CALL HCHAR(X,Y,K2)
4640 GOTO 1150
4650 X=3
4660 Y=4
4670 GOSUB 5290
4680 RETURN
4690 X=3
4700 Y=9
4710 GOSUB 5330
4720 RETURN
4730 X=3
4740 Y=14
4750 GOSUB 5290
4760 RETURN
4770 X=3
4780 Y=19
4790 GOSUB 5330
4800 RETURN
4810 X=8
4820 Y=4
4830 GOSUB 5330
4840 RETURN
4850 X=8
4860 Y=9
4870 GOSUB 5290
4880 RETURN
4890 X=8
4900 Y=14
4910 GOSUB 5330
4920 RETURN
4930 X=8
4940 Y=19
4950 GOSUB 5290
4960 RETURN
4970 X=13
4980 Y=4
4990 GOSUB 5290
5000 RETURN
5010 X=13
5020 Y=9
5030 GOSUB 5330
5040 RETURN
5050 X=13
5060 Y=14
5070 GOSUB 5290
5080 RETURN
5090 X=13
5100 Y=19
5110 GOSUB 5330
5120 RETURN
5130 X=18
5140 Y=4
5150 GOSUB 5330
5160 RETURN
5170 X=18
5180 Y=9
5190 GOSUB 5290
5200 RETURN
5210 X=18
5220 Y=14
5230 GOSUB 5330
5240 RETURN
5250 X=18
5260 Y=19
5270 GOSUB 5290
5280 RETURN
5290 FOR XX=X-1 TO X+2
```

```
5300 CALL HCHAR(XX,Y-1,43,4)
5310 NEXT XX
5320 RETURN
5330 FOR XX=X-1 TO X+2
5340 CALL HCHAR(XX,Y-1,44,4)
5350 NEXT XX
5360 RETURN
5370 CH(SS)=HH(1)
5380 X=3
5390 Y=4
5400 RETURN
5410 CH(SS)=HH(2)
5420 X=3
5430 Y=9
5440 RETURN
5450 CH(SS)=HH(3)
5460 X=3
5470 Y=14
5480 RETURN
5490 CH(SS)=HH(4)
5500 X=3
5510 Y=19
5520 RETURN
5530 CH(SS)=HH(5)
5540 X=8
5550 Y=4
5560 RETURN
5570 CH(SS)=HH(6)
5580 X=8
5590 Y=9
5600 RETURN
5610 CH(SS)=HH(7)
5620 X=8
5630 Y=14
5640 RETURN
5650 CH(SS)=HH(8)
5660 X=8
5670 Y=19
5680 RETURN
5690 CH(SS)=HH(9)
5700 X=13
5710 Y=4
5720 RETURN
5730 CH(SS)=HH(10)
5740 X=13
5750 Y=9
5760 RETURN
5770 CH(SS)=HH(11)
5780 X=13
5790 Y=14
5800 RETURN
5810 CH(SS)=HH(12)
5820 X=13
5830 Y=19
5840 RETURN
5850 CH(SS)=HH(13)
5860 X=18
5870 Y=4
5880 RETURN
5890 CH(SS)=HH(14)
5900 X=18
5910 Y=9
5920 RETURN
5930 CH(SS)=HH(15)
5940 X=18
5950 Y=14
5960 RETURN
5970 CH(SS)=HH(16)
5980 X=18
5990 Y=19
6000 RETURN
```

EXTENDED BASIC

FORCE 1

You are the Captain of the Force 1, a United Federation of Planets police cruiser. A message has just come in that a large number of alien bandits have ntered your sector and are planning an attack on your home lanet. The bandits cannot be taken alive and therefore must e destroyed. The job won't be easy, so you'd better stay lert.

Since the bandits are armed with short-range laser cannons, they should be encountered when beyond their firing ange. As you become a better pilot, you may choose to ncrease your ship's speed with higher levels of difficulty. This means that the alien craft will be approached much nore rapidly, and more accuracy on your part is needed.

On first sighting, your radar screen will show the alien o be no larger than the background stars, and very difficult to pick out among them. As you approach the ship, will become larger and larger, until the alien is either in ange to fire its laser cannon, or slightly out of range flying ight past you.

To maneuver your ship in order to set your gun sights n the alien bandit, you must use the four arrow keys. If ou hold a key down continually, your ship will keep accelerating in that direction. This will, of course, cause the ar field and the alien ship to move more in the opposite irection. For example, if the alien ship were moving off the right of the screen and you wanted to bring him back to the center, you would hold the D key down until the ien started moving toward the center. Then to halt all ovement by the alien and keep him from going to the left f the screen, press the S key until the alien either stops or ows down to a minimum speed. The idea is to slow his orizontal and vertical speed to a minimum and position m in the center of your gun sight. To fire your laser blaster, ess ENTER. Getting the alien in your gun sight may not as easy as it sounds, for the alien is intelligent and eriodically shifts course like all skillful space bandits. So st when you think you have him, he's off in another rection . . .

You have 1000 units of time to complete your mission before the strike on your planet. If 25 or more bandit ships are destroyed, you will gain an extra 1000 units of time to attack the second wave of aliens.

The Program

The program is written in Extended BASIC. I decided here to make use of the MAGNIFY commands to create a series of space ships that start off very small and gradually become larger. This gives a more realistic view of an object coming closer. I gave the ship a random speed—slow at first when it's at a great distance, and accelerating as it gets closer. I also gave the ship the ability to change directions randomly 10 percent of the time. The ability to use sprites for both the ship and the star field made it possible to create the illusion of actual motion—not just changing the alien's direction in reference to yours, but also with respect to every star in the star field. For example, take the case of the alien ship traveling to the right of the screen and all of the stars not moving. If you press the D key until the alien stops moving, all of the stars will now be moving to the left, and the alien will be still. This works the same way vertically.

By using the COINCidence statement and the tolerance option, I was able to make it more difficult to hit a ship at a greater distance (where it needs to be a direct hit) than to hit one that is nearby. There is however a slight time delay from the time you press the [ENTER] key until the laser fires. This makes it almost impossible to hit a moving target. So the challenge will be to get the alien in your gun sights and hold him there long enough to make a successful strike.

The laser bolts that you fire at the alien are there all of the time, but kept invisible. I then use the CALL COLOR statement twice—once to turn on the bolts, and once to turn them back off.

If the alien ship is still in your gun sight when it reaches maximum size, you will be within range of his laser cannon and be fired upon. WARNING: Laser cannons never miss at short range!

EXPLANATION OF THE PROGRAM

Force 1

Line Nos.

130-210	Display levels of difficulty; accept answer.
220-460	Assign variables, color, and characters.
470-560	Read keyboard, branch to subroutine, or adjust variables.
570-610	Adjust distance to alien; branch to display new alien ship.
620-630	Randomly change motion of alien ship.
640-650	Change motion of stars.
660-670	Display score, time; check for out of time (time = 1000).
680-830	Display laser beams on screen.
840-860	Assign alien space craft to a new location.
870-880	Fire laser, check for hit.
890-910	Alien destroyed; Adjust score, re-initialize variables.
920-1060	Subroutine when hit by alien; branch for bonus, or branch to end-of-game messages.
1070-1180	End-of-game messages.
1190-1210	Check to play again.
1220-1250	Change alien shape.
1260-1270	Check for alien to fire back.
1280-1340	Alien is at maximum size and moves off screen faster.
1350-1420	Alien fires and hits your ship; sound effects.
1430-1520	Display star pattern.
1530-1630	Display title page.

```
100 REM *********
110 REM * FORCE 1 *
120 REM *********
130 CALL CLEAR :: GOSUB 1530
140 DISPLAY AT(2,10):"FORCE 1"
150 DISPLAY AT(4,3):"LEVEL OF DIFFICULTY:"
160 DISPLAY AT(6,5):"1. BEGINNER" :: DISPLAY AT(7,5):"2. NOVICE" :: DISPLAY AT(8,5):"3. INTERMEDIATE"
170 DISPLAY AT(9,5):"4. SEMI-PRO" :: DISPLAY AT(10,5):"5. PRO"
180 ACCEPT AT(11,5)VALIDATE(DIGIT)SIZE(1):L1 :: L=L1*4
190 RANDOMIZE :: CALL CLEAR
200 FOR CO=1 TO 8 :: CALL COLOR(CO,16,1):: NEXT CO
210 COU=0 :: D=1 :: DIS=11000 :: IF SC>=25 THEN L=L*2
220 CALL CHAR(88,"0102040810204080"):: CALL CHAR(89,"8040201008040201")
230 CALL CHAR(90,"03070E1C3870E0C0"):: CALL CHAR(91,"C0E070381C0E0703")
240 CALL CHAR(92,"070F1F3E7CF8F0E0"):: CALL CHAR(93,"E0F0F87C3E1F0F07")
250 CALL CHAR(94,"0306060C1830606C0"):: CALL CHAR(95,"C0603018C060301")
260 CALL COLOR(8,1,1):: CALL SCREEN(2)
270 CALL CHAR(96,"0101010101010101"):: CALL CHAR(97,"8080808080808080")
280 CALL CHAR(98,"0000000000000000FF"):: CALL CHAR(99,"FF")
290 CALL COLOR(9,16,1)
300 CALL VCHAR(7,12,96,9):: CALL VCHAR(7,21,97,9)
310 CALL HCHAR(6,13,98,8):: CALL HCHAR(16,13,99,8)
320 CALL CHAR(33,"FF"):: CALL CHAR(34,"0101010101010101")
330 CALL VCHAR(12,15,33):: CALL VCHAR(12,18,33):: CALL VCHAR(10,16,34):: CALL VCHAR(13,16,34)
340 FOR COL=10 TO 12 :: CALL COLOR(COL,7,1):: NEXT COL
350 GOSUB 360 :: GOTO 450
360 CALL CHAR(104,"0000000008"):: CALL CHAR(105,"0000000018"):: CALL CHAR(106,"000000001C")
370 CALL CHAR(107,"000000003C"):: CALL CHAR(108,"0000183C"):: CALL CHAR(109,"00001C3E")
380 CALL CHAR(110,"00003C7E18"):: CALL CHAR(111,"0018FEFF3C42")
390 CALL CHAR(112,"000C1E7FFF3F40"):: CALL CHAR(113,"00"):: CALL CHAR(114,"0000000C0C0C00"):: CALL CHAR(115,"00")
400 CALL CHAR(116,"0000000061F7FFFFF"):: CALL CHAR(117,"3F2040"):: CALL CHAR(118,"000000000E0F0F0"):: CALL CHAR(119,"C04020")
410 CALL CHAR(120,"00000000010737FFF"):: CALL CHAR(121,"FF1F13306040"):: CALL CHAR(122,"000000000E0FCFF")
420 CALL CHAR(123,"FFF8F80C0602")
430 CALL CHAR(124,"02604CD70033090C01")
440 RETURN
450 CALL COLOR(12,7,1)
460 GOSUB 690 :: CALL MAGNIFY(1):: GOSUB 840 :: GOSUB 1410
470 CALL KEY(0,K,S)
480 CALL POSITION(#1,P01,P02)
490 IF K=13 THEN GOSUB 870
500 T=INT(RND*10):: IF T=4 THEN DEV=L/10-INT(RND*L/5):: DEU=L/10-INT(RND*L/5)ELSE DEV,DEU=0
510 IF K=69 THEN D1=D1+L/5 :: SA=SA+L/5 :: IF SA>127 THEN SA=127
520 IF K=88 THEN D1=D1-L/5 :: SA=SA-L/5 :: IF SA<-128 THEN SA=-128
530 IF K=83 THEN D2=D2+L/5 :: SB=SB+L/5 :: IF SB>127 THEN SB=127
540 IF K=68 THEN D2=D2-L/5 :: SB=SB-L/5 :: IF SB<-128 THEN SB=-128
550 DIS=DIS-(L*15):: D=11-INT(DIS/10000):: IF DIS<200 THEN GOSUB 1260 :: GOTO 570
560 IF D<9 THEN GOSUB 1220 ELSE ON D-8 GOSUB 1230,1240,1250
570 D1=D1+DEV*(D/11):: D2=D2+DEU*(D/11)
580 IF D1>127 THEN D1=127
590 IF D1<-128 THEN D1=-128
600 IF D2>127 THEN D2=127
610 IF D2<-128 THEN D2=-128
620 CALL MOTION(#1,D1,D2)
630 IF S=0 THEN 650
640 FOR SM=2 TO 15 :: CALL MOTION(#SM,SA,SB):: NEXT SM
650 CALL SOUND(-100,800,15)
660 TIME=TIME+1 :: IF TIME=1000 THEN 970
670 DISPLAY AT(1,3):"SCORE:";SC,"TIME:";TIME :: GOTO 470
680 CALL CHARSET
690 DISPLAY AT(24,2):CHR$(92);CHR$(93)
700 DISPLAY AT(23,3):CHR$(92);CHR$(93)
710 DISPLAY AT(22,4):CHR$(92);CHR$(93)
720 DISPLAY AT(21,5):CHR$(92);CHR$(93)
730 DISPLAY AT(20,6):"Z";CHR$(91)
740 DISPLAY AT(19,7):"Z";CHR$(91)
750 DISPLAY AT(18,8):"Z";CHR$(91)
760 DISPLAY AT(17,9):"Z";CHR$(91)
770 DISPLAY AT(16,10)SIZE(1):"^" :: DISPLAY AT(16,19)SIZE(1):"^"
780 DISPLAY AT(15,11)SIZE(8):"^"
790 DISPLAY AT(14,12)SIZE(6):"X";Y"
800 DISPLAY AT(13,13)SIZE(1):"X" :: DISPLAY AT(13,16)SIZE(1):"Y"
```

```
810 DISPLAY AT(12,14)SIZE(2):"XY"
820 CALL HCHAR(11,16,32,2)
830 RETURN
840 IF SP1=0 THEN D1=INT(L-(RND*L*2)):
    :D2=INT(L-(RND*L*2)):: CALL SPRIT
    E(#1,104,7,INT(RND*256)+1,INT(RND*
    256)+1,D1/((11/D)),D2/((11/D)))
850 SP1=1 :: DIS=11000
860 L=L+1 :: RETURN
870 CALL COLOR(8,7,1):: CALL COLOR(8,1
    ,1)
880 CALL COINC(#1,87,124,D,C1)
890 CALL SOUND(20,880,2,990,2,10000,30
    ,-4,2)
900 IF C1=-1 THEN SP1=0 :: CALL MAGNIF
    Y(1):: CALL DELSPRITE(#1):: GOTO 9
    20
910 RETURN
920 SC=SC+1 :: FOR CS=1 TO 5 :: CALL S
    CREEN(7):: CALL SCREEN(2):: NEXT C
    S
930 CALL SOUND(500,110,2,-4,2):: CALL
    HCHAR(12,16,124,2):: CALL HCHAR(11
    ,16,124,2):: CALL SOUND(1000,110,2
    ,220,2,330,2,-8,2)
940 CALL SOUND(1,44000,30):: GOSUB 810
950 SA=0 :: SB=0 :: D=1 :: DIS=11000 :
    :L=L+2 :: GOSUB 840
960 RETURN
970 CALL CLEAR :: CALL SOUND(1000,440,
    2,550,2,660,2):: CALL SOUND(2000,7
    70,2,880,2,990,2)
980 CALL DELSPRITE(ALL)
990 SCO=SCO+SC
1000 IF SCO>=25 AND SCO-SC1>=25 THEN TI
     ME=1000 :: SC1=SCO :: DISPLAY AT(2
     ,3):"BONUS GAME" :: GOTO 210
1010 CALL CHARSET :: CALL SCREEN(6):: C
     ALL DELSPRITE(#1,#2,#3)
1020 IF SCO>=40 THEN 1070
1030 IF SCO>=30 THEN 1090
1040 IF SCO>=20 THEN 1110
1050 IF SCO>=10 THEN 1130
1060 IF SCO>=5 THEN 1150 ELSE 1170
1070 DISPLAY AT(4,1):"A VERY GOOD BATTL
     E.":"YOUR NAME WILL GO DOWN IN":"H
     ISTORY AS ONE OF THE"
1080 DISPLAY AT(7,1):"GREATEST STARSHIP
      CAPTAINS":"OF YOUR TIME.":"YOUR S
     CORE=";SC :: GOTO 1190
1090 DISPLAY AT(4,1):"YOU ARE TO BE CON
     GRADULATED":"ON YOUR FINE MISSION.
     FEW":"PILOTS HAVE ATTAINED SUCH"
1100 DISPLAY AT(7,1):"SUCCESS.GOOD LUCK
      ON FUTURE":"MISSIONS.":" YOUR SCO
     RE=";SC :: GOTO 1190
1110 DISPLAY AT(4,1):"A FAIR SHOWING. T
     HE ALIENS":"HAVE BEEN TURNED BACK
     AND":"YOUR HOME WORLD IS SAFE."
1120 DISPLAY AT(7,1):"YOUR SCORE=";SC :
     : GOTO 1190
1130 DISPLAY AT(4,1):"YOUR FLEET WAS BA
     DLY DAMAGED":"IN THE FIGHT, BUT YO
     U":"MANAGED TO FOGHT OFF THE"
1140 DISPLAY AT(7,1):"ALIEN ATTACK. BET
     TER LUCK":"NEXT TIME.":"YOUR SCORE
     =";SC :: GOTO 1190
1150 DISPLAY AT(4,1):"YOUR FLEET HAS BE
     EN":"DESTROYED. YOU ARE THE":"ONL
     Y SURVIVOR."
1160 DISPLAY AT(7,1):"YOUR HOME PLANET
     IS SAFE":"AT LEAST UNTIL THE NEXT
     :"ATTACK.":"YOUR SCORE=";SC :: GOT
     O 1190
1170 DISPLAY AT(4,1):"ALL HOPE IS LOST
     IN TRYING":"TO SAVE YOUR PLANET.":
     "YOUR MISSION HAS FAILED."
1180 DISPLAY AT(7,1):"AND YOU ARE DISGR
     ACED.":"YOUR SCORE=";SC :: GOTO 11
     90
1190 DISPLAY AT(10,1):"DO YOU WISH TO P
     LAY AGAIN":"ENTER (Y/N).";

1200 ACCEPT AT(11,14)SIZE(1)VALIDATE("Y
     N"):ANS$
1210 IF ANS$="N" THEN END ELSE SC,SC0,S
     C1,TIME=0 :: CALL MAGNIFY(1):: GOT
     O 190
1220 CALL MAGNIFY(1):: CALL PATTERN(#1,
     103+D):: RETURN
1230 CALL PATTERN(#1,112):: CALL MAGNIF
     Y(3):: RETURN
1240 CALL PATTERN(#1,116):: RETURN
1250 CALL PATTERN(#1,120):: RETURN
1260 CALL MAGNIFY(4):: CALL POSITION(#1
     ,PO1,PO2):: IF PO1>3 AND PO2>8 THE
     N CALL LOCATE(#1,PO1-8,PO2-8)
1270 IF PO1<110 AND PO1>36 AND PO2<148
     AND PO2>88 THEN GOTO 1350
1280 IF D1<-19.6 THEN R1=-19.6 :: GOTO
     1300
1290 IF D1>19.6 THEN R1=19.6 ELSE R1=D1
1300 IF D2<-19.6 THEN R2=-19.6 :: GOTO
     1320
1310 IF D2>19.6 THEN R2=19.6
1320 CALL MOTION(#1,R1*5+40*SIN(R1),R2*
     5+40*SIN(R2)):: FOR TD=1 TO 20 ::
     NEXT TD
1330 CALL DELSPRITE(#1):: SP1=0 :: DIS=
     11000 :: D=1 :: FOR MO=1 TO 15 ::
     CALL MOTION(#MO,0,0):: NEXT MO
1340 SA=0 :: SB=0 :: SP1=0 :: CALL MAGN
     IFY(1):: GOSUB 840 :: RETURN
1350 CALL POSITION(#1,D3,D4):: CALL VCH
     AR(D3/8+3,D4/8+2,34,21-D3/8):: CRA
     SH=CRASH+1 :: CALL SOUND(1000,110,
     2,220,2,10000,30,-4,2)
1360 CALL VCHAR(D3/8+3,D4/8+2,32,21-D3/
     8):: GOSUB 1280
1370 CALL SOUND(300,110,2,220,2,20000,3
     0,-8,2)
1380 CALL SOUND(500,440,2,660,2,3000,30
     ,-4,2)
1390 CALL SOUND(600,110,2,220,2,5000,30
     ,-8,2)
1400 CALL SOUND(1000,220,2,230,2,1000,3
     0,-8,2):: SA,SB=0 :: GOTO 970
1410 Z$="8161163BC4241211"
1420 CALL COLOR(13,16,1)
1430 Z1$="0000001000000000"
1440 ST=2
1450 CALL CHAR(128,Z1$)
1460 CALL CHAR(129,"00"):: CALL CHAR(13
     0,"00"):: CALL CHAR(131,"00")
1470 FOR ST=2 TO 15
1480 STA1=INT(RND*256)+1 :: STA2=INT(RN
     D*256)+1
1490 CALL SPRITE(#ST,128,16,STA1,STA2)
1500 NEXT ST
1510 RETURN
1520 END
1530 DISPLAY AT(11,8):"**********"
1540 DISPLAY AT(12,8):"* FORCE 1 *"
1550 DISPLAY AT(13,8):"**********"
1560 GOSUB 360 :: DISPLAY AT(24,1):"PRE
     SS ANY KEY TO CONTINUE"
1570 CALL KEY(0,K,S):: IF S<>0 THEN CAL
     L SOUND(-1,40000,30):: CALL CLEAR
     :: RETURN ELSE CALL MAGNIFY(1)
1580 T1=INT(RND*192)+1 :: T2=INT(RND*25
     6)+1 :: CALL SPRITE(#1,104,2,T1,T2
     ,INT(RND*7)-3,INT(RND*7)-3)
1590 D=0
1600 D=D+1 :: IF D<9 THEN GOSUB 1220 EL
     SE ON D-8 GOSUB 1230,1240,1250
1610 CALL SOUND(-4000,600,(11-D)*3,400,
     (D-1)*3)
1620 CALL KEY(0,K,S):: IF S<>0 THEN CAL
     L DELSPRITE(#1):: CALL SOUND(-1,40
     000,30):: CALL CLEAR :: RETURN
1630 IF D<11 THEN 1600 ELSE CALL DELSPR
     ITE(#1):: GOTO 1570
```

DODGE 'EM

EXTENDED BASIC

Remember going to the amusement park and riding the bumper cars or *Dodge'ems*? Some people like to drive and try not to hit any other cars. Other drivers see how many cars they can hit. This computer version of *Dodge'em* has several cars randomly moving up and down the screen. The object of the game is to drive as quickly as you can from the right to the left of the screen. See what your minimum time is for crossing. A short victory melody will be played if you cross successfully (no crashing). Of course, some of you players may tire of that and try to see how many crashes you can have in each crossing or within a certain time limit.

Programming

My goal for this game was to make a game in Extended BASIC with as short a listing as possible so even the nontypers would not take too long to key in a program to run. This program is a total of 73 statements yet contains 27 moving sprites. The actual game logic is contained in 21 lines (Lines 160 to 360). You could really have fewer lines by stacking statements if you don't mind long lines.

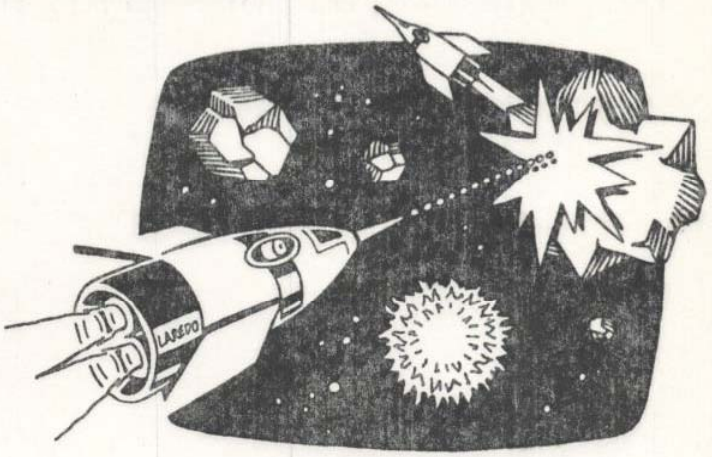
EXPLANATION OF THE PROGRAM
DODGE'EM

Line Nos.	
100-150	Introductory REMarks; branches to title screen.
160	Clears screen.
170-180	Draws left and right borders; prints "TIME:"
190-220	Places 26 cars moving vertically at random speeds.
230-240	Initializes variables; randomly places red car at right side of screen; beeps.
250-300	Depending on key pressed, sets row velocity and column velocity and moves red car.
310	Increments and prints time counter.
320	Checks coincidence for a crash.
330	If there is a crash, sounds a crashing noise and increments number of crashes.
340	If the car is not at the left border, program branches to Line 240.
350-360	Stops the red car and prints the number of crashes.
370-410	If there were no crashes, plays victory melody.
420-460	Asks if player wants to try again and branches appropriately.
470-570	Prints title screen while sounding crashing noises and defining special characters.
580-710	Prints instructions.

```
100 REM *********
110 REM * DODGE'EM *
120 REM *********
130 REM
140 REM
150 GOTO 470
160 CALL CLEAR
170 CALL VCHAR(1,2,112,24):: CALL VCHA
    R(1,31,112,24)
180 DISPLAY AT(1,1):"TIME:"
190 RANDOMIZE
200 FOR I=2 TO 27
210 CALL SPRITE(#I,96,5,9,I*8+1,((-1)^
    I)*INT(RND*12+1),0)
220 NEXT I
230 RV=0 :: CV=0 :: T=0 :: CR=0
240 CALL SPRITE(#1,104,7,INT(RND*180+1
    ),233):: CALL SOUND(150,1397,0)
250 CALL KEY(0,K,S)
260 IF K=69 THEN RV=-5 :: CV=0
270 IF K=88 THEN RV=5 :: CV=0
280 IF K=83 THEN CV=-6 :: RV=0
290 IF K=68 THEN CV=0 :: RV=0
300 CALL MOTION(#1,RV,CV)
310 T=T+1 :: DISPLAY AT(1,7):T
320 CALL COINC(ALL,C):: IF C=0 THEN 34
    0
330 CALL SOUND(150,-6,0):: CR=CR+1
340 CALL POSITION(#1,RO,CO):: IF CO>16
    THEN 250
350 CALL MOTION(#1,0,0)
360 DISPLAY AT(3,1):"CRASHES: ";CR
370 IF CR>0 THEN 420
380 RESTORE 400
390 FOR I=1 TO 19 :: READ N :: CALL SO
    UND(100,N,1):: NEXT I
400 DATA 262,330,392,523,392,523,330,3
    92,523,659
410 DATA 523,659,392,523,659,784,659,7
    84,784
420 DISPLAY AT(24,1):"TRY AGAIN? (Y/N)
    "
430 CALL KEY(0,KE,S)
440 IF KE=89 THEN DISPLAY AT(3,1):"" ::
    : GOTO 230
450 IF KE=78 THEN CALL CLEAR ELSE 430
460 STOP
470 CALL CLEAR :: CALL SOUND(500,-6,1,
    440,5)
480 DISPLAY AT(11,7):"D O D G E ' E M"
490 CALL SOUND(500,-7,1)
500 CALL CHAR(96,"387C7C3838387C7C38")
510 CALL SOUND(500,-5,1)
520 CALL CHAR(104,"0066FFFFFF660")
530 CALL COLOR(11,11,11)
540 CALL CHAR(74,"1038549210101010")
550 CALL CHAR(81,"0808080808849241C08")
560 CALL CHAR(88,"10204080FF402010")
570 CALL CHAR(90,"0804020FF0204080")
580 CALL CLEAR
590 DISPLAY AT(2,7):"D O D G E ' E M"
600 DISPLAY AT(5,2):"DRIVE THE RED CAR
    FROM THE"
610 DISPLAY AT(6,2):"RIGHT SIDE OF THE
    SCREEN"
620 DISPLAY AT(7,2):"TO THE LEFT SIDE
    BY USING"
630 DISPLAY AT(8,2):"THE ARROW KEYS I
    Q X Z."
640 DISPLAY AT(10,2)BEEP:"TRY NOT TO C
    RASH INTO"
650 DISPLAY AT(11,2):"OTHER CARS."
660 DISPLAY AT(13,2):"YOU MAY SLOW DOW
    N OR STOP"
670 DISPLAY AT(14,2):"BY PRESSING Z BU
    T YOU MAY"
680 DISPLAY AT(15,2):"NOT BACK UP."
690 DISPLAY AT(24,2):"PRESS ANY KEY TO
    START."
700 CALL KEY(0,KEY,S):: IF S=0 THEN 70
    0 ELSE 160
710 END
```

TI BASIC

SPACE WAR



Space War is a two-player game written in TI BASIC. Each player has one rocket. The object of the game is to destroy your opponent either by missile fire, forcing him to crash with an asteroid, or by causing him to use up his allotment of fuel.

You can fire missiles in any of the eight directions selectable from each side of the split keyboard. Missiles emit a nerve gas that paralyzes any moving object on the screen until a hit is made or the missile goes out of range—i.e., off the screen. Firing a missile, however, does require an expenditure of fuel.

Each rocket starts out with 50 units of fuel. One unit is subtracted for each move, and a missile shot costs 5 units of fuel, so you must try to move efficiently and shoot accurately. If you run out of fuel, the game ends and the other player receives 2 points.

If your missile hits the enemy rocket, you score 5 points. If you crash into an asteroid, your opponent receives 3 points. And if you crash into each other, no points are awarded. If you shoot an asteroid you lose 1 point but the game does not stop.

Note: If using a disk system, type CALL FILES(1) prior to RUNing. Even so, you still might encounter some conditions during play when the memory will fill and the program will halt. To eliminate this, you can delete all the instructional PRINT statements.

EXPLANATION OF THE PROGRAM
Space War

Line Nos.	
150-180	Clears screen, initializes fuel 50 units; makes black screen.
190-370	Definition of characters and colors for title screen and instructions. Characters 152-159 are arrows.
380-570	Draws title screen.
580-660	Draws border and blinks colors until user presses a key.
670-800	Asks if user wants instructions and waits for response.
810-1270	Prints instructions invisibly and makes white letters appear on black screen.
1280-1350	Clears screen, resets letters to white on black and defines colors for game.
1360-1790	Defines characters for graphics. Characters starting with R are the rockets in different directions; VS is for the missile; S indicates asteroids, and D crashing graphics.
1800-1880	Clears screen for game, initializes variables and draws rockets. A1, B1 are coordinates for crashing; A, B, and C, D are the rockets' coordinates.
1890-2150	Draws center asteroid, then 7 random asteroids, making sure asteroids do not overlap.
2160-2240	Receives players' input. If a key has been pressed, branches accordingly. If no key has been pressed, goes to the other player's keyboard input. Initializes variables. G indicates who is playing. V = 1 when an asteroid has been hit.
2250-2310	Procedure when yellow rocket fires a missile.
2320-2470	Procedure when yellow rocket moves.
2480-2540	Procedure when blue rocket fires missile.
2550-2700	Procedure when blue rocket moves.
2710-3460	Routines for moving the blue rocket different directions.
3470-4740	Routines for shooting missile different directions.
4750-5500	Routines for moving yellow rocket different directions.
5510-5830	Sounds crashing noise and flashes graphics.
5840-5980	Calculates and prints scores and ending remarks.
5990-6040	Prints option to play again and receives user input.
6050-6130	Subroutine to check if asteroid is shot; if so, score is decreased one point.
6140-6200	Subroutine to check if rocket is hit or if rocket hits asteroid.
6210-6260	Subroutine to check if asteroid is in that position; if so, V = 1.
6270-6330	Procedure if rocket runs out of fuel.

```
100 REM     *********
110 REM     * SPACE WAR *
120 REM     *********
130 CALL CLEAR
140 CALL SCREEN(2)
150 CALL CHAR(128,"03030303030303030303")
160 CALL CHAR(129,"C0C0C0C0C0C0C0C0C0C0")
170 CALL CHAR(120,"F0F0F0F0F0F0F0F0F0F0")
180 CALL CHAR(121,"FFFFFFFF")
190 CALL CHAR(152,"081C2A4908080808080808")
200 CALL CHAR(153,"08080808492A1C0808")
210 CALL CHAR(154,"0008040402FF020408")
220 CALL CHAR(155,"0010204040FF402010")
230 CALL CHAR(156,"1F0305091120488801")
240 CALL CHAR(157,"F8C0A0908804020180")
250 CALL CHAR(158,"8040201108050301F8")
260 CALL CHAR(159,"0102048890A0C0F808")
270 A1$=CHR$(157)&"  &CHR$(152)&"  &
    CHR$(156)
280 A2$=CHR$(159)&"  &CHR$(153)&"  &
    CHR$(158)
290 CALL COLOR(13,2,2)
300 CALL COLOR(15,2,2)
310 FOR I=1 TO 8
320 CALL COLOR(I,2,2)
330 NEXT I
340 PRINT TAB(4);"PRESS ANY KEY TO BEG
    IN"::::::
350 DATA 3,6,3,4,6,1,5,6,3,6,8,1,7,6,3
    ,3,11,1,5,11,1,3,15,2,5,15,2,3,19,
    3,7,19,3,3,23,3
360 DATA 5,24,1,7,23,3,13,14,5,9,21,2,
    11,21,2,9,25,3,11,26,1,13,27,1
370 DATA 3,10,5,3,12,3,3,14,5,3,17,5,4
    ,19,3,4,23,3,9,14,4,11,16,2,9,18,4
380 DATA 9,20,5,9,23,5,10,25,4,10,27,2
390 FOR I=1 TO 20
400 READ A,B,C
410 CALL HCHAR(A,B,144,C)
420 NEXT I
430 FOR I=1 TO 13
440 READ A,B,C
450 CALL VCHAR(A,B,144,C)
460 NEXT I
470 CALL HCHAR(12,26,128)
480 CALL HCHAR(12,27,129)
490 CALL COLOR(15,16,16)
500 CALL COLOR(13,16,2)
510 FOR I=1 TO 8
520 CALL COLOR(I,16,2)
530 NEXT I
540 CALL HCHAR(1,2,120,29)
550 CALL HCHAR(24,2,120,29)
560 CALL VCHAR(1,30,121,24)
570 CALL VCHAR(1,2,121,24)
580 CALL COLOR(12,7,2)
590 CALL KEY(0,T,S)
600 IF S=1 THEN 630
610 CALL COLOR(12,2,7)
620 GOTO 580
630 CALL CLEAR
640 FOR I=1 TO 8
650 CALL COLOR(I,2,2)
660 NEXT I
670 PRINT "INSTRUCTIONS?(Y OR N)":::::
    :::::
680 FOR I=1 TO 8
690 CALL COLOR(I,16,2)
700 NEXT I
710 CALL KEY(0,T,S)
720 IF S=0 THEN 710
730 FOR I=1 TO 8
740 CALL COLOR(I,2,2)
750 NEXT I
760 IF T=78 THEN 1230
770 PRINT TAB(8);"* SPACE WAR *"
780 PRINT "  A GAME FOR TWO PLAYERS"
790 PRINT :"THE OBJECT OF THE GAME IS
    "
800 PRINT "TO DESTROY THE OPPONENT BY"
810 PRINT "MISSLE FIRE OR BY FORCING"
820 PRINT "HIM INTO AN ASTEROID.":::
830 PRINT TAB(11);"* KEYS *"::::
840 PRINT "TO MOVE YOUR ROCKET, PRESS"
850 PRINT "KEYS AS SHOWN BELOW:"
860 PRINT "PLAYER 1","PLAYER 2":::
870 PRINT TAB(2);A1$;TAB(16);A1$
880 PRINT "  W  E R";TAB(17);"U  I O"
890 PRINT TAB(2);CHR$(155)&"  S  D &"
    ;CHR$(154);TAB(16);CHR$(155)&"  J  K &"
    CHR$(154)
900 PRINT "  Z  X C";TAB(17);"N  M ,"
910 PRINT TAB(2);A2$;TAB(16);A2$:::
920 PRINT "PLAYER 1 USES Q TO FIRE,"
930 PRINT "AND PLAYER 2 USES P."
940 PRINT "PRESS ANY KEY TO CONTINUE"
950 FOR I=1 TO 8
960 CALL COLOR(I,16,2)
970 NEXT I
980 CALL COLOR(16,16,2)
990 CALL KEY(0,T,S)
1000 IF S=0 THEN 990
1010 CALL COLOR(16,2,2)
1020 FOR I=1 TO 8
1030 CALL COLOR(I,2,2)
1040 NEXT I
1050 PRINT TAB(9);"* ASTEROIDS *"::::
1060 PRINT "THIS WAR TAKES PLACE IN AN"
1070 PRINT "ASTEROID BELT. YOU CANNOT"
1080 PRINT "SHOOT THROUGH THEM, AND"
1090 PRINT "RUNNING INTO ONE IS FATAL."
1100 PRINT ::TAB(9);"*MISSLES*"::::
1110 PRINT "MISSLES CAN BE FIRED AT"
1120 PRINT "ANY ANGLE. THEY EMIT A GAS"
1130 PRINT "THAT PARALYZES ANY MOVING"
1140 PRINT "OBJECTS ON THE SCREEN FROM"
1150 PRINT "THE TIME THAT IT WAS FIRED"
1160 PRINT "UNTIL IT HITS OR DISAPPEARS
    .":::::::
1170 PRINT "PRESS ANY KEY TO CONTINUE"
1180 FOR I=1 TO 8
1190 CALL COLOR(I,16,2)
1200 NEXT I
1210 CALL KEY(0,T,S)
1220 IF S=0 THEN 1210
1230 CALL CLEAR
1240 FOR I=1 TO 8
1250 CALL COLOR(I,16,2)
1260 NEXT I
1270 CALL COLOR(9,5,2)
1280 CALL COLOR(10,11,2)
1290 CALL COLOR(11,16,2)
1300 CALL COLOR(14,7,2)
1310 RU$="0808081C1C3E2A22"
1320 RD$="222A3E1C1C080808"
1330 RR$="0000E0387F38E"
1340 RL$="000007 1CFE1C07"
1350 RUR$="01021478 9C280881"
1360 RUL$="8040281E39141008"
1370 RDR$="1008289C781402 01"
1380 RDL$="0810143 91E284080"
1390 B$="0000001818"
1400 S1$="7F7F3737 1F0F03"
1410 S2$="FEFEFCFCF8F0C"
1420 S3$="000030F1F3F3F7F7F"
1430 S4$="00C0F0F8FCFCFEFE"
1440 D$="995A3CFFFF3C5A99"
1450 D1$="0102040810204081"
1460 D2$="8040201008040201"
1470 D3$="1818181818181818"
1480 D4$="00000FFFF"
1490 CALL CHAR(96,RU$)
1500 CALL CHAR(97,RD$)
1510 CALL CHAR(98,RR$)
1520 CALL CHAR(99,RL$)
1530 CALL CHAR(100,RUR$)
1540 CALL CHAR(101,RUL$)
1550 CALL CHAR(102,RDR$)
```

```
1560 CALL CHAR(103,RDLS$)
1570 CALL CHAR(104,RUS$)
1580 CALL CHAR(105,RDS$)
1590 CALL CHAR(106,RRS$)
1600 CALL CHAR(107,RLS$)
1610 CALL CHAR(108,RURS$)
1620 CALL CHAR(109,RULS$)
1630 CALL CHAR(110,RDRS$)
1640 CALL CHAR(111,RDLS$)
1650 CALL CHAR(112,B$)
1660 CALL CHAR(136,D$)
1670 CALL CHAR(137,D1$)
1680 CALL CHAR(138,D2$)
1690 CALL CHAR(139,D3$)
1700 CALL CHAR(140,D4$)
1710 CALL CHAR(144,S1$)
1720 CALL CHAR(145,S2$)
1730 CALL CHAR(146,S3$)
1740 CALL CHAR(147,S4$)
1750 CALL CLEAR
1760 BPR=50
1770 YPR=50
1780 A1=0
1790 B1=0
1800 A=12
1810 B=3
1820 C=12
1830 D=30
1840 CALL HCHAR(A,B,98)
1850 CALL HCHAR(C,D,107)
1860 CALL COLOR(15,16,2)
1870 CALL HCHAR(13,16,144)
1880 CALL HCHAR(13,17,145)
1890 CALL HCHAR(12,16,146)
1900 CALL HCHAR(12,17,147)
1910 FOR I=1 TO 7
1920 RANDOMIZE
1930 SH=INT(20*RND)+3
1940 SV=INT(23*RND)+5
1950 FOR I1=0 TO 1
1960 CALL GCHAR(SH,SV+I1,P)
1970 GOSUB 6160
1980 IF V=1 THEN 2060
1990 NEXT I1
2000 FOR I1=0 TO 1
2010 CALL GCHAR(SH-1,SV+I1,P)
2020 GOSUB 6160
2030 IF V=1 THEN 2060
2040 NEXT I1
2050 GOTO 2080
2060 V=0
2070 GOTO 1920
2080 CALL HCHAR(SH,SV,144)
2090 CALL HCHAR(SH,SV+1,145)
2100 CALL HCHAR(SH-1,SV,146)
2110 CALL HCHAR(SH-1,SV+1,147)
2120 NEXT I
2130 CALL KEY(2,K1,S1)
2140 G=0
2150 V=0
2160 IF S1<>0 THEN 2220
2170 CALL KEY(1,K,S)
2180 G=0
2190 V=0
2200 IF S<>0 THEN 2450
2210 GOTO 2130
2220 IF K1<>11 THEN 2290
2230 YPR=YPR-5
2240 CALL GCHAR(A,B,P)
2250 CALL HCHAR(C,D,32)
2260 CALL SOUND(200,-6,0,440,0)
2270 G=1
2280 GOTO 3440
2290 YPR=YPR-1
2300 IF YPR<0 THEN 6220
2310 CALL GCHAR(C,D,P)
2320 CALL HCHAR(C,D,32)
2330 CALL SOUND(200,-6,0,440,0)
2340 G=1
2350 IF K1+1=1 THEN 4720
2360 IF K1=5 THEN 4800
2370 IF K1=3 THEN 4880
2380 IF K1=2 THEN 4960
2390 IF K1=6 THEN 5040
2400 IF K1=4 THEN 5150
2410 IF K1=14 THEN 5260
2420 IF K1=15 THEN 5370
2430 CALL HCHAR(C,D,P)
2440 GOTO 2170
2450 IF K<>18 THEN 2520
2460 BPR=BPR-5
2470 IF BPR<0 THEN 6250
2480 E=A
2490 F=B
2500 G=1
2510 GOTO 3440
2520 BPR=BPR-1
2530 IF BPR<0 THEN 6250
2540 CALL GCHAR(A,B,P)
2550 CALL HCHAR(A,B,32)
2560 CALL SOUND(200,-6,0,440,0)
2570 G=2
2580 IF K=5 THEN 2680
2590 IF K+1=1 THEN 2760
2600 IF K=3 THEN 2840
2610 IF K=2 THEN 2920
2620 IF K=6 THEN 3000
2630 IF K=4 THEN 3110
2640 IF K=14 THEN 3220
2650 IF K=15 THEN 3330
2660 CALL HCHAR(A,B,P)
2670 GOTO 2130
2680 A=A-1
2690 IF A<>0 THEN 2710
2700 A=24
2710 CALL GCHAR(A,B,P)
2720 GOSUB 6090
2730 IF V=1 THEN 5480
2740 CALL HCHAR(A,B,96)
2750 GOTO 2130
2760 A=A+1
2770 IF A<>25 THEN 2790
2780 A=1
2790 CALL GCHAR(A,B,P)
2800 GOSUB 6090
2810 IF V=1 THEN 5480
2820 CALL HCHAR(A,B,97)
2830 GOTO 2130
2840 B=B+1
2850 IF B<>33 THEN 2870
2860 B=1
2870 CALL GCHAR(A,B,P)
2880 GOSUB 6090
2890 IF V=1 THEN 5480
2900 CALL HCHAR(A,B,98)
2910 GOTO 2130
2920 B=B-1
2930 IF B<>0 THEN 2950
2940 B=32
2950 CALL GCHAR(A,B,P)
2960 GOSUB 6090
2970 IF V=1 THEN 5480
2980 CALL HCHAR(A,B,99)
2990 GOTO 2130
3000 A=A-1
3010 B=B+1
3020 IF A<>0 THEN 3040
3030 A=24
3040 IF B<>33 THEN 3060
3050 B=1
3060 CALL GCHAR(A,B,P)
3070 GOSUB 6090
3080 IF V=1 THEN 5480
3090 CALL HCHAR(A,B,100)
3100 GOTO 2130
3110 A=A-1
3120 B=B-1
3130 IF A<>0 THEN 3150
```

```
3140 A=24
3150 IF B<>0 THEN 3170
3160 B=32
3170 CALL GCHAR(A,B,P)
3180 GOSUB 6000
3190 IF V=1 THEN 5480
3200 CALL HCHAR(A,B,101)
3210 GOTO 2130
3220 A=A+1
3230 B=B+1
3240 IF A<>25 THEN 3260
3250 A=1
3260 IF B<>33 THEN 3280
3270 B=1
3280 CALL GCHAR(A,B,P)
3290 GOSUB 6000
3300 IF V=1 THEN 5480
3310 CALL HCHAR(A,B,102)
3320 GOTO 2130
3330 A=A+1
3340 B=B-1
3350 IF A<>25 THEN 3370
3360 A=1
3370 IF B<>0 THEN 3390
3380 B=32
3390 CALL GCHAR(A,B,P)
3400 GOSUB 6000
3410 IF V=1 THEN 5480
3420 CALL HCHAR(A,B,103)
3430 GOTO 2130
3440 CALL GCHAR(E,F,Z)
3450 CALL SOUND(100,440,0,880,0)
3460 IF Z=96 THEN 3480
3470 IF Z<>104 THEN 3610
3480 FOR I=E-1 TO 1 STEP -1
3490 CALL GCHAR(I,F,P)
3500 GOSUB 6000
3510 CALL HCHAR(I,F,112)
3520 CALL HCHAR(I,F,32)
3530 IF A<>I THEN 3560
3540 IF B<>F THEN 3560
3550 GOTO 5480
3560 IF C<>I THEN 3590
3570 IF D<>F THEN 3590
3580 GOTO 5480
3590 NEXT I
3600 GOTO 2170
3610 IF Z=97 THEN 3630
3620 IF Z<>105 THEN 3760
3630 FOR I=E+1 TO 24
3640 CALL GCHAR(I,F,P)
3650 GOSUB 6000
3660 CALL HCHAR(I,F,112)
3670 CALL HCHAR(I,F,32)
3680 IF A<>I THEN 3710
3690 IF B<>F THEN 3710
3700 GOTO 5480
3710 IF C<>I THEN 3740
3720 IF D<>F THEN 3740
3730 GOTO 5480
3740 NEXT I
3750 GOTO 2170
3760 IF Z=98 THEN 3780
3770 IF Z<>106 THEN 3910
3780 FOR I=F+1 TO 32
3790 CALL GCHAR(E,I,P)
3800 GOSUB 6000
3810 CALL HCHAR(E,I,112)
3820 CALL HCHAR(E,I,32)
3830 IF A<>E THEN 3860
3840 IF B<>I THEN 3860
3850 GOTO 5480
3860 IF C<>E THEN 3890
3870 IF D<>I THEN 3890
3880 GOTO 5480
3890 NEXT I
3900 GOTO 2170
3910 IF Z=99 THEN 3930
3920 IF Z<>107 THEN 4060
3930 FOR I=F-1 TO 1 STEP -1
3940 CALL GCHAR(E,I,P)
3950 GOSUB 6000
3960 CALL HCHAR(E,I,112)
3970 CALL HCHAR(E,I,32)
3980 IF A<>E THEN 4010
3990 IF B<>I THEN 4010
4000 GOTO 5480
4010 IF C<>E THEN 4040
4020 IF D<>I THEN 4040
4030 GOTO 5480
4040 NEXT I
4050 GOTO 2170
4060 IF Z=100 THEN 4080
4070 IF Z<>108 THEN 4230
4080 FOR I=E-1 TO 1 STEP -1
4090 F=F+1
4100 IF F=33 THEN 4220
4110 CALL GCHAR(I,F,P)
4120 GOSUB 6000
4130 CALL HCHAR(I,F,112)
4140 CALL HCHAR(I,F,32)
4150 IF A<>I THEN 4180
4160 IF B<>F THEN 4180
4170 GOTO 5480
4180 IF C<>I THEN 4210
4190 IF D<>F THEN 4210
4200 GOTO 5480
4210 NEXT I
4220 GOTO 2170
4230 IF Z=101 THEN 4250
4240 IF Z<>109 THEN 4400
4250 FOR I=E-1 TO 1 STEP -1
4260 F=F-1
4270 IF F=0 THEN 4390
4280 CALL GCHAR(I,F,P)
4290 GOSUB 6000
4300 CALL HCHAR(I,F,112)
4310 CALL HCHAR(I,F,32)
4320 IF A<>I THEN 4350
4330 IF B<>F THEN 4350
4340 GOTO 5480
4350 IF C<>I THEN 4380
4360 IF D<>F THEN 4380
4370 GOTO 5480
4380 NEXT I
4390 GOTO 2170
4400 IF Z=102 THEN 4420
4410 IF Z<>110 THEN 4570
4420 FOR I=E+1 TO 24
4430 F=F+1
4440 IF F=33 THEN 4560
4450 CALL GCHAR(I,F,P)
4460 GOSUB 6000
4470 CALL HCHAR(I,F,112)
4480 CALL HCHAR(I,F,32)
4490 IF A<>I THEN 4520
4500 IF B<>F THEN 4520
4510 GOTO 5480
4520 IF C<>I THEN 4550
4530 IF D<>F THEN 4550
4540 GOTO 5480
4550 NEXT I
4560 GOTO 2170
4570 FOR I=E+1 TO 24
4580 F=F-1
4590 IF F=0 THEN 4710
4600 CALL GCHAR(I,F,P)
4610 GOSUB 6000
4620 CALL HCHAR(I,F,112)
4630 CALL HCHAR(I,F,32)
4640 IF A<>I THEN 4670
4650 IF B<>F THEN 4670
4660 GOTO 5480
4670 IF C<>I THEN 4700
4680 IF D<>F THEN 4700
4690 GOTO 5480
4700 NEXT I
4710 GOTO 2170
```

```
4720 C=C+1
4730 IF C<>25 THEN 4750
4740 C=1
4750 CALL GCHAR(C,D,P)
4760 GOSUB 6090
4770 IF V=1 THEN 5480
4780 CALL HCHAR(C,D,105)
4790 GOTO 2170
4800 C=C-1
4810 IF C<>0 THEN 4830
4820 C=24
4830 CALL GCHAR(C,D,P)
4840 GOSUB 6090
4850 IF V=1 THEN 5480
4860 CALL HCHAR(C,D,104)
4870 GOTO 2170
4880 D=D+1
4890 IF D<>33 THEN 4910
4900 D=1
4910 CALL GCHAR(C,D,P)
4920 GOSUB 6090
4930 IF V=1 THEN 5480
4940 CALL HCHAR(C,D,106)
4950 GOTO 2170
4960 D=D-1
4970 IF D<>0 THEN 4990
4980 D=32
4990 CALL GCHAR(C,D,P)
5000 GOSUB 6090
5010 IF V=1 THEN 5480
5020 CALL HCHAR(C,D,107)
5030 GOTO 2170
5040 C=C-1
5050 D=D+1
5060 IF C<>0 THEN 5080
5070 C=24
5080 IF D<>33 THEN 5100
5090 D=1
5100 CALL GCHAR(C,D,P)
5110 GOSUB 6090
5120 IF V=1 THEN 5480
5130 CALL VCHAR(C,D,108)
5140 GOTO 2170
5150 C=C-1
5160 D=D-1
5170 IF C<>0 THEN 5190
5180 C=24
5190 IF D<>0 THEN 5210
5200 D=32
5210 CALL GCHAR(C,D,P)
5220 GOSUB 6090
5230 IF V=1 THEN 5480
5240 CALL HCHAR(C,D,109)
5250 GOTO 2170
5260 C=C+1
5270 D=D+1
5280 IF C<>25 THEN 5300
5290 C=1
5300 IF D<>33 THEN 5320
5310 D=1
5320 CALL GCHAR(C,D,P)
5330 GOSUB 6090
5340 IF V=1 THEN 5480
5350 CALL HCHAR(C,D,110)
5360 GOTO 2170
5370 C=C+1
5380 D=D-1
5390 IF C<>25 THEN 5410
5400 C=1
5410 IF D<>0 THEN 5430
5420 D=32
5430 CALL GCHAR(C,D,P)
5440 GOSUB 6090
5450 IF V=1 THEN 5480
5460 CALL HCHAR(C,D,111)
5470 GOTO 2170
5480 CALL SOUND(1000,-5,0)
5490 IF G=2 THEN 5530
5500 E=C
5510 F=D
5520 GOTO 5550
5530 E=A
5540 F=B
5550 CALL HCHAR(E,F,136)
5560 CALL COLOR(14,2,2)
5570 IF F+1<33 THEN 5590
5580 A1=1
5590 IF F-1>0 THEN 5610
5600 B1=1
5610 IF E-1<1 THEN 5670
5620 IF B1=1 THEN 5640
5630 CALL HCHAR(E-1,F-1,138)
5640 IF A1=1 THEN 5660
5650 CALL HCHAR(E-1,F+1,137)
5660 CALL HCHAR(E-1,F,139)
5670 IF E+1>24 THEN 5730
5680 IF A1=1 THEN 5700
5690 CALL HCHAR(E+1,F+1,138)
5700 IF B1=1 THEN 5720
5710 CALL HCHAR(E+1,F-1,137)
5720 CALL HCHAR(E+1,F,139)
5730 IF B1=1 THEN 5750
5740 CALL HCHAR(E,F-1,140)
5750 IF A1=1 THEN 5770
5760 CALL HCHAR(E,F+1,140)
5770 FOR I=1 TO 10
5780 CALL COLOR(14,2,2)
5790 CALL COLOR(14,7,2)
5800 NEXT I
5810 IF PT<>9 THEN 5850
5820 PRINT "TIE GAME!"
5830 PT=0
5840 GOTO 5950
5850 IF PTS>0 THEN 5870
5860 PTS=5
5870 IF G=2 THEN 5920
5880 PRINT "BLUE WINS!"
5890 BL=BL+PTS
5900 PTS=0
5910 GOTO 5950
5920 PRINT "YELLOW WINS!"
5930 YL=YL+PTS
5940 PTS=0
5950 PRINT "SCORE: BLUE "&STR$(BL)&", Y
     ELLOW "&STR$(YL)&"."
5960 INPUT "PLAY AGAIN?":BS
5970 IF SEG$(BS,1,1)<>"N" THEN 1750
5980 CALL CLEAR
5990 STOP
6000 FOR X=144 TO 147
6010 IF P<>X THEN 6070
6020 IF Z>103 THEN 6050
6030 BL=BL-1
6040 GOTO 2170
6050 YL=YL-1
6060 GOTO 2170
6070 NEXT X
6080 RETURN
6090 IF P=32 THEN 6150
6100 IF P<144 THEN 6110 ELSE 6130
6110 PT=9
6120 GOTO 5140
6130 PTS=3
6140 V=1
6150 RETURN
6160 FOR X=144 TO 147
6170 IF P<>X THEN 6200
6180 V=1
6190 RETURN
6200 NEXT X
6210 RETURN
6220 PRINT "YELLOW RUNS OUT OF FUEL"
6230 PTS=2
6240 GOTO 5880
6250 PRINT "BLUE RUNS OUT OF FUEL"
6260 PTS=2
6270 GOTO 5920
6280 END
```

TI BASIC

MAZE RACE

Maze Race is a game written in TI BASIC for two players; one controls the red soldier, and one controls the blue soldier. The game starts out with the opposing soldiers lost at the ends of a forest maze. The object is to reach the safe zone across the field without meeting the enemy. The first soldier to cross his boundary into safety (through the entrance) wins the round, and the game continues until one soldier scores ten times. If the soldiers collide, neither one scores.

The maze is drawn randomly by the computer, so if an impossible maze is drawn (an entrance blocked or a soldier surrounded), it may be redrawn by answering the "Change Maze?" option with "Y" for yes.

The red soldier is moved by pressing the arrow keys on the left keyboard. The blue soldier is moved by pressing I for up, J for left, K for right, and M for down. You may wish to use the Video Games 1 Command Cartridge overlay. No diagonal moves are allowed, and a soldier cannot go through a barrier. Once a key is pressed, the soldier moves in that direction until another key is pressed.

The difficulty of the maze may be altered by adjusting the PRINT statements 220-560. The & is a blank space on the maze, and # is a barrier.

EXPLANATION OF THE PROGRAM
Maze Race

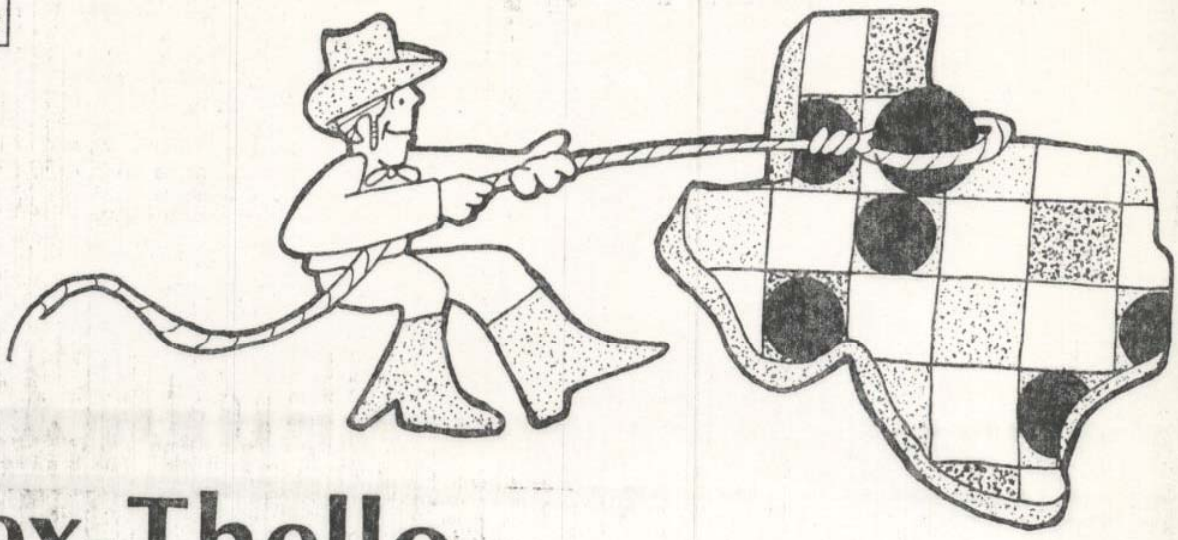
170	Branches to title screen and instructions.
180-210	Subroutine to print messages on screen below maze.
220-570	Subroutines to print maze a line at a time.
580-700	Clears screen and prints maze. Lines of maze are chosen randomly then printed.
710-740	Places soldiers at opposite ends of maze in random horizontal position.
750-810	Prints message, "CHANGE MAZE?", waits for response and branches accordingly.
820-900	Initializes variables. RX, RY, BX, and BY are directional increments. RXC, RYC, BXC, and BYC are coordinates for the red and blue soldiers. RED and BLUE = 1 for a win, 0 for a loss. Sounds a "beep" to start game.
910-1070	Reads red soldier's keyboard entry to move.
1080-1230	Checks where soldier will move and redraws soldier. Checks location for space, block, enemy entrance, or his goal.
1240-1400	Reads blue soldier's keyboard entry to move.
1410-1580	Checks blue soldier's move and location.
1590-1690	Routine if soldiers collide.
1700-1760	Prints message when one soldier wins.
1770-1940	Prints scores.
1950-2000	Asks "TRY AGAIN?" and branches accordingly.
2010-2180	Prints title screen and defines characters and colors; asks if instructions are needed and waits for response.
2190-2270	Prints instructions if desired.

```
100 REM   **************
110 REM   * MAZE RACE  *
120 REM   **************
130 REM
140 REM
150 REM
160 REM
170 GOTO 2010
180 FOR J=1 TO LEN(M$)
190 CALL HCHAR(X,J,ASC(SEG$(M$,J,1)))
200 NEXT J
210 RETURN
220 PRINT "&##&#&#&&&&&&&#&&#&&&&&##&#&"
230 RETURN
240 PRINT "&####&&&#&&&&##&&#&&#&&#&&&&"
250 RETURN
260 PRINT "&&&&&&&#&#&#&&#&&&&#&&&&#&&#"
270 RETURN
280 PRINT "&#&&&#&&&&&&&&&#&&&&&#&&&&&#"
290 RETURN
300 PRINT "&&&#&&##&&&#&&#&&#&&&&#&&#&&"
310 RETURN
320 PRINT "&&##&##&##&&&&&&#&&&&&&&#&&&"
330 RETURN
340 PRINT "#&&#&&##&#&&&&&&&&##&&&###&&"
350 RETURN
360 PRINT "&#&&&#&&&&#&&###&##&&&#&&&&#"
370 RETURN
380 PRINT "&&#&&&&&#&&&&&&&#&&&#&&&&&#&"
390 RETURN
400 PRINT "&&##&&&&##&&&&&&##&&&#&&&&&&"
410 RETURN
420 PRINT "#&#&&&##&&&&##&&&&&&#&&#&&&#"
430 RETURN
440 PRINT "#&&&#&&&&#&&&&#&&&&&&#&&&#&&"
450 RETURN
460 PRINT "&#&&&#&&&&&##&&&&&&&&#&&&&##"
470 RETURN
480 PRINT "&&#&&&&#&&&&&&&#&&&#&&&&#&&&"
490 RETURN
500 PRINT "#&&&#&&&#&#&#&&&&&&&#&&&&##&"
510 RETURN
520 PRINT "&##&&&##&&&&&###&&#&&&#&&&&&"
530 RETURN
540 PRINT "&&&#&&&&&#&&&&&&#&&&&&&#&&&&"
550 RETURN
560 PRINT "&&&&##&&&&##&&&#&&&&##&&&&&&"
570 RETURN
580 CALL CLEAR
590 RANDOMIZE
600 CALL HCHAR(23,3,35,28)
610 FOR I=2 TO 20
620 AA=INT(18*RND)+1
630 ON AA GOSUB 220,240,260,280,300,32
    0,340,360,380,400,420,440,460,480,
    500,520,540,560
640 NEXT I
650 PRINT ::
660 CALL HCHAR(21,3,35,28)
670 CALL VCHAR(1,2,113,21)
680 CALL VCHAR(1,30,105,21)
690 CALL VCHAR(4,2,38,3)
700 CALL VCHAR(16,30,38,3)
710 RXC=INT(18*RND)+2
720 CALL HCHAR(RXC,3,104)
730 BXC=INT(18*RND)+2
740 CALL HCHAR(BXC,29,112)
750 M$="CHANGE MAZE? (Y/N)"
760 X=24
770 GOSUB 180
780 CALL KEY(0,KEY,S)
790 IF KEY=89 THEN 580
800 IF KEY<>78 THEN 780
810 CALL HCHAR(24,1,32,19)
820 RX=0
830 RY=0
840 BX=0
850 BY=0
860 RYC=3
870 BYC=29
880 RED=0
890 BLUE=0
900 CALL SOUND(150,1397,2)
910 CALL KEY(1,K1,S1)
920 IF S1=0 THEN 1080
930 IF K1<>5 THEN 970
940 RX=-1
950 RY=0
960 GOTO 1080
970 IF K1<>3 THEN 1010
980 RY=1
990 RX=0
1000 GOTO 1080
1010 IF K1+1<>1 THEN 1050
1020 RX=1
1030 RY=0
1040 GOTO 1080
1050 IF K1<>2 THEN 1080
1060 RY=-1
1070 RX=0
1080 IF RYC+RY>1 THEN 1120
1090 M$="RED WENT WRONG WAY-BLUE WON."
1100 BLUE=1
1110 GOTO 1440
1120 CALL GCHAR(RXC+RX,RYC+RY,CC)
1130 IF CC=38 THEN 1170
1140 IF CC=112 THEN 1590
1150 CALL SOUND(150,-5,0)
1160 GOTO 1240
1170 CALL HCHAR(RXC,RYC,38)
1180 RXC=RXC+RX
1190 RYC=RYC+RY
1200 CALL HCHAR(RXC,RYC,104)
1210 IF RYC<>30 THEN 1240
1220 RED=1
1230 GOTO 1700
1240 CALL KEY(2,K2,S2)
1250 IF S2=0 THEN 1410
1260 IF K2<>5 THEN 1300
1270 BX=-1
1280 BY=0
1290 GOTO 1410
1300 IF K2<>3 THEN 1340
1310 BY=1
1320 BX=0
1330 GOTO 1410
1340 IF K2+1<>1 THEN 1380
1350 BX=1
1360 BY=0
1370 GOTO 1410
1380 IF K2<>2 THEN 1410
1390 BY=-1
1400 BX=0
1410 IF BYC+BY<31 THEN 1470
1420 M$="BLUE WENT WRONG WAY-RED WON."
1430 RED=1
1440 X=22
1450 GOSUB 180
1460 GOTO 1770
1470 CALL GCHAR(BXC+BX,BYC+BY,DD)
```

```
1480 IF DD=38 THEN 1520
1490 IF DD=104 THEN 1590
1500 CALL SOUND(150,-7,0)
1510 GOTO 910
1520 CALL HCHAR(BXC,BYC,38)
1530 BXC=BXC+BX
1540 BYC=BYC+BY
1550 CALL HCHAR(BXC,BYC,112)
1560 IF BYC<>2 THEN 910
1570 BLUE=1
1580 GOTO 1700
1590 CALL HCHAR(BXC,BYC,38)
1600 CALL HCHAR(RXC,RYC,120,2)
1610 CALL HCHAR(RXC+RX,RYC+RY,120,2)
1620 FOR I=1 TO 15
1630 CALL COLOR(12,7,6)
1640 CALL COLOR(12,6,7)
1650 NEXT I
1660 M$="DUEL!! BOTH SOLDIERS DIE."
1670 X=23
1680 GOSUB 180
1690 GOTO 1930
1700 CALL HCHAR(23,1,32,64)
1710 IF BLUE=1 THEN 1740
1720 M$="RED WINS THIS TIME."
1730 GOTO 1750
1740 M$="BLUE WINS THIS TIME."
1750 X=22
1760 GOSUB 180
1770 M$="TOTAL SCORE:   RED          BLUE "
1780 X=23
1790 GOSUB 180
1800 REDS=REDS+RED
1810 BLUES=BLUES+BLUE
1820 IF REDS<10 THEN 1860
1830 CALL HCHAR(23,19,49)
1840 CALL HCHAR(23,20,48)
1850 GOTO 1910
1860 CALL HCHAR(23,19,REDS+48)
1870 IF BLUES<10 THEN 1910
1880 CALL HCHAR(23,29,49)
1890 CALL HCHAR(23,30,48)
1900 GOTO 2000
```

```
1910 CALL HCHAR(23,29,BLUES+48)
1920 IF REDS=10 THEN 2000
1930 FOR DELAY=1 TO 1000
1940 NEXT DELAY
1950 PRINT "TRY AGAIN? (Y/N)"
1960 CALL KEY(0,KEY,S)
1970 IF S=0 THEN 1960
1980 IF KEY=89 THEN 580
1990 IF KEY<>78 THEN 1960
2000 STOP
2010 CALL CLEAR
2020 PRINT TAB(7);"**********":TAB(7
     );"*        *"
2030 PRINT TAB(7);"* MAZE RACE *":TAB(7
     );"*        *"
2040 PRINT TAB(7);"**********":::::::
     :::::
2050 CALL CHAR(38,"0")
2060 CALL CHAR(35,"FFFFFFFFFFFFFFFF")
2070 CALL COLOR(1,13,1)
2080 CALL CHAR(104,"18187E18182424")
2090 CALL CHAR(105,"FFFFFFFFFFFFFFFF")
2100 CALL COLOR(10,7,1)
2110 CALL CHAR(120,"4922144914224249")
2120 CALL CHAR(112,"18187E18182424")
2130 CALL CHAR(113,"FFFFFFFFFFFFFFFF")
2140 CALL COLOR(11,6,1)
2150 PRINT "WANT INSTRUCTIONS? (Y/N)"
2160 CALL KEY(0,KEY,S)
2170 IF KEY=78 THEN 580
2180 IF KEY<>89 THEN 2160
2190 CALL CLEAR
2200 PRINT "TWO OPPOSING SOLDIERS ARE":
     "LOST IN A JUNGLE MAZE."
2210 PRINT :"EACH MAN MUST FIND HIS WAY
     ":"TO THE OPPOSITE BORDER"
2220 PRINT "TO BE SAFE.":"DO NOT COLLI
     DE OR ELSE"
2230 PRINT "BOTH SOLDIERS WILL DIE."
2240 PRINT :::"PRESS ENTER TO START"
2250 CALL KEY(0,KEY,S)
2260 IF KEY=13 THEN 580 ELSE 2250
2270 END
```

TI BASIC



Tex-Thello

Tex-Thello is a microcomputer version of the popular Othello (a trademark of Gabriel Industries, Inc.) board game. The program written in TI BASIC, pits the human player against the computer for an exciting game on three levels of difficulty: On Level 1, the computer just tries to capture the most markers. On Level 3 (the highest level), the computer takes into account the edge squares and corner squares—thus providing it with more of a theoretical advantage. Level 2 is an intermediate level. The program will check for illegal moves (sounding a warning tone within 30 seconds) and change the color of "captured" markers according to the moves.

Game Rules

1. Since the first four squares in the middle of the board must be occupied (in "checkerboard fashion") first, the program automatically provides this initial setup.

2. The player alternates turns with the computer by entering the grid coordinates for a move. A move consists of placing a color square so that it "captures" (by completing the outflanking of) one or more of the opposite color squares. The computer will then change all the captured squares to the opposite color.

3. A move must always consist of capturing at least one square.

4. If a legal move cannot be made, it then becomes the opponent's turn to move.

5. Capturing may be accomplished horizontally, vertically, or diagonally in one or more rows or directions.

6. The game is over either when the board is filled with color squares, when it is not possible for either opponent to move, or when the board is filled (or partially filled) with all one color. The opponent with the most squares is the winner.

EXPLANATION OF THE PROGRAM

Tex-Thello

Line Nos.	
160	Dimensions arrays for squares captured.
170-240	Stores the name "COMPUTER" for player.
250-400	Option screens; user presses a key for choices.
410-510	Players input names; stored in PLAY(1,10).
520-610	Initializes positions of board.
620-730	Prints labels for game.
740-920	Defines graphics characters and colors.
930-990	Draws starting *Tex-Thello* board; then row number.
1000-1050	Draws starting four positions.
1060-1070	Initializes squares around four center squares; checks for first player on move number 5.
1080-1230	Prints player's name (computer) and black squares indicating whose move.
1240-1330	Player presses column number then row number for move.
1340-1360	Computer prints move.
1370-1480	Checks for legal move.
1490-1550	Sets values of surrounding squares to zero.
1560-1620	Shows move on screen and switches appropriate captured squares; increments TURN (number of moves).
1630-1740	Checks to see if board still contains two colors, otherwise branches to end of game.
1750-1790	Changes player number for next turn and branches to beginning of main loop.
1800-2040	Tallies squares for each player and prints score.
2050-2100	Asks if player wants to play again; branches appropriately or ends program.
2110-2250	Subroutine to check if there is a legal move.
2260-2510	Subroutine to place colored square on board where player or computer wants to move.
2520-2820	Subroutine to check how many squares may be captured.
2830-2940	Subroutine to color captured squares.
2950-4240	Subroutine to calculate computer's move. EXTRA is the number of squares that can be captured; HARD is the level of difficulty (1, 2, 3). For the different levels, the board positions have different values.

```
100 REM  *********************
110 REM  *  TEX-THELLO  *
120 REM  *********************
130 REM
140 REM
150 REM
160 DIM EX(18),EY(18)
170 DATA 67,79,77,80,85,84,69,82
180 RESTORE
190 COMPLAY=0
200 CALL CLEAR
210 FOR I=1 TO 8
220 READ PLAY(1,I)
230 PLAY(2,I)=PLAY(1,I)
240 NEXT I
250 PRINT TAB(8);"TEX-THELLO":::::::
260 PRINT "CHOOSE:::::1 ONE PLAYER V
    S. COMPUTER:::2 TWO PLAYERS:::::
270 CALL KEY(0,K,S)
280 IF (K<49)+(K>50)=-1 THEN 270
290 IF K=50 THEN 410
300 CALL CLEAR
310 PRINT "CHOOSE:::::1 EASY GAME:::
    2 INTERMEDIATE GAME:::3 HARD G
    AME:::::
320 CALL KEY(0,K,S)
330 IF (K<49)+(K>51)=-1 THEN 320
340 HARD=K-48
350 CALL CLEAR
360 PRINT "CHOOSE --- COMPUTER PLAYS:::
    :1 FIRST--RED:::2 SECOND--YELL
    OW:::::
370 CALL KEY(0,K,S)
380 IF (K<49)+(K>50)=-1 THEN 370
390 COMPLAY=K-48
400 IF COMPLAY=1 THEN 470
410 PRINT :::"FIRST PLAYER NAME (RED)"
420 INPUT Z1$
430 FOR I=1 TO 10
440 CALL GCHAR(23,I+4,PLAY(1,I))
450 NEXT I
460 IF COMPLAY=2 THEN 520
470 PRINT :::"SECOND PLAYER NAME (YELLO
    W)"
480 INPUT Z1$
490 FOR I=1 TO 10
500 CALL GCHAR(23,I+4,PLAY(2,I))
510 NEXT I
520 FOR I=1 TO 3
530 DIR(I)=I-2
540 NEXT I
550 FOR I=0 TO 9
560 FOR J=0 TO 9
570 A(I,J)=1
580 IF I*J*(I-9)*(J-9)<>0 THEN 600
590 A(I,J)=A(I,J)+1
600 NEXT J
610 NEXT I
620 CALL CLEAR
630 PRINT TAB(19);"X=    Y="
640 E$="FFFFFFFFFFFFFFFF"
650 CALL CHAR(120,E$)
660 CALL COLOR(12,2,16)
670 FOR I=1 TO 8
680 Y=5+2*I
690 X=Y+4
700 CD=48+I
710 CALL VCHAR(Y,8,CD)
720 CALL VCHAR(4,X,CD)
730 NEXT I
740 A$="FF8080808080808080"
750 B$="FF0101010101010101"
760 C$="8080808080808080FF"
770 D$="0101010101010101FF"
780 CALL CHAR(96,A$)
790 CALL CHAR(97,B$)
800 CALL CHAR(98,C$)
810 CALL CHAR(99,D$)
820 CALL CHAR(104,A$)
830 CALL CHAR(105,B$)
840 CALL CHAR(106,C$)
850 CALL CHAR(107,D$)
860 CALL CHAR(112,A$)
870 CALL CHAR(113,B$)
880 CALL CHAR(114,C$)
890 CALL CHAR(115,D$)
900 CALL COLOR(9,2,16)
910 CALL COLOR(10,2,9)
920 CALL COLOR(11,2,11)
930 TYPE=1
940 FOR X=1 TO 8
950 FOR Y=1 TO 8
960 GOSUB 2270
970 NEXT Y
980 NEXT X
990 FOR X=4 TO 5
1000 FOR Y=4 TO 5
1010 IF X=Y THEN 1050
1020 TYPE=2
1030 A(X,Y)=3
1040 GOTO 1070
1050 TYPE=3
1060 A(X,Y)=4
1070 GOSUB 2330
1080 NEXT Y
1090 NEXT X
1100 FOR I=3 TO 6
1110 FOR J=3 TO 6
1120 IF A(I,J)<>1 THEN 1140
1130 A(I,J)=0
1140 NEXT J
1150 NEXT I
1160 PL=1
1170 TURN=5
1180 REM  BEGIN MAIN LOOP
1190 FOR I=1 TO 10
1200 CALL HCHAR(23,I+7,PLAY(PL,I))
1210 NEXT I
1220 CALL VCHAR(25,23,120)
1230 CALL VCHAR(23,28,120)
1240 IF COMPLAY=PL THEN 1340
1250 CALL KEY(0,RE,ST)
1260 IF ST<>1 THEN 1250
1270 X=RE-48
1280 CALL VCHAR(23,23,RE)
1290 CALL KEY(0,RE,ST)
1300 IF ST<>1 THEN 1290
1310 Y=RE-48
1320 CALL VCHAR(23,28,RE)
1330 GO TO 1370
1340 GOSUB 2960
1350 CALL VCHAR(23,23,48+X)
1360 CALL VCHAR(23,28,48+Y)
1370 IF X>8 THEN 1460
1380 IF X<1 THEN 1460
1390 IF Y>8 THEN 1460
1400 IF Y<1 THEN 1460
1410 IF A(X,Y)>1 THEN 1460
1420 GOSUB 2530
1430 IF EXTRA>0 THEN 1490
1440 GOSUB 2120
1450 IF SW=0 THEN 1640
1460 CALL SOUND(500,200,5)
1470 IF COMPLAY=PL THEN 1640
1480 GOTO 1220
1490 A(X,Y)=PL+2
1500 FOR I=X-1 TO X+1
1510 FOR J=Y-1 TO Y+1
1520 IF A(I,J)<>1 THEN 1540
1530 A(I,J)=0
1540 NEXT J
1550 NEXT I
1560 TURN=TURN+1
1570 TYPE=2
1580 IF PL=1 THEN 1600
1590 TYPE=3
1600 GOSUB 2270
1610 GOSUB 2530
```

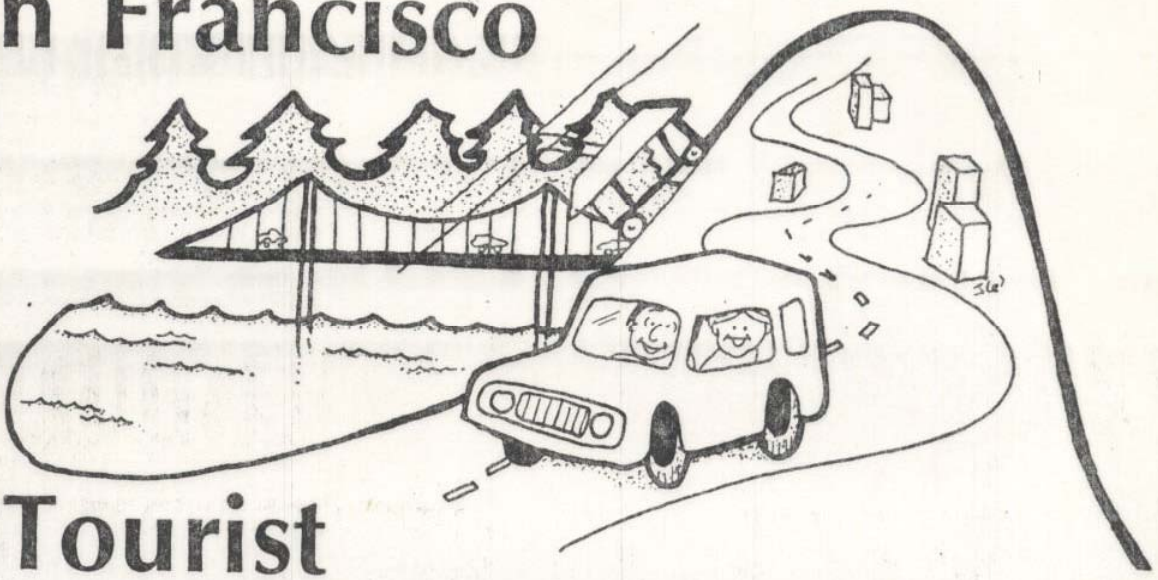
```
1620 GOSUB 2840
1630 IF TURN=65 THEN 1810
1640 IF PL=1 THEN 1670
1650 A1=4
1660 GOTO 1680
1670 A1=3
1680 FOR I=1 TO 8
1690 FOR J=1 TO 8
1700 IF A(I,J)<=1 THEN 1720
1710 IF A(I,J)<>A1 THEN 1750
1720 NEXT J
1730 NEXT I
1740 GOTO 1810
1750 IF PL=1 THEN 1780
1760 PL=1
1770 GOTO 1190
1780 PL=2
1790 GOTO 1190
1800 REM END MAIN LOOP AND BEGIN TOTAL
     S REGION
1810 TOT1=0
1820 TOT2=0
1830 FOR I=1 TO 8
1840 FOR J=1 TO 8
1850 IF A(I,J)=3 THEN 1890
1860 IF A(I,J)<>4 THEN 1900
1870 TOT2=TOT2+1
1880 GO TO 1900
1890 TOT1=TOT1+1
1900 NEXT J
1910 NEXT I
1920 IF TOT2<>TOT1 THEN 1950
1930 PRINT "DRAW 32 TO 32"
1940 GO TO 2050
1950 PRINT
1960 PRINT "AT A SCORE OF";TOT1;" TO ";
     TOT2;":"
1970 PRINT " WINS"
1980 IF TOT1>TOT2 THEN 2010
1990 PWIN=2
2000 GO TO 2020
2010 PWIN=1
2020 FOR I=1 TO 10
2030 CALL HCHAR(23,I+1,PLAY(PWIN,I))
2040 NEXT I
2050 PRINT :"PLAY AGAIN? (Y/N)";
2060 CALL KEY(0,K,S)
2070 IF K=89 THEN 180
2080 IF K<>78 THEN 2060
2090 STOP
2100 REM
2110 REM IS THERE A LEGAL MOVE?
2120 SW=0
2130 ZZX=X
2140 ZZY=Y
2150 FOR X=1 TO 8
2160 FOR Y=1 TO 8
2170 IF A(X,Y)<>0 THEN 2210
2180 GOSUB 2530
2190 IF EXTRA=0 THEN 2210
2200 SW=1
2210 NEXT Y
2220 NEXT X
2230 X=ZZX
2240 Y=ZZY
2250 RETURN
2260 REM ---COLOR ONTO BOARD ROUTINE
2270 IF TYPE<>1 THEN 2330
2280 S1=104
2290 S2=105
2300 S3=106
2310 S4=107
2320 GO TO 2430
2330 IF TYPE<>2 THEN 2390
2340 S1=104
2350 S2=105
2360 S3=106
2370 S4=107
2380 GO TO 2430
2390 S1=112
2400 S2=113
2410 S3=114
2420 S4=115
2430 X1=7+2*X
2440 X2=X1+1
2450 Y1=3+2*Y
2460 Y2=Y1+1
2470 CALL HCHAR(Y1,X1,S1)
2480 CALL HCHAR(Y1,X2,S2)
2490 CALL HCHAR(Y2,X1,S3)
2500 CALL HCHAR(Y2,X2,S4)
2510 RETURN
2520 REM EXTRA SQUARES
2530 EXTRA=1
2540 FOR I=1 TO 3
2550 FOR J=1 TO 3
2560 U=X+DIR(I)
2570 V=Y+DIR(J)
2580 IF U=X THEN 2590 ELSE 2600
2590 IF V=Y THEN 2790
2600 IF PL=1 THEN 2620
2610 IF A(U,V)=3 THEN 2630 ELSE 2790
2620 IF A(U,V)=4 THEN 2630 ELSE 2790
2630 U=U+DIR(I)
2640 V=V+DIR(J)
2650 IF A(U,V)<=1 THEN 2790
2660 IF A(U,V)=2 THEN 2790
2670 IF PL=1 THEN 2690
2680 IF A(U,V)=4 THEN 2700 ELSE 2630
2690 IF A(U,V)=3 THEN 2700 ELSE 2630
2700 U=X+DIR(I)
2710 V=Y+DIR(J)
2720 IF A(U,V)=PL+2 THEN 2790
2730 EX(EXTRA)=U
2740 EY(EXTRA)=V
2750 EXTRA=EXTRA+1
2760 U=U+DIR(I)
2770 V=V+DIR(J)
2780 GO TO 2720
2790 NEXT J
2800 NEXT I
2810 EXTRA=EXTRA-1
2820 RETURN
2830 REM COLOR ADDITION SQUARES
2840 XX=X
2850 YY=Y
2860 FOR K=1 TO EXTRA
2870 X=EX(K)
2880 Y=EY(K)
2890 GOSUB 2270
2900 A(X,Y)=PL+2
2910 NEXT K
2920 X=XX
2930 Y=YY
2940 RETURN
2950 REM GET COMPUTER MOVE
2960 EXTRAP=-100
2970 FOR X=1 TO 8
2980 FOR Y=1 TO 8
2990 IF A(X,Y)<>0 THEN 4110
3000 GOSUB 2530
3010 IF EXTRA=0 THEN 4110
3020 IF TURN<57 THEN 3040
3030 HARD=1
3040 IF HARD=1 THEN 4070
3050 IF X=1 THEN 3300
3060 IF X=8 THEN 3380
3070 IF Y=1 THEN 3260
3080 IF Y=8 THEN 3260
3090 IF HARD=2 THEN 4070
3100 IF X=2 THEN 3180
3110 IF X=7 THEN 3180
3120 IF Y=2 THEN 4070
3130 IF Y=7 THEN 4070
3140 IF X*X+Y*Y+36<>9*(X+Y)THEN 3160
3150 EXTRA=EXTRA+1.1
3160 EXTRA=EXTRA+4
3170 GO TO 4070
```

```
3180 IF Y=2 THEN 3210
3190 IF Y=7 THEN 3210
3200 GO TO 4070
3210 U=(7*X-9)/5
3220 V=(7*Y-9)/5
3230 IF A(U,V)>1 THEN 4070
3240 EXTRA=EXTRA-12
3250 GO TO 4070
3260 IF HARD=2 THEN 3360
3270 FOR I=1 TO 8
3280 EDGE(I)=A(I,Y)
3290 NEXT I
3300 FOR I=1 TO EXTRA
3310 IF EY(I)<>Y THEN 3330
3320 EDGE(EX(I))=PL+2
3330 NEXT I
3340 Z=X
3350 GO TO 3570
3360 EXTRA=EXTRA+3
3370 GO TO 4070
3380 IF Y=1 THEN 3520
3390 IF Y=8 THEN 3520
3400 IF HARD=2 THEN 3500
3410 FOR I=1 TO 8
3420 EDGE(I)=A(X,I)
3430 NEXT I
3440 FOR I=1 TO EXTRA
3450 IF EX(I)<>X THEN 3470
3460 EDGE(EY(I))=PL+2
3470 NEXT I
3480 Z=Y
3490 GO TO 3570
3500 EXTRA=EXTRA+3
3510 GO TO 4070
3520 IF HARD=2 THEN 3550
3530 EXTRA=EXTRA+14
3540 GO TO 4070
3550 EXTRA=EXTRA+6
3560 GO TO 4070
3570 CORN=0
3580 FOR I=1 TO EXTRA.
3590 IF 2*(EX(I)-2)*(EX(I)-7)+(EY(I)-2)
     *(EY(I)-7)=0 THEN 3610
3600 GO TO 3660
3610 U=(7*EX(I)-9)/5
3620 V=(7*EY(I)-9)/5
3630 IF A(U,V)<=1 THEN 3650
3640 GO TO 3660
3650 CORN=1
3660 NEXT I
3670 TYPE=0
3680 IF EDGE(1)=PL+2 THEN 3730
3690 LT=EDGE(1)
3700 IF LT<>0 THEN 3740
3710 LT=1
```

```
3720 GO TO 3740
3730 LT=2
3740 FOR I=2 TO Z-1
3750 IF EDGE(I)=PL+2 THEN 3800
3760 IF EDGE(I)<=1 THEN 3790
3770 LT=EDGE(I)
3780 GO TO 3800
3790 LT=1
3800 NEXT I
3810 IF LT=2 THEN 3970
3820 IF EDGE(8)=PL+2 THEN 3870
3830 HT=EDGE(8)
3840 IF HT<>0 THEN 3880
3850 HT=1
3860 GO TO 3880
3870 HT=2
3880 FOR I=7 TO Z+1 STEP -1
3890 IF EDGE(I)=PL+2 THEN 3940
3900 IF EDGE(I)<=1 THEN 3930
3910 HT=EDGE(I)
3920 GO TO 3940
3930 HT=1
3940 NEXT I
3950 IF HT=2 THEN 3970
3960 IF HT<>LT THEN 4030
3970 TYPE=1
3980 IF CORN=0 THEN 4010
3990 EXTRA=EXTRA-8
4000 GO TO 4070
4010 EXTRA=EXTRA+8
4020 GO TO 4070
4030 IF CORN=0 THEN 4060
4040 EXTRA=EXTRA-12
4050 GO TO 4070
4060 EXTRA=EXTRA-4
4070 IF EXTRA>EXTRAP THEN 4080 ELSE 411
     0
4080 EXTRAP=EXTRA
4090 ZX=X
4100 ZY=Y
4110 NEXT Y
4120 NEXT X
4130 IF EXTRAP=-100 THEN 4140 ELSE 4210
4140 FOR X=1 TO 8
4150 FOR Y=1 TO 8
4160 IF A(X,Y)>1 THEN 4190
4170 ZX=X
4180 ZY=Y
4190 NEXT Y
4200 NEXT X
4210 X=ZX
4220 Y=ZY
4230 RETURN
4240 END
```

San Francisco Tourist

TI BASIC



"I left my heart in San Francisco . . ." Designed to highlight the sights that abound in and around the City by the Bay, this TI BASIC program is actually two games in one.

First, try your skill at driving down Lombard Street between Hyde and Leavenworth. It's on a steep hill and is known as the "crookedest street in the world." Use the left and right arrow keys (S and D) to steer down the red brick road without bumping into the white concrete sides—or onto someone's green lawn.

Now drive north across the Golden Gate Bridge to Muir Woods, a beautiful, peaceful forest with some of the world's tallest living trees. Start at the upper left corner of the screen and take a quiet walking tour through the woods. Use the arrow keys to change direction, then press ENTER to mark the trees you've seen on your map.

Programming Techniques

This game program implements many of the features discussed in the article *Fun and Games*. The title screen presents the choice of games, and the player need only press the key of his choice (wrong keys are ignored). The program will then branch to the appropriate game, and a screen of instructions is printed. The screen stays on the instructions only as long as the player wishes. The player can just press any key when he is ready to start the game.

Crookedest Street uses scrolling during printing to simulate the road going past. A DEFinition statement near the beginning of the program on line 170 defines a random coordinate R between −3 and +3. A line of road is printed offset R from the previous line. Lines 820 to 850 make sure the road line stays on the screen.

Both games move an object (represented by one graphics character for simplicity and speed) by using the arrow keys. In *Crookedest Street* only the left and right arrow keys are used. The car is always drawn on Row 7, and the arrow keys determine whether the car is drawn in the same column, two columns to the left, or two columns to the right. Lines 930-980 keep the car on the screen. In *Muir Woods* the person may move up, down, left, or right, but will not wrap—staying at the edge, instead. The person will also continue to move in one direction until another arrow key is pressed; the character is moved in each CALL KEY loop.

CALL GCHAR(X,Y,G) is used in both games. In *Crookedest Street* you need to know if the new position of the car is a red square (okay), a white square (crash), or a green square (fatal crash). After the new car position is drawn, the old position must be replaced by the appropriately colored square.

Muir Woods uses GCHAR to determine positions of trees for marking. Also, the person leaves a trail. So if the square was a blank, the trail is printed; but if it was a tree or a marked tree, that character stays there.

Muir Woods also demonstrates the use of a timer or counter in the CALL KEY loop. You may change the value 100 for SH in line 1910 for more or less time.

I wanted to use [ENTER] as the key to press for "firing," so the split keyboard method of detecting the "fire" key was not possible. If you use the split keyboard you can alternate calling the halves of the keyboard and detect the "fire" key sooner, but since the codes are different for the 99/4 and the 99/4A consoles, the game instructions would have to be different. [ENTER] is not detected on the 99/4A; you must press the period to return key code 13. In these games the quickest way to detect [ENTER] is to let go of the arrow keys before pressing [ENTER].

EXPLANATION OF THE PROGRAM

San Francisco Tourist

Line Nos.	
150-170	Defines functions to be used as random coordinates.
180-240	Clears screen; defines graphics characters for bridge.
250-380	Prints bridge and title; if the program is just starting, plays "I Left My Heart in San Francisco."
390	Prints choices of games.
400-420	Defines graphics characters for games.

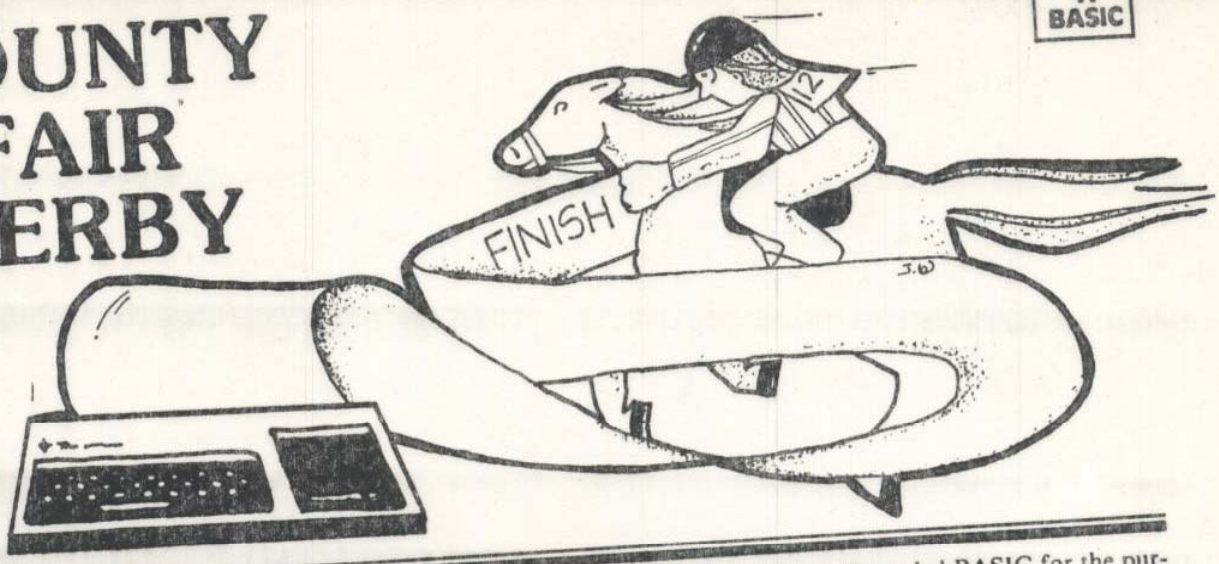
430-460	Receives player's input and branches appropriately.
470-500	Subroutine to press any key to start.
510-530	Subroutine to delay.
540-610	Prints instructions for *Crookedest Street*; defines graphics characters; waits for player to press a key.
620-750	Clears screen; defines graphics colors; prints game screen. DATA contains coordinates for printing road.
760-780	Initializes coordinates of road and car.
790-860	Prints 75 lines of crooked street randomly; last 15 lines are straight.
870-980	Makes sound, draws new position of car depending on key pressed; replaces old position with proper graphics character.
990-1070	Tests for crash; makes a sound and increments number of crashes.
1080-1170	Ending remarks; plays victory melody for zero crashes.
1180-1210	Procedure if car goes into green.
1220-1250	Delays, then waits for player to press a key.
1260-1290	Clears screen; returns colors to black; branches to menu screen.
1300-1400	Prints instructions for *Muir Woods* and defines graphics characters and colors.
1410-1470	Clears screen, randomly draws 70 trees on screen.
1480-1520	Initializes time, marked trees, coordinates, graphics.
1530-1540	Places person at entrance and sounds initial beep.
1550-1860	Moves person depending on key pressed. Person will not "wrap" but stays at edge.
1870-1920	Increments time and prints time; if time = 100, ends game.
1930-2010	Procedure for marking tree.
2020-2100	Ending statements; returns colors to black; branches to menu screen.
2110-2120	Ends program.

```
100 REM  *********
110 REM  *  SF  TOURIST  *
120 REM  *********
130 REM
140 REM
150 DEF  R19=INT(19*RND)+1
160 DEF  R28=INT(26*RND)+2
170 DEF  R=(-1)^(INT(4*RND))*(INT(4*RND
    ))
180 CALL  CLEAR
190 CALL  CHAR(64,"18183C3C5A5A9999")
200 CALL  CHAR(35,"01010204083C7FFF")
210 CALL  CHAR(47,"1818181818187FFF")
220 CALL  CHAR(36,"8066180667FFF")
230 CALL  CHAR(37,"0106186067FFF")
240 CALL  CHAR(38,"80804026100CFFFF")
250 CALL  SCREEN(4)
260 PRINT  TAB(12);"@  @";:TAB(11);"#/$%
    /&"
270 PRINT  :::"**  SAN  FRANCISCO  TOURIST
    **":::::
280 IF  I<>0  THEN  390
290 P=300
300 CALL  SOUND(P,330,0)
310 CALL  SOUND(P,349,0)
320 CALL  SOUND(P,440,0)
330 CALL  SOUND(4*P,392,0,131,10,165,8)
340 CALL  SOUND(P,440,0)
350 CALL  SOUND(P,494,0,165,10,196,8)
360 CALL  SOUND(P,523,0)
370 CALL  SOUND(P,440,0,147,10)
380 CALL  SOUND(4*P,294,0,220,8,175,10)
390 PRINT  "WHICH  DO  YOU  WISH  TO  VISIT?
    ":::"  1  CROOKEDEST  STREET":::"  2
    MUIR  WOODS":::::"  3  END  PROGRAM":
    :::
400 CALL  CHAR(33,"FFFFFFFFFFFFFFFF")
410 CALL  CHAR(41,"FFFFFFFFFFFFFFFF")
420 CALL  CHAR(40,"0")
430 CALL  KEY(0,K,S)
440 IF  (K<49)+(K>51)=-1  THEN  430
450 CALL  CLEAR
460 ON  K-48  GOTO  540,1300,2110
470 PRINT  "PRESS  ANY  KEY  TO  START.";
480 CALL  KEY(0,K,S)
490 IF  S<1  THEN  480
500 RETURN
510 FOR  I=1  TO  500
520 NEXT  I
530 RETURN
540 PRINT  "  **  CROOKEDEST  STREET  **"
550 CALL  CHAR(96,"3C7C7C3838387C7C38")
560 CALL  CHAR(97,"C636336181818366C3")
570 PRINT  :::"LOMBARD  STREET  IS  THE":::"
    CROOKEDEST  STREET  IN  THE":::"WORLD.
    IT  IS  ONE  BLOCK"
580 PRINT  :"LONG  ON  A  STEEP  HILL.":::"Y
    OUR  CHALLENGE  IS  TO  DRIVE":::"DOWN
    THE  RED  BRICK  ROAD"
590 PRINT  :"WITHOUT  BUMPING  THE  CONCRE
    TE":::"SIDES.  USE  THE  ARROW  KEYS":
    :"TO  STEER.":::
600 C=0
610 GOSUB  470
620 DATA  6,6,7,8,9,11,13,15,18,20,20,1
    7,15,13,11,9,6,6,8,11,13,16,19,20
630 CALL  CLEAR
640 CALL  SCREEN(3)
650 CALL  COLOR(2,7,16)
660 CALL  COLOR(9,2,11)
670 CALL  COLOR(1,12,3)
680 RESTORE  620
690 FOR  I=1  TO  24
700 READ  J
710 CALL  HCHAR(I,J,40,8)
720 CALL  HCHAR(I,J+1,41,6)
730 NEXT  I
740 CALL  HCHAR(7,17,96)
750 RANDOMIZE
760 J=18
770 X=17
780 G=41
790 FOR  I=1  TO  75
800 IF  I>59  THEN  860
810 J=J+R
820 IF  J<21  THEN  840
830 J=21
840 IF  J>1  THEN  860
850 J=1
860 PRINT  TAB(J);"(((((((((("
870 CALL  SOUND(-100,110,1,-1,1)
880 CALL  HCHAR(6,X,G)
890 CALL  KEY(0,K,S)
900 IF  (K<>83)+(K<>68)=-2  THEN  990
910 IF  K=83  THEN  960
920 X=X+2
930 IF  X<31  THEN  990
940 X=31
950 GOTO  990
960 X=X-2
970 IF  X>2  THEN  990
980 X=2
990 CALL  GCHAR(7,X,G)
1000 IF  G=41  THEN  1060
1010 IF  G=96  THEN  1060
1020 IF  G=32  THEN  1180
1030 CALL  SOUND(-50,-5,0)
1040 CALL  HCHAR(7,X,97)
1050 C=C+1
1060 CALL  HCHAR(7,X,96)
1070 NEXT  I
1080 CALL  HCHAR(22,1,32,64)
```

```
1090 PRINT "YOU MADE IT";"::::"NUMBER OF C
     RASHES:";C
1100 IF C>0 THEN 1220
1110 DATA 330,392,523,659,523,659,659
1120 RESTORE 1110
1130 FOR I=1 TO 7
1140 READ S
1150 CALL SOUND(150,S,0)
1160 NEXT I
1170 GOTO 1220
1180 CALL SOUND(200,-5,0,400,0)
1190 CALL HCHAR(7,X,97,2)
1200 CALL HCHAR(22,1,32,64)
1210 PRINT "SORRY; THE CAR IS DAMAGED":
     :"BEYOND REPAIR"
1220 GOSUB 510
1230 PRINT :::"PRESS ANY KEY"
1240 CALL KEY(0,K,S)
1250 IF S<1 THEN 1240
1260 CALL CLEAR
1270 CALL COLOR(2,2,1)
1280 CALL COLOR(1,2,1)
1290 GOTO 250
1300 PRINT TAB(6);"** MUIR WOODS **"
1310 CALL CHAR(96,"1C1C087F0814222")
1320 CALL CHAR(97,"10107C101")
1330 CALL COLOR(9,7,1)
1340 CALL CHAR(104,"1039107C10FE101")
1350 CALL CHAR(105,"FF81A59999A581FF")
1360 CALL COLOR(10,13,1)
1370 PRINT :::"MUIR WOODS IS A BEAUTIFUL
     "::::"FOREST OF GIANT TREES"
1380 PRINT :"NORTH OF SAN FRANCISCO.":::
     "TAKE A TOUR AND MARK AS":::"MANY T
     REES ON YOUR MAP AS"
1390 PRINT :"YOU CAN. MOVE BY PRESSING
     "::::"THE ARROW KEYS; MARK TREES":::"
     BY PRESSING <ENTER>.":::::
1400 GOSUB 470
1410 CALL CLEAR
1420 CALL SCREEN(12)
1430 PRINT "TIME":::"TREES"
1440 RANDOMIZE
1450 FOR I=1 TO 70
1460 CALL HCHAR(R19,R28,104)
1470 NEXT I
1480 SH=0
1490 P=0
1500 X=2
1510 Y=2
1520 G=32
1530 CALL HCHAR(2,2,96)
1540 CALL SOUND(150,1397,4)
1550 CALL KEY(0,K,S)
1560 IF K=13 THEN 1930
1570 IF K<>69 THEN 1610
1580 DX=-1
1590 DY=0
1600 GOTO 1720
1610 IF K<>68 THEN 1650
1620 DX=0
1630 DY=1
1640 GOTO 1720
1650 IF K<>88 THEN 1690
1660 DX=1
1670 DY=0
1680 GOTO 1720
1690 IF K<>83 THEN 1720
1700 DX=0
1710 DY=-1
1720 IF G>=104 THEN 1740
1730 G=97
1740 CALL HCHAR(X,Y,G)
1750 X=X+DX
1760 IF X>1 THEN 1780
1770 X=1
1780 IF X<20 THEN 1800
1790 X=20
1800 Y=Y+DY
1810 IF Y>2 THEN 1830
1820 Y=2
1830 IF Y<31 THEN 1850
1840 Y=31
1850 CALL GCHAR(X,Y,G)
1860 CALL HCHAR(X,Y,96)
1870 SH=SH+1
1880 FOR II=1 TO LEN(STR$(SH))
1890 CALL HCHAR(21,II+9,ASC(SEG$(STR$(S
     H),II,1)))
1900 NEXT II
1910 IF SH=100 THEN 2020
1920 GOTO 1550
1930 CALL SOUND(100,-2,0)
1940 IF G<>104 THEN 1870
1950 CALL HCHAR(X,Y,105)
1960 G=105
1970 P=P+1
1980 FOR II=1 TO LEN(STR$(P))
1990 CALL HCHAR(23,II+9,ASC(SEG$(STR$(P
     ),II,1)))
2000 NEXT II
2010 GOTO 1870
2020 PRINT :"TIME IS UP.":::"YOU LOOKED
     AT";P;"TREES."
2030 GOSUB 510
2040 PRINT :::"PRESS ANY KEY"
2050 CALL KEY(0,K,S)
2060 IF S<1 THEN 2050
2070 CALL CLEAR
2080 CALL COLOR(9,2,1)
2090 CALL COLOR(10,2,1)
2100 GOTO 250
2110 CALL CHAR(33,"0101010101010001")
2120 END
```

TI BASIC

COUNTY FAIR DERBY



County Fair Derby is a party game in which up to eight players bet on horses in a color-animated race. Our family finds it quite exciting—especially with three or more players. There is, however, only *one* keyboard operator; the rest is up to the computer. In addition to running the five horses, the computer keeps tabs on each horse's track record, plus the bankroll of each player. The program operation is simple and self-prompting. To break the input loop, the word LAST must be entered. If this word is misspelled, it then becomes just another player's name. When this TI BASIC program was loaded into Extended BASIC for the purpose of checking available memory left, the SIZE command revealed that there were 4873 bytes left. This leaves enough memory for you to add to, or use to modify the program. You might try giving the computer a fixed amount of money before the races start and having the players try to "break the track." Other bells and whistles I leave to your imagination. Here's hoping you enjoy the program as much as I enjoyed writing it. But don't waste another minute. It's already post time—the horses will soon be off and running . . . .

EXPLANATION OF THE PROGRAM
County Fair Derby

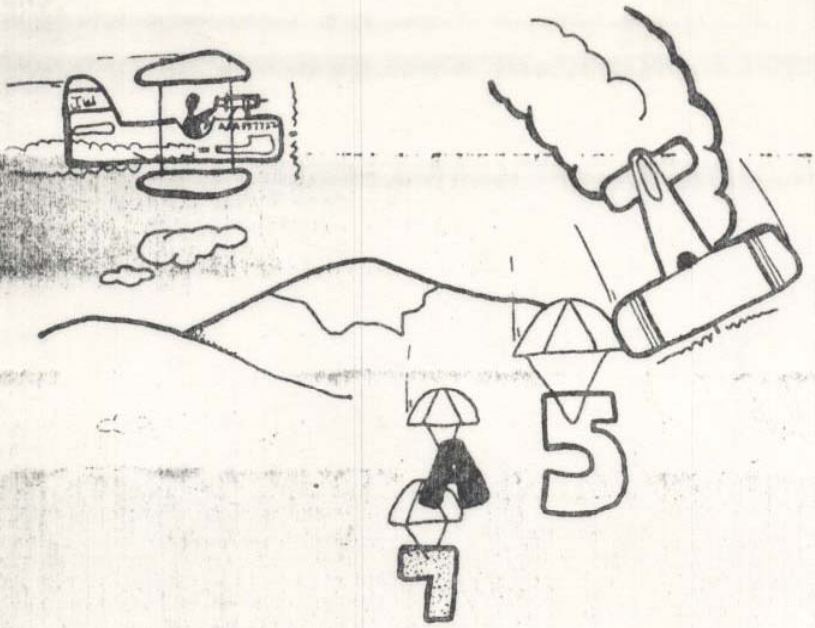
Line Nos.	
140-340	Introduction display and odds table.
350-420	Introductory music and wait for key.
430-1060	Initialization and define characters to be used for display.
1070-1560	Input routines: players' names, choices for horse selection, kind of bet and amount. Typing LAST for player's name breaks the INPUT loop.
1570-1810	Draws track with lane numbers and plays post-time tune.
1820	Z is a switch to control RETURN from subroutine at 2490.
1830-2020	Positions horses on the track in the proper place and color (subroutine at 2490 draws horse and RETURNs if Z equals 1).
2030-2190	Rests Z; sets starting coordinates for horses (K and S are variables used later in determing win, place and show.) Waits for "S" key to start.
2200-2460	Generates random number from one to five to determine which horse to move. Line number 2220 (ON N GOTO) finds position of horse, sets coordinates for move routine and jumps to move routine. (If the vertical coordinate has been set to zero, the horse has finished and the program jumps back for a new random number.)
2470-2600	Moves horse through an animation loop and redraws it two positions forward from where it started. ("Q" is used as a control switch to pass through the loop twice.)
2610-2830	Checks if V>28 (end of race for that horse); if not, sets new coordinate values and jumps back for new random number.
2840-3120	Calculates the finishing horse (D). If S equals 0, the horse wins. Set S equals winning number. Line 2870 (ON S GOSUB) sets color for winning announcement. Line 2990 (ON S GOTO) sets column to zero to remove horse from race; jumps back for a new random number and continues. If S<>0, then the finishing horse becomes K for second place and, (except for setting color) a similar routine is followed. If K<>0, then D becomes the third place horse and the race stops.
3130-3510	Displays win, place, show announcement and waits for key.
3520-3870	On K1(X) goes to the kind of bet player (X) made. Checks to see if player (X) has won and calculates the amount. If there are winnings, goes to subroutine 4090. For no winnings, GOSUB 3970. On return goes to 3880. Increments (X). Checks to see if four results have been displayed; if so, goes to 4130 and waits for key before returning for next results (3550); if not over four, goes directly to 3550.
3880-3960	
3970-4010	Subroutine to update and display losers.
4020-4080	Subroutine to update and display losers in debt.
4090-4120	Subroutine to update and display winners.
4130-4160	Wait for key and check for LAST before continuing.
4170-4290	Update past records and display for players betting on trends. Wait for key.
4300-4340	Loop back for INPUTS of next race.
4350-4380	Data for music. Use "break" key to end program.

```
100 REM  *********************
110 REM  *  COUNTY FAIR DERBY  *
120 REM  *********************
130 REM
135 REM
140 CALL CLEAR
150 CALL COLOR(2,2,14)
160 FOR I=3 TO 8
170 CALL COLOR(I,2,12)
180 NEXT I
190 CALL HCHAR(24,2,42,29)
200 PRINT
210 PRINT TAB(8);"COUNTY FAIR DERBY"::::
220 PRINT TAB(8);"A FIVE HORSE RACE"::::
230 PRINT TAB(4);"YOU CAN BET FOUR WAY
    S"::::
240 PRINT "<1> WIN      PAYS 4 TO 1"::::
250 PRINT "<2> PLACE    PAYS 3 TO 2"::::
260 PRINT "<3> SHOW     PAYS 2 TO 3"::::
270 PRINT "<4> PARLAY   PAYS 15 TO 1"::
    :
280 CALL HCHAR(24,9,42,14)
290 PRINT ::::
300 PRINT "PARLAY<PICK 1ST; AND 2ND;>
    "::::
310 PRINT "EACH PLAYER IS GIVEN $200"::
    :
320 CALL HCHAR(24,2,42,29)
330 CALL VCHAR(1,2,42,24)
340 CALL VCHAR(1,30,42,24)
350 RESTORE 4370
360 READ DU,NO
370 IF DU=0 THEN 400
380 CALL SOUND(300*DU,NO,5)
390 GOTO 360
400 PRINT "PRESS ANY KEY"
410 CALL KEY(0,KEY,STAT)
420 IF STAT=0 THEN 410
430 CALL CLEAR
440 PRINT TAB(7);"***HANG ON***"::::::
450 PRINT TAB(7);"GOTTA GET THE"::::
460 PRINT TAB(11);"HORSES"::::::::
470 DIM H$(50)
480 H$(1)="0000000004020100FF"
490 H$(2)="0000002080F1F30F0F"
500 H$(3)="0F0F0F1020400000FF"
510 H$(4)="F0F0F0804020000000"
520 H$(5)="000000000000007F7F"
530 H$(6)="000000000601E3EFF0"
540 H$(7)="0F0F0F8040200000FF"
550 H$(8)="F0F0F0102040000000"
560 H$(9)="0000000010307010FF"
570 H$(10)="00000F1F3000000007"
580 H$(11)="00001F3F3100000003"
590 H$(12)="0006000000010306FC"
600 H$(13)="0000007070606070F7"
610 H$(14)="0000080808080808F8"
620 H$(15)="0000C0E0606060E0F0"
630 H$(16)="0000C0E0F0707E0E0F"
640 H$(17)="0000606E0E0606060F"
650 H$(18)="0000F0F00000C0F0FF"
660 H$(19)="0101010107070000FF"
670 H$(20)="1F1830303F3F0000FF"
680 H$(21)="0300003131F1F0000F"
690 H$(22)="0F0F0000000000000F"
700 H$(23)="0000006070301000FF"
710 H$(24)="8080808080E0E00000"
720 H$(25)="8000000000E0E00000"
730 H$(26)="E0706070F0E0C00000"
740 H$(27)="F1F1606060600000FF"
750 H$(28)="7111111110E0000000"
760 H$(29)="0000303030303131FF"
770 H$(30)="000000C0C0C0C8C8CF"
780 H$(31)="00003C3C1818181818"
790 H$(32)="0000303030C3E37333"
800 H$(33)="000000C0C0C0C0C8CF"
810 H$(34)="0000F3F3030303F3FF"
820 H$(35)="0000FCFC0C0C0000FF"
830 H$(36)="1F1F1F1E1C1C0000FF"
840 H$(37)="F8F8F8F8383830000F"
850 H$(38)="1818181833C3C0000F"
860 H$(39)="3130303030300000FF"
870 H$(40)="CCEC7C3C1C0C0000FF"
880 H$(41)="3F0000303F3F0000FF"
890 H$(42)="FC0C0C0CFCFC0000FF"
900 D=120
910 K=1
920 FOR D=D TO D+7
930 CALL CHAR(D,H$(K))
940 K=K+1
950 NEXT D
960 IF D>152 THEN 980
970 GOTO 910
980 CALL CLEAR
990 CALL COLOR(11,15,6)
1000 CALL COLOR(12,14,11)
1010 CALL COLOR(13,13,11)
1020 CALL COLOR(14,2,11)
1030 CALL COLOR(15,7,11)
1040 CALL COLOR(16,5,11)
1050 CALL COLOR(2,2,12)
1060 CALL CLEAR
1070 X=1
1080 CALL CLEAR
1090 PRINT "TYPE PLAYER'S NAME ?"::::
1100 PRINT "AFTER THE LAST PLAYER'S NAM
     E"::::
1110 PRINT "HAS BEEN ENTERED TYPE LAST"
     :::::
1120 INPUT "NAME ?":NAME$(X)
1130 IF NAME$(X)="LAST" THEN 1570
1140 IF X>=9 THEN 1160
1150 GOTO 1190
1160 PRINT "EIGHT IS THE MAX.NUM.OF PLA
     YERS"
1170 PRINT "TYPE LAST TO CONTINUE"
1180 GOTO 1120
1190 TOT(X)=200
1200 CALL CLEAR
1210 GOSUB 1230
1220 GOTO 1080
1230 PRINT ::"O.K. ";NAME$(X);" PICK A H
     ORSE ?"::::
1240 INPUT "HORSE ? ":HO(X)
1250 IF HO(X)>5 THEN 1270
1260 GOTO 1310
1270 GOSUB 1290
1280 GOTO 1240
1290 PRINT ::"NUM. TOO BIG TRY AGAIN"::
1300 RETURN
1310 PRINT ::"WHAT KIND OF BET ?<1 TO 4>
     "::::
1320 PRINT "<1>= WIN"::
1330 PRINT "<2>= PLACE"::
1340 PRINT "<3>= SHOW"::
1350 PRINT "<4>= PARLAY"::
1360 INPUT "KIND.? ":KI(X)
1370 IF KI(X)>4 THEN 1400
1380 IF KI(X)=4 THEN 1500
1390 GOTO 1420
1400 GOSUB 1290
1410 GO TO 1360
1420 PRINT ::"HOW MUCH DO YOU BET ?<$1 T
     O $200>"::::
1430 INPUT "BET ? ":BET(X)
1440 IF BET(X)>200 THEN 1460
1450 GOTO 1480
1460 GOSUB 1290
1470 GOTO 1430
1480 X=X+1
1490 RETURN
1500 PRINT ::"YOU PICKED NO.";HO(X);"  T
     O WIN"::::
1510 PRINT "WHICH HORSE TO PLACE ?"::::
1520 INPUT "PLACE ?":PA2(X)
1530 IF PA2(X)>5 THEN 1550
1540 GOTO 1420
1550 GOSUB 1290
```

```
1560 GOTO 1520
1570 CALL CLEAR
1580 PRINT "PRESS S TO START"
1590 CALL COLOR(2,11,11)
1600 FOR X=1 TO 22
1610 PRINT
1620 NEXT X
1630 CALL CHAR(119,"81C3661818663C381")
1640 CALL HCHAR(9,1,119,30)
1650 CALL HCHAR(20,1,119,30)
1660 X=10
1670 Y=2
1680 FOR A=1 TO 10
1690 CALL HCHAR(X,Y,42,29)
1700 X=X+1
1710 NEXT A
1720 RESTORE 4350
1730 READ DU,NO                    IF DU=0 THEN 1720
1750 CALL SOUND(200*DU,NO,5)
1760 GOTO 1730
1770 CALL HCHAR(10,2,49)
1780 CALL HCHAR(12,2,50)
1790 CALL HCHAR(14,2,51)
1800 CALL HCHAR(16,2,52)
1810 CALL HCHAR(18,2,53)
1820 Z=1
1830 D=120
1840 R=10
1850 V=3
1860 GOSUB 2490
1870 D=128
1880 R=12
1890 V=3
1900 GOSUB 2490
1910 D=136
1920 R=14
1930 V=3
1940 GOSUB 2490
1950 D=144
1960 R=16
1970 V=3
1980 GOSUB 2490
1990 D=152
2000 R=18
2010 V=3
2020 GOSUB 2490
2030 Z=0
2040 A=10
2050 B=4
2060 I=16
2070 J=4
2080 E=12
2090 F=4
2100 O=18
2110 P=4
2120 G=14
2130 H=4
2140 K=0
2150 S=0
2160 CALL KEY(0,KEY,STATUS)
2170 IF STATUS=0 THEN 2160
2180 IF KEY=83 THEN 2200
2190 GOTO 2160
2200 RANDOMIZE
2210 N=INT(5*RND)+1
2220 ON N GOTO 2230,2280,2330,2380,2430
2230 R=A
2240 V=B
2250 IF B=0 THEN 2200
2260 D=120
2270 GOTO 2470
2280 R=E
2290 V=F
2300 IF F=0 THEN 2200
2310 D=128
2320 GOTO 2470
2330 R=G
2340 V=H
2350 IF H=0 THEN 2200
2360 D=136
2370 GOTO 2470
2380 R=I
2390 V=J
2400 IF J=0 THEN 2200
2410 D=144
2420 GOTO 2470
2430 R=O
2440 V=P
2450 D=152
2460 IF P=0 THEN 2200
2470 CALL HCHAR(R,V-1,42)
2480 CALL HCHAR(R+1,V-1,42)
2490 CALL HCHAR(R,V,D)
2500 CALL HCHAR(R,V+1,D+1)
2510 CALL HCHAR(R+1,V,D+2)
2520 CALL HCHAR(R+1,V+1,D+3)       5,200,2
2530 CALL SOUND(-5,700,2)
2540 IF Z=0 THEN 2560
2550 RETURN
2560 IF Q=1 THEN 2610
2570 Q=1
2580 V=V+1
2590 D=D+4
2600 GOTO 2470
2610 D=D-4
2620 Q=0
2630 IF V>28 THEN 2840
2640 V=V+1
2650 IF D=120 THEN 2720
2660 IF D=128 THEN 2750
2670 IF D=144 THEN 2780
2680 IF D=152 THEN 2810
2690 G=R
2700 H=V
2710 GOTO 2200
2720 A=R
2730 B=V
2740 GOTO 2200
2750 E=R
2760 F=V
2770 GOTO 2200
2780 I=R
2790 J=V
2800 GOTO 2200
2810 O=R
2820 P=V
2830 GOTO 2200
2840 D=(D-112)/8
2850 IF S<>0 THEN 3000
2860 S=D
2870 ON S GOSUB 2890,2910,2930,2950,297
     0
2880 GOTO 2990
2890 CALL COLOR(9,2,14)
2900 RETURN
2910 CALL COLOR(9,15,13)
2920 RETURN
2930 CALL COLOR(9,15,2)
2940 RETURN
2950 CALL COLOR(9,2,7)
2960 RETURN
2970 CALL COLOR(9,2,5)
2980 RETURN
2990 ON S GOTO 3030,3050,3070,3090,3110
3000 IF K<>0 THEN 3130
3010 K=D
3020 ON K GOTO 3030,3050,3070,3090,3110
3030 B=0
3040 GOTO 2200
3050 F=0
3060 GOTO 2200
3070 H=0
3080 GOTO 2200
3090 J=0
3100 GOTO 2200
3110 P=0
3120 GOTO 2200
```

```
3130 R=22
3140 V=10
3150 X=S+8
3160 FOR Y=1 TO 4
3170 CALL CHAR((95+Y),HS(X))
3180 X=X+5
3190 NEXT Y
3200 FOR Y=1 TO 2
3210 CALL HCHAR(R,V,95+Y)
3220 V=V+1
3230 NEXT Y
3240 V=V-2
3250 R=R+1
3260 FOR Y=3 TO 4
3270 CALL HCHAR(R,V,95+Y)
3280 V=V+1
3290 NEXT Y
3300 CALL COLOR(10,15,6)
3310 CALL COLOR(11,15,6)
3320 R=22
3330 V=13
3340 Q=1
3350 Y=1
3360 FOR Y=Y TO Y+6
3370 CALL CHAR(103+Y,HS(28+Y))
3380 CALL HCHAR(R,V,103+Y)
3390 V=V+1
3400 NEXT Y
3410 IF Q=0 THEN 3470
3420 R=23
3430 V=13
3440 Q=0
3450 Y=8
3460 GOTO 3360
3470 PRINT :TAB(7);K;" PLACES"
3480 PRINT :TAB(7);D;" SHOWS"::
3490 PRINT "PRESS ANY KEY"
3500 CALL KEY(0,KEY,STATUS)
3510 IF STATUS=0 THEN 3500
3520 CALL COLOR(2,2,12)
3530 CALL CLEAR
3540 X=1
3550 IF NAME$(X)="LAST" THEN 4130
3560 ON KI(X)GOTO 3570,3640,3720,3810
3570 IF HO(X)=S THEN 3600
3580 GOSUB 3970
3590 GOTO 3880
3600 BET(X)=BET(X)*4
3610 BET(X)=INT(BET(X)*100+.5)/100
3620 GOSUB 4090
3630 GOTO 3880
3640 IF HO(X)=S THEN 3680
3650 IF HO(X)=K THEN 3680
3660 GOSUB 3970
3670 GOTO 3880
3680 BET(X)=BET(X)*3/2
3690 BET(X)=INT(BET(X)*100+.5)/100
3700 GOSUB 4090
3710 GOTO 3880
3720 IF HO(X)=S THEN 3770
3730 IF HO(X)=K THEN 3770
3740 IF HO(X)=D THEN 3770
3750 GOSUB 3970
3760 GOTO 3880
3770 BET(X)=BET(X)*2/3
3780 BET(X)=INT(BET(X)*100+.5)/100
3790 GOSUB 4090
3800 GOTO 3880
3810 IF HO(X)<>S THEN 3830
3820 IF PA2(X)=K THEN 3850
3830 GOSUB 3970
3840 GOTO 3880
3850 BET(X)=BET(X)*15
3860 BET(X)=INT(BET(X)*100+.5)/100
3870 GOSUB 4090
3880 X=X+1
3890 IF X>5 THEN 3550
3900 IF X>4 THEN 3920
3910 GOTO 3550
3920 GOTO 4130
3930 CALL CLEAR
3940 GOTO 3550
3950 IF X<=8 THEN 3550
3960 GOTO 3930
3970 IF TOT(X)<BET(X) THEN 4020
3980 PRINT "SO SORRY ";NAME$(X);" YOU LOSE $";BET(X)::
3990 TOT(X)=TOT(X)-BET(X)
4000 PRINT "YOU NOW HAVE $";TOT(X)::
4010 RETURN
4020 TTOT(X)=TOT(X)*-1
4030 PRINT "HEY! ";NAME$(X);" YOU LOSE AGAIN !"::
4040 TOT(X)=TOT(X)-BET(X)
4050 TTOT(X)=TOT(X)*-1
4060 PRINT "YOU OWE THE TRACK $";TTOT(X)::
4070 PRINT "WE HOPE YOUR CREDIT IS GOOD!"::
4080 RETURN
4090 TOT(X)=TOT(X)+BET(X)
4100 PRINT "GREAT! ";NAME$(X);" YOU WIN $";BET(X)::
4110 PRINT "YOU NOW HAVE $";TOT(X)::
4120 RETURN
4130 PRINT "PRESS ANY KEY"
4140 CALL KEY(0,KEY,STATUS)
4150 IF STATUS=0 THEN 4140
4160 IF NAME$(X)<>"LAST" THEN 3930
4170 CALL CLEAR
4180 L(K)=L(K)+1
4190 U(D)=U(D)+1
4200 W(S)=W(S)+1
4210 PRINT TAB(8);"PAST RECORDS"::::
4220 PRINT "NO:1 ";W(1);"WIN";L(1);"PLACE";U(1);"SHOW"
4230 PRINT :"NO:2 ";W(2);"WIN";L(2);"PLACE";U(2);"SHOW"
4240 PRINT :"NO:3 ";W(3);"WIN";L(3);"PLACE";U(3);"SHOW"
4250 PRINT :"NO:4 ";W(4);"WIN";L(4);"PLACE";U(4);"SHOW"
4260 PRINT :"NO:5 ";W(5);"WIN";L(5);"PLACE";U(5);"SHOW"
4270 PRINT :::"PRESS ENTER"
4280 CALL KEY(0,KEY,STATUS)
4290 IF STATUS=0 THEN 4280
4300 CALL CLEAR
4310 X=1
4320 IF NAME$(X)="LAST" THEN 1570
4330 GOSUB 1230
4340 GOTO 4320
4350 DATA 1,523,1,523,1,523,1,440,1,440,1,440,1,440,1,349,1,440,1,349,2,262
4360 DATA 1,349,1,440,1,523,1,523,1,523,1,440,1,440,1,440,1,262,1,262,1,330,2,349,0,0
4370 DATA 1,392,1,392,1,392,1,330,1,392,1,440,1,392,1,330,2,294,1,330,2,262,2,294
4380 DATA 1,392,1,392,1,392,1,330,1,392,1,440,1,392,2,330,2,294,1,330,1,262,2,294,2,262,0,0
```

EXTENDED BASIC



SPRITE CHASE

Wait . . . Wait . . . Wait . . . When will they get here? Wait . . . Wait . . . Wait . . . "Hi dear, anything in the mail today? Did you look between the doors? Oh. Shucks."

"Hello Ginny. What? You accepted a package from UPS for me? Great! Could you get it for me? Thanks."

"See you later dear, I'll be downstairs."

They're here . . .
The SPRITES are here . . .
NOW, WHAT CAN I DO WITH THEM?

Skim through the manual, page 25. Uh huh. OK. Yeah. This looks great! Let's get a little deeper. Page 64. Oh, oh. Looks like the ALL option of COINC doesn't tell you which SPRITES "coincidenced." I hope someone can find out where to PEEK for this.

Now, what shall I do with them? Something simple. Design some cute characters? No, let's just get those SPRITES moving. Since it might take some time for COINC (ALL,. . .) to figure out which SPRITES are coincidental, I'll stick to one SPRITE versus another. How about a series to chase? Numbers . . . Letters . . . ROTATION . . . That's it . . .

A short game chasing the 10 numbers or a longer game chasing the 26 letters. I'll try the MAGNIFY too. I'll need a numeric variable for the COINC tolerance for that. I guess 8 for normal size and 16 for double size. I'll generate the number or letter SPRITES to go any which way at some speed between −25 and 25. I'll stick to the 8 directions around the arrows for the chaser or else I'll get so tangled up in the math that I'll never move a SPRITE. Wish I had joysticks. I guess some kind of clock would be good for scoring.

Well, here we go!

EXPLANATION OF THE PROGRAM

Sprite Chase

Line Nos.	
170-200	Instructions.
210-280	Set up variations for play.
290-300	Reset for start of game.
310	Make clock numbers reverse image.
320-330	Put the Chaser somewhere in middle of the screen.
340-360	Create the Chasees.
370-390	The chase has begun.
400-450	While waiting for a direction key to be pressed, keep the clock going and check for a coincidence when the Chaser is stationary.
460-530	Start the Chaser in the direction of key pressed.
540-590	While awaiting release of direction key, check for a coincidence when the Chaser is moving; keep clock going.
600-610	Stop the Chaser; wait for another key to be pressed.
620-650	Caught one; go for the next one.
660-710	End of game.
720	That's it.

A FEW POSTSCRIPT NOTES:

If a SPRITE is moving slowly in a vertical direction, it might go off the top or bottom of the screen for a while, but it can be caught there.

If you insert COINC statements between a lot of the instructions and check the HIT field, you probably would reduce the number of times a coincidence is missed.

If you leave the Chaser in its original position, all targets will eventually pass through it. I wonder how long this would take?

(If it sounded like I was talking to myself, I was! Doesn't everyone???)

```
100 REM *********************
110 REM *  SPRITE CHASE  *
120 REM *********************
130 REM
140 REM
150 REM
160 REM
170 CALL CLEAR
180 PRINT "USE THE FOUR ARROW KEYS AND
    W,R,Z,C KEYS TO CHASE THE LETTE
    RS OR NUMBERS.":::PRINT
190 PRINT "YOU MUST CATCH THEM IN ALPH
    AOR NUMERIC SEQUENCE.":::PRINT
200 PRINT "PRESS 'L' FOR LARGE TARGETS
    ,'S' FOR SMALL TARGETS.":::
210 CALL KEY(0,GOT,STATUS)
220 IF STATUS=0 THEN 210
230 IF GOT=76 THEN T=16 ::: CALL MAGNIF
    Y(2)ELSE IF GOT=83 THEN T=8 ELSE 2
    10
240 PRINT "FOR NUMBERS PRESS 'N',":"FO
    R LETTERS PRESS 'L'.":::PRINT
250 CALL KEY(0,GOT,STATUS)
260 IF STATUS=0 THEN 250
270 IF GOT=78 THEN TARGS=10 ::: CH=47 E
    LSE IF GOT=76 THEN TARGS=26 ::: CH=
    64 ELSE 250
280 CALL CLEAR
290 RANDOMIZE
300 COUNT=0
310 CALL COLOR(3,2,9)::: CALL COLOR(4,2
    ,9)
320 CALL CHAR(96,"FFFFFFFFFFFFFFFF")
330 CALL SPRITE(#28,96,2,90,120,0,0)
340 FOR A=1 TO TARGS
350 CALL SPRITE(#A,A+CH,2,90,120,INT(R
    ND*50-25),INT(RND*30-25))
360 NEXT A
370 CALL SOUND(100,555,0)
380 FOR A=1 TO TARGS
390 CALL COLOR(#A,16)
400 CALL KEY(0,GOT,STATUS)
410 COUNT=COUNT+1
420 DISPLAY AT(24,1)SIZE(6):COUNT
430 CALL COINC(#28,#A,T,HIT)
440 IF HIT=-1 THEN 620
450 IF STATUS=0 THEN 400
460 IF GOT=69 THEN CALL MOTION(#28,-30
    ,0)::GOTO 540
470 IF GOT=88 THEN CALL MOTION(#28,30,
    0)::GOTO 540
480 IF GOT=68 THEN CALL MOTION(#28,0,3
    0)::GOTO 540
490 IF GOT=83 THEN CALL MOTION(#28,0,-
    30)::GOTO 540
500 IF GOT=87 THEN CALL MOTION(#28,-30
    ,-30)::GOTO 540
510 IF GOT=82 THEN CALL MOTION(#28,-30
    ,30)::GOTO 540
520 IF GOT=90 THEN CALL MOTION(#28,30,
    -30)::GOTO 540
530 IF GOT=67 THEN CALL MOTION(#28,30,
    30)::GOTO 540
540 CALL KEY(0,GOT,STATUS)
550 CALL COINC(#28,#A,9,HIT)
560 IF HIT=-1 THEN 620
570 COUNT=COUNT+1
580 DISPLAY AT(24,1)SIZE(6):COUNT
590 IF STATUS=-1 THEN 540
600 CALL MOTION(#28,0,0)
610 GOTO 400
620 CALL DELSPRITE(#A)
630 CALL SOUND(100,-7,0)
640 CALL MOTION(#28,0,0)
650 NEXT A
660 CALL CHARSET
670 PRINT "YOUR SCORE IS ":COUNT
680 PRINT "PRESS 'Y' TO PLAY AGAIN: "
690 CALL KEY(0,GOT,STATUS)
700 IF STATUS=0 THEN 690
710 IF GOT=89 THEN 280
720 PRINT "BYE"
```

DOGFIGHT

EXTENDED BASIC

Dogfight is a two-player game written in Extended BASIC. Each player has control of one aircraft—a biplane. You must outmaneuver your opponent and shoot him down before he can do the same to you. If both planes crash into each other, the score will not change. The first player to destroy 10 enemy aircraft wins the game.

Your plane is controlled with four directional keys. Unlike most games, the key pressed will not cause your plane to immediately move in that particular direction. For example, if your plane is traveling down and to the right at a bearing of 135 degrees and you press E or the up arrow, your plane will turn its nose up and first change its heading to 90 degrees, then to 45 degrees, and finally to due north—i.e., straight up. This gives the plane a more realistic movement and makes unrealistic, 180-degree hairpin turns impossible.

To make a 180-degree turn to go due west, you must first press either E or X to indicate the upward or downward turn. Pressing S, the left directional key, will have no effect. In a similar way, the player using the right-hand side of the keyboard uses I, J, K, and M to move the plane. (If you have the overlay for the *Video Games* Command Cartridge, you might find it convenient.)

Pressing the F and H keys will fire the guns, but only one shot can be fired at a time. Each shot has a limited range and cannot be carried over the edge of the screen to the opposite edge. You can't terminate a bad shot; it must first go off screen. The limit of only one shot at a time was placed in the program so that the computer could make accurate coincidence checks with rapidly moving objects on the screen.

Four levels of difficulty make the game easy enough for beginners and challenging enough to hold the interest of experts. The higher the level of difficulty, the faster the planes and shots will move. The option to fly a day or night mission will change the screen color to either light blue or black.

Features of Extended BASIC Used in *Dogfight*

For you *99'er* readers who are also programmers, *Dogfight* illustrates many features of Extended BASIC.

Lines 190-280 define special graphics characters four at a time. Then line 290, CALL MAGNIFY(3) indicates that sprites will actually consist of four regular size characters, and only the first character number needs to be specified.

CALL SPRITE(#1,96,2,120,20,0,5) defined Sprite #1 (the first plane) as characters 96, 97, 98, and 99, black, starting in dot-row 120 and dot-column 20, and going zero velocity up or down, and to the right at velocity 5. More than one sprite may be defined at a time, as in line 310.

CALL DELSPRITE(#1) deletes Sprite #1, and more than one sprite may be deleted at a time. This statement is used when the planes are hit or they crash, or when the bullet leaves the edge of the screen.

Complex IF-THEN-ELSE statements are used in lines 530-780 to determine in what direction the plane is headed. CALL PATTERN then draws the plane depending on its heading; all other sprite characteristics remain the same.

CALL POSITION(#1,B1,B2) in line 1000 returns the row and column position of Sprite #1 so the bullet can be shot from that same position.

CALL MOTION can change the motion of a sprite without affecting other characteristics.

CALL COINC(#2,#3,3*V,PC) determines if Sprite #2 (second plane) and Sprite #3 (bullet from first plane) are coincident within a tolerance of 3*V. If so, a value of −1 is returned for PC. Coincidence is reported only when the CALL COINC statement is actually executed during the program. At greater velocities of the sprites, coincidence will not be detected if the program is busy elsewhere.

There are several ways to avoid concidences not being detected: 1) CALL COINC more often; 2) increase the tolerance level with increasing velocity; or 3) increase the execution speed so that the CALL COINC statements are executed more often. But, as with many programming problems, the solution involves a trade-off: The first slows down other action; the second could have planes crash when they're not touching; the third means cutting back on other types of action. One low-cost way to speed up execution, however, is to use multiple statements per line: CALL MOTION (#1,0,V)::RETURN executes faster than two separate lines with those statements.

Although kept to a minimum, the coincidence problem does still exist in this program: Once in a while a bullet will pass right through a plane. You'll just have to visualize a three-dimensional situation—a bullet passing directly over or under the plane but at a different altitude. Also, once in a while a bullet won't disappear at the edge of the screen, but will "wrap." Just consider it a stray bullet—a frequent occurrence in a real dogfight.

EXPLANATION OF THE PROGRAM

Dogfight

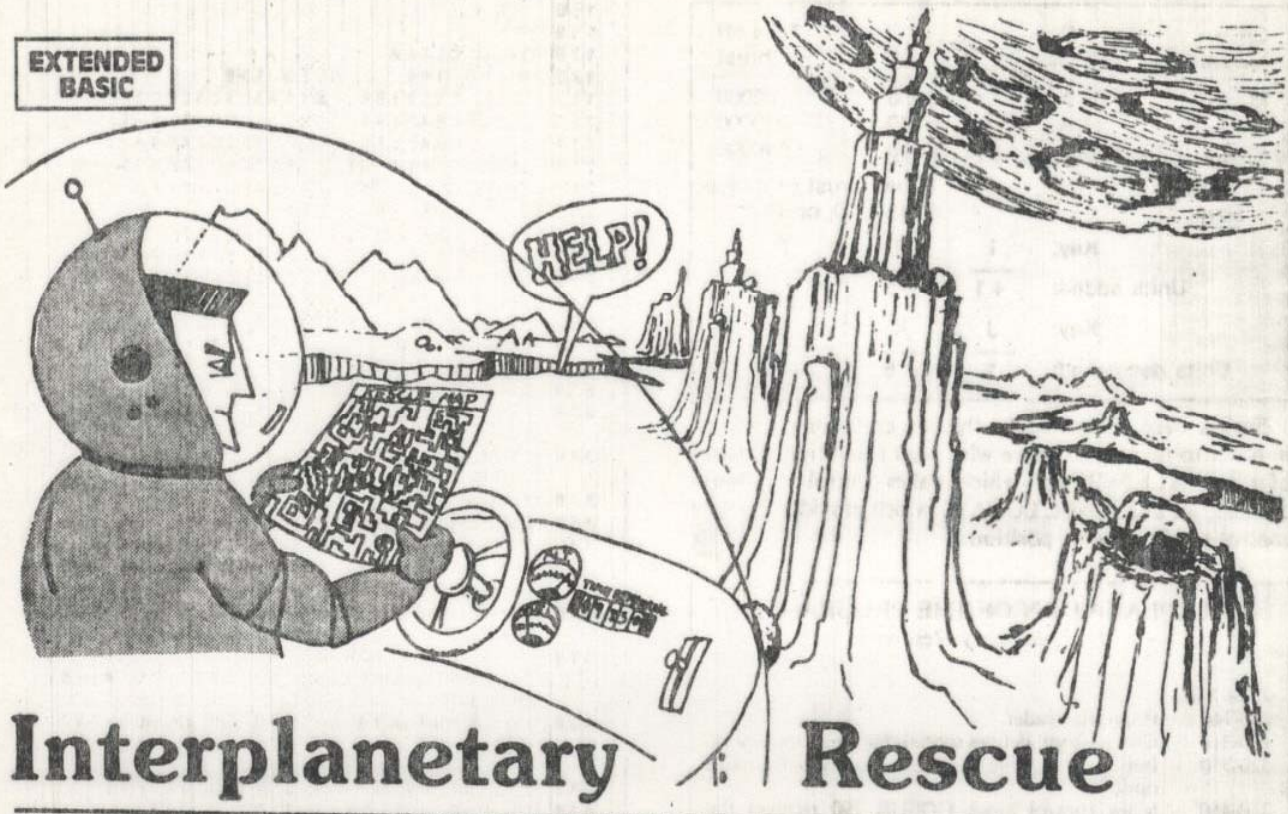
Line Nos.	
170	Clears screen, makes screen light blue.
180	P is the orientation of Plane 1; P1 the orientation of Plane 2.
190-290	Defines special characters for graphics. Using CALL MAGNIFY(3) allows sprites to be defined as 4 regular-sized graphics characters.
300-340	Draws title screen with two planes and sound.
350-360	Receives players' names for scoring.
370-420	Allows choice of one of four options.
430-460	Allows choice of day or night mission.
470-490	Starts *Dogfight* with planes going toward each other at velocity V.
500	Waits for players to press key.
510	If no key is pressed, branches to checking coincidence.
520-660	If key on left half of keyboard is pressed, checks which key and either moves Plane #1 that direction or shoots.
670-810	If key on right half of keyboard is pressed, moves Plane #2 appropriately or shoots.
820-850	Checks for coincidence between bullets and planes or between planes. If the sprites are coincident, branches to section of program for crashing. If not, continues sound of flying.
860-890	Deletes the bullet at the edges of the screen.
900	Returns to line 500 to check players' action.
910-980	Subroutines to move Plane #1 the correct direction after turning.
990	Returns if a bullet from Plane #1 is in motion.
1000-1100	Starts a bullet from the plane.
1110-1180	Subroutines to move Plane #2 the correct direction.
1190-1210	Returns if a bullet from Plane #2 is already in motion; otherwise shoots from Plane #2.
1220-1340	Procedure if planes crash into each other; makes crashing noise; draws planes falling; deletes sprites at the bottom of the screen.
1350-1410	Procedure if Plane #1 is shot.
1420-1490	Increments score for Player 2 and prints score.
1500-1560	Procedure if Plane #2 is shot.
1570-1630	Increments score for Player 1 and prints scores.
1640	Ends game if either player's score is 10.
1650-1680	If score is less than 10, players may choose to try again or stop.
1690-1720	Prints final score and ending remarks.

```
100 REM *********
110 REM *  DOG  FIGHT  *
120 REM *********
130 REM
140 REM
150 REM
160 REM
170 CALL CLEAR :: CALL SCREEN(6)
180 P=1 :: P1=5
190 CALL CHAR(96,"00000000081C0E2F73F3E
    0710200000000000000F8017DFDFF95FD
    21707070")
200 CALL CHAR(100,"0000001020408090103070
    6FFFF7E4A0202020900086CFAE9D8B07CDC9
    8")
210 CALL CHAR(104,"0F0107FFFFFFFFFF7F0
    6070703060E1EE000C0FCFEFEFEFEFCC0C
    0C080C0E0F0")
220 CALL CHAR(108,"0409103659F971B0D3E3
    B1900000000000000080402090800D0E060F
    FFF7C52404040")
230 CALL CHAR(112,"000000000001F80BEBFFFA
    0BF840E0E0000000000000081034 7EFFCFCF
    0030C4")
240 CALL CHAR(116,"0000000000050811274F1
    E9D4B3C1008043033C383CF8FFB060C080E
    0E0C0")
250 CALL CHAR(120,"0F0703010303033F7F7
    F7F3F03000077870600C0E0E060FEFFFFF
    FFFFEE080F0")
260 CALL CHAR(124,"0C3C1C1C3C1FFF0D060
    301070703000000000000A01088E4F278B
    9D27C08102040")
270 CALL CHAR(128,"0000000000000000010101
    0000000000000000000000008080")
280 CALL CHAR(132,"FC7E3C381810C0401203
    0301C1F03000FC7838780F0604080040C0
    C38F0C0")
290 CALL MAGNIFY(3)
300 DISPLAY AT(12,9):"DOG FIGHT"
310 CALL SPRITE(#1,96,2,120,20,0,5,#2,
    112,2,70,240,0,-5)
320 FOR TD=1 TO 2 :: CALL SOUND(1500,1
    10,2,220,2,1000,30,-8,2)
330 NEXT TD :: CALL SOUND(1,9999,30)
340 CALL DELSPRITE(#1,#2)::  CALL CLEAR
350 DISPLAY AT(5,1):"ENTER FIRST PLAYE
    R'S NAME:" :: ACCEPT AT(7,3)SIZE(8
    ):PLA1$
360 DISPLAY AT(11,1):"ENTER SECOND PLA
    YER'S NAME:" :: ACCEPT AT(13,3)SIZ
    E(8):PLA2$
370 CALL CLEAR
380 DISPLAY AT(4,3):"OPTIONS:"
390 DISPLAY AT(6,5):"1. BEGINNER" :: D
    ISPLAY AT(8,5):"2. NOVICE"
400 DISPLAY AT(10,5):"3. INTERMEDIATE"
    :: DISPLAY AT(12,5):"4. PROFESSIO
    NAL"
410 CALL KEY(0,KEY,S)
420 IF KEY<49 OR KEY>52 THEN 410 ELSE
    V=2*(KEY-48)
430 CALL CLEAR :: DISPLAY AT(5,3):"DO
    YOU WANT A DAY MISSION, OR A NIGH
    T MISSION? (D/N)"
440 CALL KEY(0,KEY,S)
450 IF KEY=68 THEN CO1=6 :: CO2=2 :: G
    OTO 470
460 IF KEY=78 THEN CO1=2 :: CO2=6 ELSE
    440
470 CALL CLEAR
480 CALL SCREEN(CO1)
490 CALL SPRITE(#1,96,8,70,16,0,V,#2,1
    12,11,70,240,0,-V)
500 CALL KEY(1,K1,S1):: CALL KEY(2,K2,
    S2)
510 IF S1=0 AND S2=0 THEN 820
520 IF S1=0 THEN 680
530 IF K1<>5 THEN 570 ELSE IF P>5 AND
    P<8 THEN P=P-1 ELSE IF P<3 OR P=8
    THEN P=P+1
540 IF P=0 THEN P=8
550 IF P=9 THEN P=1
560 GOTO 650
570 IF K1<>0 THEN 610 ELSE IF P>3 AND
    P<7 THEN P=P+1 ELSE IF P<3 OR P=8
    THEN P=P-1
580 IF P=9 THEN P=1
590 IF P=0 THEN P=8
600 GOTO 650
610 IF K1<>2 THEN 630 ELSE IF P>5 THEN
    P=P-1 ELSE IF P<5 AND P>1 THEN P=
    P+1
620 GOTO 650
630 IF K1<>3 THEN 680 ELSE IF P<5 AND
    P>1 THEN P=P-1 ELSE IF P>5 THEN P=
    P+1
640 IF P=9 THEN P=1
650 CALL PATTERN(#1,(P*4)+92):: ON P G
    OSUB 910,920,930,940,950,960,970,9
    80 :: GOTO 670
660 IF K1=12 THEN GOSUB 990
670 IF S2=0 THEN 820
680 IF K2<>5 THEN 720 ELSE IF P1>3 AND
    P1<8 THEN P1=P1-1 ELSE IF P1<3 OR
    P1=8 THEN P1=P1+1
```

```
690 IF P1=0 THEN P1=8
700 IF P1=9 THEN P1=1
710 GOTO 800
720 IF K2<>8 THEN 760 ELSE IF P1>3 AND
    P1<7 THEN P1=P1+1 ELSE IF P1<3 OR
    P1=8 THEN P1=P1-1
730 IF P1=0 THEN P1=8
740 IF P1=9 THEN P1=1
750 GOTO 800
760 IF K2<>2 THEN 780 ELSE IF P1>5 THE
    N P1=P1-1 ELSE IF P1<5 AND P1>1 TH
    EN P1=P1+1
770 GOTO 800
780 IF K2<>3 THEN 810 ELSE IF P1<5 AND
    P1>1 THEN P1=P1-1 ELSE IF P1>5 TH
    EN P1=P1+1
790 IF P1=9 THEN P1=1
800 CALL PATTERN(#2,(P1*4)+92)::: ON P1
    GOSUB 1110,1120,1130,1140,1150,11
    60,1170,1180 ::: GOTO 820
810 IF K2=1 THEN GOSUB 1190
820 CALL COINC(#2,#3,3*V,PC)::: IF PC=-
    1 THEN 1500
830 CALL COINC(#1,#4,3*V,PC)::: IF PC=-
    1 THEN 1350
840 CALL COINC(#1,#2,2.5*V,PC)::: IF PC
    =-1 THEN 1220
850 CALL SOUND(-4250,110,2,220,2,-8,2)
860 IF SH1=0 THEN 880
870 CALL POSITION(#3,B3,B4)::: IF B3<6
    OR B3>186 OR B4<8 OR B4>250 THEN C
    ALL DELSPRITE(#3)::: SH1=0
880 IF SH2=0 THEN 900
890 CALL POSITION(#4,C3,C4)::: IF C3<6
    OR C3>186 OR C4<8 OR C4>250 THEN C
    ALL DELSPRITE(#4)::: SH2=0
900 GOTO 500
910 CALL MOTION(#1,0,V)::: RETURN
920 CALL MOTION(#1,-V*.6,V*.6)::RETURN
930 CALL MOTION(#1,-V,0)::: RETURN
940 CALL MOTION(#1,-V*.6,-V*.6)::: RETU
    RN
950 CALL MOTION(#1,0,-V)::: RETURN
960 CALL MOTION(#1,V*.6,-V*.6)::RETURN
970 CALL MOTION(#1,V,0)::: RETURN
980 CALL MOTION(#1,V*.6,V*.6)::: RETURN
990 IF SH1=1 THEN RETURN
1000 CALL POSITION(#1,B1,B2)
1010 P3=P ::: SP=3 ::: A1=B1 ::: A2=B2 :::
    SH1=1
1020 ON P3 GOTO 1030,1040,1050,1060,107
    0,1080,1090,1100
1030 CALL SPRITE(#SP,128,CO2,A1,A2,0,V*
    2)::: RETURN
1040 CALL SPRITE(#SP,128,CO2,A1,A2,-V*1
    .6,V*1.6)::: RETURN
1050 CALL SPRITE(#SP,128,CO2,A1,A2,-V*2
    ,0)::: RETURN
1060 CALL SPRITE(#SP,128,CO2,A1,A2,-V*1
    .6,-V*1.6)::: RETURN
1070 CALL SPRITE(#SP,128,CO2,A1,A2,0,-V
    *2)::: RETURN
1080 CALL SPRITE(#SP,128,CO2,A1,A2,V*1.
    6,-V*1.6)::: RETURN
1090 CALL SPRITE(#SP,128,CO2,A1,A2,V*2,
    0)::: RETURN
1100 CALL SPRITE(#SP,128,CO2,A1,A2,V*1.
    6,V*1.6)::: RETURN
1110 CALL MOTION(#2,0,V)::: RETURN
1120 CALL MOTION(#2,-V*.6,V*.6)::RETURN
1130 CALL MOTION(#2,-V,0)::: RETURN
1140 CALL MOTION(#2,-V*.6,-V*.6)::: RETU
    RN
1150 CALL MOTION(#2,0,-V)::: RETURN
1160 CALL MOTION(#2,V*.6,-V*.6)::RETURN
1170 CALL MOTION(#2,V,0)::: RETURN
1180 CALL MOTION(#2,V*.6,V*.6)::: RETURN
1190 IF SH2=1 THEN RETURN
```

```
1200 CALL POSITION(#2,C1,C2)
1210 A1=C1 ::: A2=C2 ::: P=P1 ::: SP=4 :::
     SH2=1 ::: GOTO 1020
1220 CALL MOTION(#1,0,0,#2,0,0)::: CALL
     POSITION(#1,P01,P02,#2,P03,P04)
1230 CALL SPRITE(#1,132,10,P01,P02,20,0
     ,#2,132,10,P03,P04,20,0)
1240 CALL POSITION(#1,CR1,CR2)
1250 CALL POSITION(#2,CR3,CR4)
1260 IF CR1>186 OR CR3>186 THEN CALL DE
     LSPRITE(#1,#2)
1270 CALL SOUND(-1000,110,2,220,2,-7,0)
1280 IF CR1<>0 OR CR3<>0 THEN 1240
1290 CALL DELSPRITE(#1,#2,#3,#4)
1300 CALL SCREEN(6)
1310 DISPLAY AT(5,13):"CRAAAASSSH"
1320 DISPLAY AT(10,5):"BOTH TEAMS CRASH
     ED."
1330 DISPLAY AT(12,5):"NO POINTS AWARDE
     D."
1340 GOTO 1640
1350 CALL DELSPRITE(#3,#4)
1360 CALL MOTION(#1,0,0)::: CALL POSITIO
     N(#1,P01,P02)
1370 CALL SPRITE(#1,132,10,P01,P02,20,0
     )
1380 CALL SOUND(-1000,440,2,220,2,-4,0)
1390 CALL POSITION(#1,CR1,CR2)
1400 IF CR1<186 THEN 1380
1410 CALL DELSPRITE(#1,#2)
1420 RED=RED+1
1430 CALL SCREEN(6)
1440 DISPLAY AT(5,1):PLA2$;" HAS WON TH
     E BATTLE."
1450 DISPLAY AT(9,3):PLA2$;" SCORE IS :
     "
1460 DISPLAY AT(9,26):RED
1470 DISPLAY AT(11,3):PLA1$;" SCORE IS
     :"
1480 DISPLAY AT(11,26):BLUE
1490 GOTO 1640
1500 CALL DELSPRITE(#3,#4)
1510 CALL MOTION(#2,0,0)::: CALL POSITIO
     N(#2,P01,P02)
1520 CALL SPRITE(#2,132,10,P01,P02,20,0
     )
1530 CALL SOUND(-1000,440,2,220,2,-4,0)
1540 CALL POSITION(#2,CR3,CR4)
1550 IF CR3<186 THEN 1530
1560 CALL DELSPRITE(#2,#1)
1570 BLUE=BLUE+1
1580 CALL SCREEN(6)
1590 DISPLAY AT(5,1):PLA1$;" HAS WON TH
     E BATTLE."
1600 DISPLAY AT(8,3):PLA1$;" SCORE IS :
     "
1610 DISPLAY AT(8,26):BLUE
1620 DISPLAY AT(10,3):PLA2$;" SCORE IS
     :"
1630 DISPLAY AT(10,26):RED
1640 IF RED=10 OR BLUE=10 THEN 1690
1650 DISPLAY AT(20,3):"WANT TO TRY AGAI
     N? (Y/N)"
1660 PC=0 ::: SH1=0 ::: SH2=0 ::: P=1 ::: P
     1=5
1670 CALL KEY(0,KEY,S)
1680 IF KEY=89 THEN 370 ELSE IF KEY<>78
     THEN 1670
1690 IF BLUE>RED THEN PRINT PLA1$;" HAS
     WON THE WAR.":"SORRY, ";PLA2$ ::
     GOTO 1720
1700 IF RED>BLUE THEN PRINT PLA2$;" HAS
     WON THE WAR.":"SORRY, ";PLA1$ ::
     GOTO 1720
1710 PRINT "THE WAR HAS ENDED IN A TIE.
     BOTH SIDES ARE NOW AT PEACE."
1720 END
```

EXTENDED BASIC



Interplanetary Rescue

You are sitting there enjoying a cup of coffee at the Interplanetary Rescue Lounge when the news arrives: A cave-in on Moon Base 2 has seriously injured a miner. Instantly you race for the shuttle, knowing you must reach the moon base, pick up the injured miner, and return to the base medical facilities. There's no time to waste!

Your ship is fueled and ready at the docking pad. In your ship you sit in front of your TI-99/4A controller panel and view the radar and instrumentation screen. You are now ready to take off. Press T on the control panel and the shuttle begins to lift. Using your six thrust control buttons, you adjust your climb to the desired level. The terrain between you and Moon Base 2 is treacherous, and you must quickly ascertain the best route. Using your horizontal thrusters (arrow keys), you start your trip across the lunar landscape.

Accidents may happen anywhere, and right now your Interplanetary Rescue Team is in charge of the moon, Mars, and Venus. Use the arrow keys (E,S,D,X) to control horizontal movement. Horizontal velocity is listed on your control screen.

The elevations of the terrain show up as different colors on the map. At the right-hand side of the screen is a visual representation of your altitude in relation to the elevation colors. Your ship must be above the color on the right of the screen to safely pass through that color on the radar screen. Be careful not to overshoot the highest elevation color you plan to cross: You will waste valuable time getting back down and precious fuel needed for the return voyage.

When you land on Moon Base 2, you must be traveling at a vertical speed of less than 6 meters/second to make a perfect landing. A rough landing of 6-10 meters/second will cause a leak in your main fuel valve, causing a loss of half your fuel. Any faster than 10 meters/second and you'll crash, never to return home. Once safely on the ground, the injured miner is put on board, and you're ready for the return trip. You won't have as much fuel holding you down, so it won't take as much thrust to accelerate vertically. Good luck on your rescue mission . . . you may need it!

Instrument Displays:

ALT = altitude in meters. Each succeeding color on the radar represents 2000 meters.

HVEL = horizontal velocity across the radar screen (dependent upon arrow keys).

VVEL = vertical velocity; + is climbing, – is falling.

TIME = number of seconds into the mission.

FUEL = weight of fuel remaining, in kilograms.

PWR = amount of thrust being generated. Each unit of thrust equals 1000 newtons; each newton equals approximately 2.05 kilograms of thrust.

Calculations and Variables

The gravity formula in line 950 is the formula for speed of a falling object. V2 is the change in velocity in m/sec, F is the thrust in newtons, S is the weight of the ship in kg., E is the weight of remaining fuel in kg., and G is the gravity in m/sec^2. The time is one second in this calculation. All variables starting with D pertain to distance, and H is the altitude.

Graphics

Characters accessible on the keyboard but not used in printing messages have been redefined (lines 190-290). Then by using DISPLAY AT and a string, you can display colorful graphics very quickly without calling each square one-by-one. This method was used to display the radar map

much more quickly than by using HCHAR and VCHAR. The strings are read in as 21 DATA statements. By changing the DATA statements in this program or adding some of your own, you may easily change the maps.

Option chosen	G (Gravity)	E (Fuel)	Take-off thrust
Moon	2	20000	65000
Mars	4	45000	230000
Venus	6	80000	540000

Thrust keys: T = initial thrust (displayed as 65, 230, or 540).

Key:	I	O	P
Units added:	+1	+5	+10
Key:	J	K	L
Units decreased:	-1	-5	-10

Sprites were used to depict the two crafts on the screen in order to be able to move with high resolution. Instead of using CALL MOTION which makes control of position difficult, we used CALL LOCATE which provides absolute control of the sprite's position.

EXPLANATION OF THE PROGRAM
Interplanetary Rescue

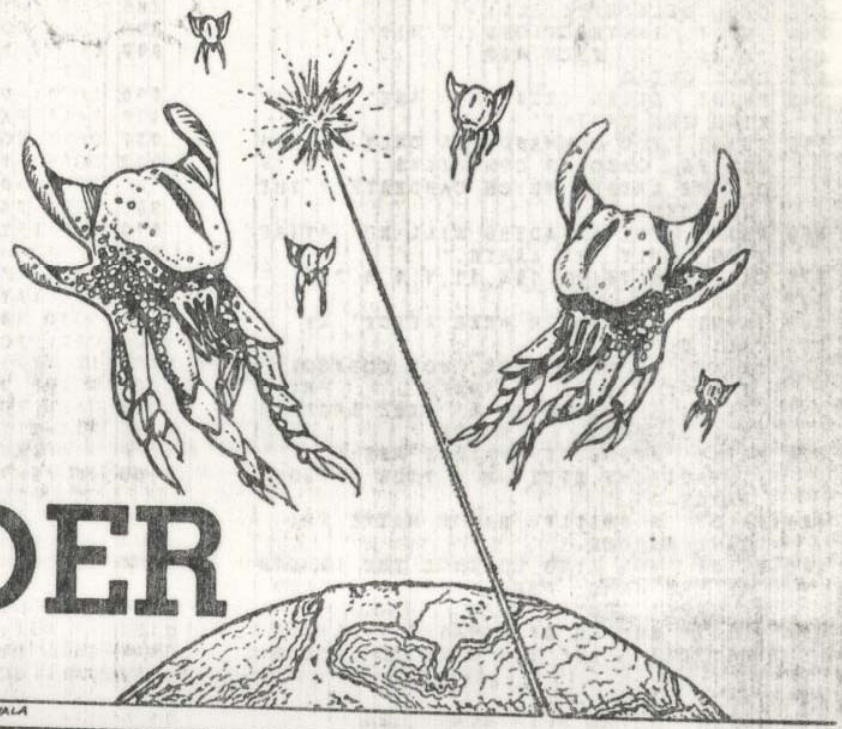
Line Nos.	
100-160	Program header
170-310	Clears screen; defines special characters and colors.
320-350	Initializes variables; branches to title screen and options.
360-460	Main control loop; GOSUB 790 receives the player's key presses. The VOL in the CALL SOUND statement depends upon the power. H is the altitude, and line 400 tests for crashes. If the rescue craft has landed at either base, the program branches.
470-680	DATA statements to draw the "Novice" map.
690-780	Subroutine to label the parameters and draw the altimeter.
790-1020	Receives player's key responses and calculates parameters.
1030-1090	Prints updated altitude, time, velocities, fuel, and power.
1100-1150	Message and procedure for crashing into the hill.
1160-1180	Subroutine for procedure for any crash.
1190-1260	Prints score and option to play again; branches appropriately.
1270-1400	Messages and procedure for crashing at high velocity.
1410-1460	Procedure if craft lands safely; starts return trip.
1470-1580	Procedure for craft landing on return trip.
1590-1640	Prints and receives options of planet and level of difficulty.
1650-1750	Depending on the options chosen, sets gravity, fuel, and initial thrust, then prints appropriate map.
1760-2380	DATA statements for three maps.
2390-2460	Prints title screen.

```
100 REM *********************
110 REM *  INTERPLANETARY RESCUE  *
120 REM *********************
130 REM BEST OF 99'ER
140 REM 99'ER VERSION 1.4.2XB
150 REM
160 REM
170 CALL CLEAR
180 GOSUB 1990 :: GOTO 320
190 CALL CHAR(96,"81423C3C3C3C4281")
200 CALL CHAR(62,"FF8181999981FF")
210 CALL CHAR(99,"026C9E1C24424201")
220 CALL CHAR(100,"00CC86C3001070C2")
230 CALL CHAR(33,"FFFFFFFFFFFFFFFF")
240 CALL CHAR(94,"0000")
250 CALL CHAR(95,"FFFFFFFFFFFFFFFF")
260 CALL CHAR(42,"FFFFFFFFFFFFFFFF")
270 CALL CHAR(63,"FFFFFFFFFFFFFFFF")
280 CALL CHAR(98,"183C3C7E7E7E66C3")
290 CALL CHAR(104,"5A5A5A185A")
300 CALL COLOR(1,4,1,2,5,1)
310 RETURN
320 CALL SCREEN(16)
330 V=0 :: V1=0 :: V2=0 :: S=5000 :: F=0 :: H=0 :: D=0 :: T=0
340 TRIP=0 :: TIME=0 :: D1=0 :: D2=0 :: D3=25 :: D4=41 :: FF=6
350 GOSUB 1590
360 REM MAIN CONTROL LOOP
370 GOSUB 790 :: VOL=ABS((6000000-F)/2000):: IF VOL>30 THEN VOL=30 ELSE IF VOL<0 THEN VOL=0
380 IF F>1 THEN CALL SOUND(-4250,110,VOL,220,VOL,110,VOL,-5,VOL)
390 CALL POSITION(#1,XC,YC):: CALL GCHAR(ABS((XC+4)/8+.5),ABS((YC+4)/8+.5),CC)
400 IF CC=94 AND H<2000 OR CC=95 AND H<4000 OR CC=42 AND H<6000 OR CC=63 AND H<8000 THEN 1100
410 IF TRIP=0 AND H>0 THEN TRIP=1
420 IF H=0 AND TRIP=1 THEN 1270
430 IF TRIP=2 AND H>0 THEN TRIP=3
440 IF TRIP=3 AND H=0 THEN 1470
450 TIME=TIME+1 :: IF H<=0 THEN V,V1,H=0
460 GOTO 370
470 DATA
480 DATA
490 DATA
500 DATA
510 DATA
520 DATA
530 DATA
540 DATA
550 DATA
560 DATA
570 DATA
580 DATA
590 DATA
600 DATA
610 DATA
620 DATA
630 DATA
640 DATA
650 DATA
660 DATA
670 DATA
680 RETURN
690 DISPLAY AT(22,1):"ALT "
700 DISPLAY AT(23,1):"HVEL"
710 DISPLAY AT(24,1):"VVEL"
720 DISPLAY AT(22,15):"TIME"
730 DISPLAY AT(23,15):"FUEL"
740 DISPLAY AT(24,15):"PWR "
750 CALL VCHAR(6,30,63,4):: CALL VCHAR(10,30,42,4):: CALL VCHAR(14,30,95,4)
```

```
760 CALL VCHAR(18,30,94,4):: CALL HCHA
    R(22,28,33,3)
770 CALL SPRITE(#2,98,2,160,222,#1,96,
    2,D3,D4)
780 RETURN
790 CALL KEY(1,K1,S1):: CALL KEY(2,K2,
    S2):: IF S1=0 AND S2=0 THEN 920
800 IF S1=0 THEN 860
810 IF K1=5 THEN D1=D1-1 :: E=E-50 ::
    GOTO 860
820 IF K1=0 THEN D1=D1+1 :: E=E-50 ::
    GOTO 860
830 IF K1=2 THEN D2=D2-1 :: E=E-50 ::
    GOTO 860
840 IF K1=3 THEN D2=D2+1 :: E=E-50 ::
    GOTO 860
850 IF K1=11 THEN F=TOFF
860 IF K2=3 THEN F=F-5000 :: GOTO 920
870 IF K2=12 THEN F=F-10000 :: GOTO 92
    0
880 IF K2=2 THEN F=F-1000 :: GOTO 920
890 IF K2=5 THEN F=F+1000 :: GOTO 920
900 IF K2=6 THEN F=F+5000 :: GOTO 920
910 IF K2=11 THEN F=F+10000
920 IF E<=0 THEN E=0 :: F=0
930 IF F<0 THEN F=0
940 IF F=0 THEN CALL PATTERN(#3,32)ELS
    E CALL PATTERN(#3,104)
950 V2=F/(S+E)-G :: V=V+V2 :: DV=V ::
    IF V<0 AND H<=0 THEN DV=0
960 D=(V1+(V2/2)):: V1=V :: H=H+D :: E
    =E-(ABS(F/2000))
970 IF H<=0 THEN H=0
980 IF H>9935 THEN 1000
990 CALL LOCATE(#2,160-(H/(500/8)),222
    ,#3,168-(H/(500/8)),222)
1000 D3=D3+D1 :: IF D3<1 THEN D3=1 ELSE
     IF D3>160 THEN D3=160
1010 D4=D4+D2 :: IF D4<17 THEN D4=17 EL
    SE IF D4>208 THEN D4=208
1020 CALL LOCATE(#1,D3,D4)
1030 DISPLAY AT(22,5)SIZE(6):H
1040 DISPLAY AT(22,20)SIZE(5):TIME
1050 DISPLAY AT(23,5)SIZE(5):SQR(D1^2+D
    2^2)*62.5
1060 DISPLAY AT(23,20)SIZE(6):E
1070 DISPLAY AT(24,5)SIZE(5):DV
1080 DISPLAY AT(24,20)SIZE(7):F/1000
1090 RETURN
1100 CALL HCHAR(22,1,32,96):: GOSUB 116
    0
1110 CALL CHARSET :: DISPLAY AT(22,3):"
    YOU CRASHEDD INTO THE HILL."
1120 IF TRIP=1 AND TIME<250 THEN CR=TIM
    E/2000 ELSE CR=.15
1130 DISPLAY AT(23,1):"   ALTITUDE=";H
1140 DISPLAY AT(24,3):"VELOCITY=";V
1150 GOTO 1190
1160 FOR REP=1 TO 5 :: CALL SOUND(300,1
    10,0,110,0,110,0,-8,0):: CALL PATT
    ERN(#1,99)
1170 CALL SOUND(400,110,0,110,0,220,0,-
    4,0):: CALL PATTERN(#1,100):: NEXT
     REP
1180 CALL CLEAR :: CALL DELSPRITE(ALL):
    : CALL CHARSET :: RETURN
1190 FOR TD=1 TO 500
1200 NEXT TD [illegible]
    [illegible] THEN [illegible] ELSE IF [illegible]
    [illegible] DISPLAY AT(20,3[illegible]
    [illegible] AGAIN?(Y/N)"
[illegible] [illegible]Y AT(18,3):"YOUR SCORE IS ";
    [illegible](2000-2*TIME)+E/G+(OPT1*OPT2*
    [illegible]*CR)
[illegible] ACCEPT AT(23,5)BEEP:ANS$
1240 IF ANS$="N" THEN 1260 ELSE IF ANS$
    <>"Y" THEN 1230
1250 CALL CLEAR :: GOTO 170
1260 STOP
1270 CALL HCHAR(22,1,32,96)
1280 IF XC<>137 AND YC<>185 THEN GOSUB
    1160 :: CR=.08 :: DISPLAY AT(22,1
    ):"YOU MISSED THE PAD.":"YOUR SHIP
    HAS CRASHED." :: GOTO 1190
1290 IF V>-30 THEN 1330 ELSE GOSUB 116
    :: DISPLAY AT(22,1):"YOU BLEW IT;
    YOU LEFT A"
1300 DISPLAY AT(23,1):"CRATER A MILE WI
    DE." :: CR=.1
1310 DISPLAY AT(24,1):"VELOCITY=";V
1320 GOTO 1190
1330 IF V>-20 THEN 1370 ELSE GOSUB 116
    :: DISPLAY AT(22,1):"A BAD LANDI
    G-TWO CREWMEN"
1340 DISPLAY AT(23,1):"ARE DEAD, AND YO
    U ARE HURT." :: CR=.15
1350 DISPLAY AT(24,1):"VELOCITY=";V
1360 GOTO 1190
1370 IF V>-10 THEN 1410 ELSE GOSUB 116
    :: DISPLAY AT(22,1):"YOUR SHIP IS
     BADLY DAMAGED."
1380 DISPLAY AT(23,1):"THIS IS YOUR LAS
    T FLIGHT." :: CR=.25
1390 DISPLAY AT(24,1):"VELOCITY=";V
1400 FF=FF-3000 :: GOTO 1190
1410 IF V>-6 THEN 1430 ELSE CALL CLEAR
    :: CALL CHARSET :: DISPLAY AT(22,1
    ):"A ROUGH LANDING. YOU HAVE"
1420 DISPLAY AT(23,1):"LOST 1/2 OF YOUR
     FUEL." :: E=E/2 :: CR=.7 :: GOTO
    1450
1430 CALL CLEAR :: CALL CHARSET :: DISP
    LAY AT(22,1):"A PERFECT LANDING. Y
    OU ARE"
1440 DISPLAY AT(23,1):"IN GOOD SHAPE TO
     RETURN." :: CR=1
1450 DISPLAY AT(24,1):"VELOCITY=";V ::
    FOR TD=1 TO 500
1460 NEXT TD :: F=0 :: TRIP=2 :: GOSUB
    190 :: GOSUB 1680 :: GOTO 370
1470 CALL DELSPRITE(#1,#2,#3)
1480 CALL CLEAR
1490 CALL CHARSET :: IF V<=-10 THEN 127
    0 ELSE CALL HCHAR(22,1,32,96)
1500 IF V>-6 THEN 1550
1510 IF TRIP=1 AND(XC<>137 OR YC<>185)T
    HEN 1270
1520 IF TRIP=3 AND(D3<>25 OR D4<>41)THE
    N 1270
1530 DISPLAY AT(22,1):"A ROUGH LANDING.
     YOUR SHIP BARELY MADE IT."
1540 CR=CR*.75 :: GOTO 1570
1550 DISPLAY AT(22,1):"CONGRADULATIONS,
     A PERFECT"
1560 DISPLAY AT(23,1):"LANDING. EARTH I
    S PROUD."
1570 DISPLAY AT(24,1):"VELOCITY=";V ::
    FOR TD=1 TO 2000
1580 NEXT TD :: GOTO 1190
1590 CALL CLEAR :: GOSUB 2390 :: CALL C
    LEAR :: DISPLAY AT(1,7):"PLANET OP
    TIONS"
1600 DISPLAY AT(3,1):"1. MOON"::"2. MA
    RS"::"3. VENUS"
1610 ACCEPT AT(10,1)VALIDATE("123")SIZE
    (1):OPT1 :: CALL CLEAR
1620 DISPLAY AT([illegible]):"[illegible]LEVEL OF [illegible]
    [illegible]
    [illegible]
     PROFESSIONA[illegible]
1640 ACCEPT AT(12,1)VALIDATE([illegible])SIZ
    E(1):OPT2
1650 IF OPT1=1 THEN G=2 :: E=20000 :: T
    OFF=65000 :: GOTO 1680
1660 IF OPT1=2 THEN G=4 :: E=45000 :: T
    OFF=230000 :: GOTO 1680
```

```
1670 IF OPT1=3 THEN G=6 :: E=80000 :: T
     OFF=540000
1680 IF TRIP>1 AND OPT1=3 THEN TOFF=250
     000
1690 IF OPT2=1 THEN RESTORE 1760 :: GOT
     O 1730
1700 IF OPT2=2 THEN RESTORE 1970 :: GOT
     O 1730
1710 IF OPT2=3 THEN RESTORE 470 :: GOTO
     1730
1720 IF OPT2=4 THEN RESTORE 2180
1730 CALL CLEAR :: CALL COLOR(8,10,12)
1740 FOR TER=1 TO 21 :: READ TERNS :: D
     ISPLAY AT(TER,1):TERNS :: NEXT TER
1750 GOSUB 690 :: RETURN
1760 DATA
1770 DATA
1780 DATA
1790 DATA
1800 DATA
1810 DATA
1820 DATA
1830 DATA
1840 DATA
1850 DATA
1860 DATA
1870 DATA
1880 DATA
1890 DATA
1900 DATA
1910 DATA
1920 DATA
1930 DATA
1940 DATA
1950 DATA
1960 DATA
1970 DATA
1980 DATA
1990 DATA
2000 DATA
2010 DATA
2020 DATA
2030 DATA
2040 DATA
2050 DATA
2060 DATA
2070 DATA
2080 DATA
2090 DATA
2100 DATA
2110 DATA
2120 DATA
2130 DATA
2140 DATA
2150 DATA
2160 DATA
2170 DATA
2180 DATA
2190 DATA
2200 DATA
2210 DATA
2220 DATA
2230 DATA
2240 DATA
2250 DATA
2260 DATA
2270 DATA
2280 DATA
2290 DATA
2300 DATA
2310 DATA
2320 DATA
2330 DATA
2340 DATA
2350 DATA
2360 DATA
2370 DATA
2380 DATA
2390 DISPLAY AT(11,2):"
2400 DISPLAY AT(12,2):"  INTERPLANETARY
     RESCUE"
2410 DISPLAY AT(15,2):"
2420 CALL SOUND(-4250,110,7,220,7,110,7
     ,-5,7)
2430 DISPLAY AT(24,1):"PRESS ANY KEY TO
     BEGIN"
2440 CALL SPRITE(#2,98,2,188,125,-3,0,#
     3,104,9,138,125,-3,0)
2450 CALL KEY(0,KEY,STAT)::  IF STAT=0 T
     HEN CALL SOUND(-1000,110,7,220,7,1
     10,7,-5,7):: GOTO 2450
2460 CALL DELSPRITE(#2,#3)::  RETURN
```

EXTENDED BASIC



N-VADER

N-Vader is a game for one or two players written in Extended BASIC. Each player controls a "defense ship" whose mission is to prevent alien creatures from reaching Earth. The game is played using either the keyboard or joysticks. If joysticks are used with the TI-99/4A, be sure to put the ALPHA LOCK key in the *up* position *after* setting the parameters of game play in response to the screen prompts.

Aliens are destroyed by positioning the defense ship in the immediate vicinity of the alien. No fire button or key is needed. Every time an alien is destroyed, the player scores a point and another alien is introduced at the top of the screen. Whenever an alien reaches Earth (bottom of the screen), the aliens score.

One unusual feature of this game is its flexibility. When the game starts, the player(s) can select the number of aliens, their speed, the speed of the defense ship and the defense range. Defense range defines the proximity necessary for a defense ship to destroy an alien.

Features of Extended BASIC Used in *N-Vader*

Several Extended BASIC features are used to make *N-Vader* work. Sprites are, of course, fundamental to the program. CALL DISTANCE is used to determine the proximity of alien and defensive ship(s). CALL COINC is used to determine when aliens reach Earth.

Because sprites move independently of the program, alien destruction is sometimes delayed or missed altogether. Aliens can also descend through the Earth for the same reason. Fortunately, these quirks of sprites actually make the game more enjoyable. For example, it is sometimes possible for a defense ship to swoop-down *into* the Earth and pick off an alien at the last possible instant!

A subprogram (lines 1390-1510) is used to allow keyboard input to be processed by the main program as joystick input. Programmers with diskettes may want to save this subprogram in a MERGE file for inclusion in other programs.

EXPLANATION OF THE PROGRAM
N-Vader

Line Nos.	
100-160	Program header.
170	Define invader character.
180-270	Display title screen with sprites.
280-470	Instructions.
480-700	Get parameters.
710-790	Draw Earth.
800-820	Draw invader sprites.
830-840	Draw player sprites. Note that positioning changes for Player #1 depending upon number of players.
850-890	Check for player scoring.
900	Check for invaders at Earth.
910-940	Adjust player motion.
950-1140	Process end of game.
1150-1170	Subroutine to introduce new invader during game.
1180-1380	Subprogram to [illegible]

```
100 !  99'ER VERSION 1.6.2XB
110 !
120 !  N-VADER
130 !
140 !  BY JIM PETERSON
150 REM 99'ER VERSION 1.6.2XB
160 REM
170 CALL CHAR(104,"FF99FFFFA5A5A5A5")
180 CALL CLEAR
```

```
150 FOR A=1 TO 20
200 FOR Y=1 TO 19 STEP 6
210 DISPLAY AT(X,Y):SIZE(B)
220 NEXT Y
230 NEXT X
240 CALL SPRITE(#1,104,11,1,1,0,20)
250 DISPLAY AT(22,2):"PRESS ENTER TO P
LAY"
```

```
260 CALL SPRITE(#2,104,13,65,256,0,-20
    )
270 CALL SPRITE(#3,104,9,73,1,0,100)
280 CALL SPRITE(#4,104,4,81,256,0,-99)
290 ACCEPT AT(22,22)BEEP:X$
300 CALL DELSPRITE(ALL)
310 INPUT "INSTRUCTIONS (Y/N)?":X$
320 IF X$<>"Y" THEN 490
330 CALL CLEAR
340 PRINT "ALIEN CREATURES ARE":"ATTAC
    KING THE EARTH!"
350 PRINT "YOU COMMAND THE ONLY DEFENS
    ESHIPS. ONBOARD COMPUTERS CONTR
    OL THE LASERS WHICH CANDESTROY THE
     INVADERS."
360 PRINT "THE INVADERS WILL NOT ATTAC
    KYOU, ONLY THE EARTH."
370 CALL SPRITE(#1,104,11,1,1,0,20)
380 PRINT
390 INPUT "HIT ENTER WHEN READY":X$
400 CALL CLEAR
410 PRINT "IN THIS GAME, YOU CONTROL
    THE NUMBER OF INVADERS, THEIR
    SPEED, YOUR SPEED & THE LASER R
    ANGE OF YOUR SHIPS."
420 PRINT "SUGGESTED VALUES ARE:
    INVADERS=6,SPEED=6":"YOUR SPEED=3
    ,RANGE=25."
430 PRINT "A SMALLER RANGE MAKES THE
    GAME HARDER."
440 PRINT "YOU ALSO CONTROL THE LENGTH
    OF THE GAME. WHEN ASKED 'END
    OF GAME', ENTER THE"
450 PRINT "NUMBER OF TIMES THE ALIENS
    HIT EARTH FOR THE GAME TO BEOVER.
    "
460 PRINT
470 INPUT "ENTER WHEN READY":X$
480 CALL DELSPRITE(#1)
490 HIT,ZAP1,ZAP2=0
500 INPUT "NUMBER OF PLAYERS? ":NP
510 IF NP<0 OR NP>2 THEN 500
520 NP=INT(NP)
530 INPUT "PLAYER 1 NAME? ":P1$
540 IF NP=1 THEN 560
550 INPUT "PLAYER 2 NAME? ":P2$
560 PRINT "NUMBER OF INVADERS?"
570 INPUT INV
580 IF INV<1 OR INV>8 THEN 560
590 PRINT "INVADER SPEED?"
600 INPUT IS
610 IF IS<1 THEN 590
620 PRINT "DEFENDER SPEED(1-9)?"
630 INPUT SPD
640 IF SPD<=0 THEN 620
650 PRINT "DEFENSE RANGE?"
660 INPUT RNG
670 IF RNG<1 OR RNG>200 THEN 650
680 PRINT "END OF GAME?"
690 INPUT WIN
700 INPUT "JOYSTICKS (Y/N)?":XS
710 IF SEG$(XS,1,1)="Y" THEN JS=1
720 IF WIN<1 THEN 680
730 CALL CHAR(100,"FFFFFFFFFFFFFFFF")
740 CALL CHAR(96,"0008081C7F1C0808")
750 CALL SCREEN(2)
760 CALL CLEAR
770 CALL COLOR(9,16,16)
780 CALL COLOR(3,2,3)
790 CALL COLOR(4,2,3)
800 FOR X=22 TO 24
810 CALL HCHAR(X,1,100,32)
820 NEXT X
830 FOR X=1 TO INV
840 CALL SPRITE(#X,104,3+X,1,INT(RND*2
    56)+1,INT(RND*IS)+1,INT(RND*IS)-IS
    /2)
850 NEXT X
```

```
860 IF NP=1 THEN CALL SPRITE(#9,90,16,
    100,128)ELSE CALL SPRITE(#9,90,16,
    100,56)
870 IF NP=2 THEN CALL SPRITE(#10,96,15
    ,100,200)
880 FOR X=1 TO INV
890 CALL COINC(#X,#9,RNG,B)
900 IF NP=2 THEN CALL COINC(#X,#10,RNG
    ,B2)
910 IF B>=0 AND B2>=0 THEN 1020
920 CALL PATTERN(#X,100)
930 CALL SOUND(-500,-3,0)
940 GOSUB 1360
950 IF B>=0 THEN 980
960 ZAP1=ZAP1+1
970 DISPLAY AT(23,3)SIZE(4):ZAP1
980 IF B2>=0 THEN 1080
990 ZAP2=ZAP2+1
1000 DISPLAY AT(23,23)SIZE(4):ZAP2
1010 GOTO 1080
1020 CALL POSITION(#X,V(X),H(X))
1030 IF V(X)<158 THEN 1080
1040 GOSUB 1360
1050 CALL SOUND(-50,-2,0)
1060 HIT=HIT+1
1070 DISPLAY AT(23,14)SIZE(4):HIT
1080 IF JS=1 THEN CALL JOYST(1,JX,JY)EL
    SE CALL KEYST(1,JX,JY)
1090 CALL MOTION(#9,-JY*SPD,JX*SPD)
1100 IF NP=1 THEN 1130
1110 IF JS=1 THEN CALL JOYST(2,JX,JY)EL
    SE CALL KEYST(2,JX,JY)
1120 CALL MOTION(#10,-JY*SPD,JX*SPD)
1130 IF HIT<WIN THEN 1340
1140 CALL DELSPRITE(ALL)
1150 CALL SCREEN(16)
1160 CALL CLEAR
1170 CALL COLOR(3,2,1)
1180 CALL COLOR(4,2,1)
1190 CALL SPRITE(#1,104,7,1,1,0,25)
1200 PRINT "GAME OVER"
1210 PRINT "EARTH HITS";TAB(18);HIT
1220 PRINT P1$;" DESTOYED";ZAP1;" ALIEN
    S"
1230 PRINT TAB(10);INT(100*ZAP1/(HIT+ZA
    P1+ZAP2));" PER CENT"
1240 IF NP=2 THEN PRINT P2$;" DESTROYED
    ";ZAP2;" ALIENS"
1250 IF NP=2 THEN PRINT TAB(10);INT(100
    *ZAP2/(HIT+ZAP1+ZAP2));" PER CENT"
1260 FOR X=1 TO 100
1270 CALL SOUND(50,440,0)
1280 CALL SOUND(99,880,0)
1290 NEXT X
1300 CALL DELSPRITE(#1)
1310 CALL SCREEN(3)
1320 IS=0
1330 GOTO 170
1340 NEXT X
1350 GOTO 880
1360 CALL DELSPRITE(#X)
1370 CALL SPRITE(#X,104,15-X,1,INT(RND*
    256)+1,INT(RND*IS)+1,INT(RND*IS)-I
    S/2)
1380 RETURN
1390 SUB KEYST(N,X,Y)
1400 CALL KEY(N,K,S)
1410 IF S=0 THEN X,Y=0 :: SUBEXIT
1420 IF K=2 THEN X=-4 :: Y=0
1430 IF K=4 THEN X=-4 :: Y=4
1440 IF K=5 THEN X=0 :: Y=4
1450 IF K=6 THEN X=4 :: Y=4
1460 IF K=12 THEN X,Y=4
1470 IF K=3 THEN X=4 :: Y=0
1480 IF K=14 THEN X=4 :: Y=-4
1490 IF K=0 THEN X=0 :: Y=-4
1500 IF K=15 THEN X=-4 :: Y=-4
1510 SUBEND
```

EXTENDED BASIC

SPACE PATROL

The Earth is at war! Another planet is trying to gain control of our solar system. You are the captain of a patrol ship armed with high-powered lasers. Your mission—destroy a fleet of 15 enemy supply ships en route to their Battle Star. But be careful, because the supply ships are armed with "killer satellites." When launched, the satellites will move in on your ship and self-destruct unless you destroy them first.

Your ship has a supply of 400 energy units, and energy is depleted by 10 units each time you fire your lasers. You also have a deflector shield that is automatically activated when a "killer" gets past your lasers and explodes near you. This will deplete your energy by 50 units. Your on-board computer will warn you if a "killer" has been launched.

At the start of the game, your gun sight will appear in the center of the screen. You may use a joystick or the arrow keys to position this on your target (depending on the option chosen at the start of the game). Then press either the FIRE button or the Y key to fire your lasers.

GOOD LUCK AND GOOD SHOOTING, CAPTAIN!!

Note: If using joysticks with the TI-99/4A, release the ALPHA LOCK key.

EXPLANATION OF THE PROGRAM
Space Patrol

Line Nos.	
150	Clears screen and makes it black.
160	Sets colors of letters and numbers to white.
170-300	Display title and define characters.
310-330	Clears screen; lets user choose joysticks or keyboard.
340-390	Clears screen; initializes energy and ships destroyed; randomizes; lets user choose high or low skill level; sets magnification 3 for high level, 4 for low level.
400-410	Sets colors for stars; randomly places 40 stars on screen.
420	Creates gun sight in center of screen.
430	Displays energy level and number of ships destroyed at bottom of screen.
440	Randomly selects number from 2 to 6; if the number is 5 or 6, branches to the satellite routine; otherwise a supply ship is defined.
450	Changes colors of stars so they will twinkle.
460-470	Randomly sets speed, direction, and location of supply ship.
480	Branches to joystick or keyboard input to move gun sight.
490-530	Checks if fire button or key is pressed; if so, stops motion of gun sight, makes laser display and sound, checks for a hit, and decreases energy.
540-590	If ship is hit makes red explosion and sound, increments ships destroyed, reduces energy, deletes sprite #3. Checks for end of game and branches.
600-610	Sounds and prints warning for satellite launching.
620-680	Creates satellite sprite and gradually increases the size.
690-720	Moves scope if user indicates or tests for hit if fire button or key is pressed. Energy is reduced by 10 for each shot fired.
730-760	If satellite is hit, it explodes; if not there is a larger blast and energy level is reduced by 50.
770-850	Sounds and messages for end of game depending on number of ships or energy level.
860-880	Displays option to play again.
890	Subroutine to move gun sight with joystick.
900-950	Subroutine to move gun sight with arrow keys.

```
100 REM ****************
110 REM * SPACE PATROL *
120 REM ****************
130 REM
140 REM
150 CALL CLEAR :: CALL SCREEN(2)
160 FOR X=0 TO 8 :: CALL COLOR(X,16,1)
    :: NEXT X
170 DISPLAY AT(9,4)BEEP:"*** SPACE PAT
    ROL ***"
180 CALL CHAR(112,"0000000000000103010
    0000000000000000000000000080C080000
    00000000000")!SHOT
190 CALL CHAR(116,"0200920042188015000
    1
    2802004408002008440021080024810024
    0094011080")!EXP
200 CALL CHAR(104,"1500400040004000400
    0400040001500540001000100010001000
    10001005400")!GUN SIGHT
210 CALL CHAR(103,"00000018")::CALL C
    HAR(111,"0000001")::CALL CHAR(95,
    "000000101")!STARS
220 TR$(1)="0000000003848888F8F8848380
    0
    000000000000000181412F1F214181000000
    0000"
230 TR$(2)="01414141417F717F7F477F40404
    0
    40000080828282FE8EFEFEE2FE02020202
    0000"
240 TR$(3)="00000E0E0E0E0E0E3F7FCECE7F
    3F1F0000007070707070 70FCFE7373FEFC
    F800"
250 TR$(4)="01010102 07E1919F9F90F1607F
    000000808080 40E08F91F1F1118F06FE00
    0000"
```

```
260 CALL CHAR(124,"0000000000000201010200000000000000000000000004080804000000000000000")!K1
270 CALL CHAR(128,"0000040D03060703 0D04000000000000000020B0C0E060C0B020000000000000000")!K2
280 CALL CHAR(132,"000C0D03070E07030D0C0000000000000098D8E079B870E0D896000000000000")!K3
290 CALL CHAR(136,"0060611 30F0F1E3CAE1D0F0F136160000000686C8F0F0B87C3C78F0F0C8860600")!K4
300 CALL CHAR(140,"C0C1271F1F3A3D7A7A3D321F1F27C1C00383E4F85CBC5E5EAC5CF8F8E48303")!K5
310 CALL CLEAR
320 DISPLAY AT(10,1):"CHOOSE METHOD OF INPUT:": :"   1  JOYSTICK": :"   2  KEYBOARD--ARROW KEYS": :"AND 'Y' TO FIRE"
330 CALL KEY(0,KEY,STAT):: IF (KEY<49)+(KEY>50)=-1 THEN 330 ELSE INP=KEY-48
340 IF INP<>1 THEN 360
350 DISPLAY AT(20,1):"REMEMBER--ALPHA LOCK UP FOR JOYSTICKS!" :: FOR TD=1 TO 700 :: NEXT TD :: CALL CLEAR
360 CALL CLEAR :: LB=400 :: SD=0 :: RANDOMIZE
370 DISPLAY AT(10,1):"CHOOSE SKILL LEVEL:": :"   1  FOR HIGH": :"   2  FOR LOW"
380 CALL KEY(0,KEY,STAT):: IF (KEY<49)+(KEY>50)=-1 THEN 380 ELSE SK=KEY-48
390 CALL MAGNIFY(SK+2):: CALL CLEAR
400 CALL COLOR(8,16,1,9,11,1,10,3,1)
410 FOR X=1 TO 40 :: CALL HCHAR(INT(23*RND+1),INT(26*RND+3),INT(3*RND+1)*8+87):: NEXT X
420 CALL SPRITE(#1,104,16,98,130)
430 DISPLAY AT(23,1):"SHIPS DESTROYED=";SD:"ENERGY UNITS REMAINING=";LB
440 TAR=INT(6*RND)+1 :: IF TAR>4 THEN 600 :: CALL CHAR(120,TR$(TAR))
450 FOR L=1 TO 4 :: CALL COLOR(INT(2*RND+9),INT(11*RND+3),1):: NEXT L
460 VM=INT(RND*19)-9 :: HM=INT(38*RND)-19 :: IF VM=0 OR HM=0 THEN 460
470 CALL SPRITE(#3,120,INT(14*RND+3),210,INT(256*RND)+1,VM,HM)
480 ON INP GOSUB 890,900
490 CALL KEY(2,K,S):: IF K<>18 THEN 480 :: CALL MOTION(#1,0,0)
500 CALL SOUND(300,1800,2,2300,2,2800,2):: CALL COINC(ALL,HIT)
510 CALL PATTERN(#1,112):: CALL COLOR(#1,7)
520 IF HIT THEN 540 :: LB=LB-10
530 CALL COLOR(#1,16):: CALL PATTERN(#1,104):: DISPLAY AT(24,25):LB :: IF LB<10 THEN 800 ELSE 480
540 CALL PATTERN(#3,116):: CALL COLOR(#3,7,#1,1)
550 SD=SD+1 :: LB=LB-10
560 CALL SOUND(200,-7,5):: CALL SOUND(700,-7,2)
570 FOR L=1 TO 80 :: NEXT L :: CALL DELSPRITE(#3)
580 DISPLAY AT(23,17):SD :: DISPLAY AT(24,25):LB :: CALL COLOR(#1,16):: IF SD>14 THEN 830 :: IF LB<10 THEN 800
590 CALL PATTERN(#1,104):: GOTO 440
600 FOR L=1 TO 4 :: DISPLAY AT(1,9):"**-WARNING-**" :: CALL SOUND(100,600,0,700,0,-2,0):: FOR D=1 TO 30 :: NEXT D
610 DISPLAY AT(1,9):"             " :: FOR D=1 TO 20 :: NEXT D :: NEXT L
620 CALL SPRITE(#3,124,4,INT(80*RND+20),INT(100*RND+80),2,2)
630 FOR ATT=1 TO 5 :: CALL PATTERN(#3,ATT*4+120)
640 FOR D=1 TO 4
650 ON INP GOSUB 890,900
660 CALL KEY(2,K,S):: IF K=18 THEN 690
670 NEXT D
680 GOTO 720
690 CALL MOTION(#1,0,0):: CALL SOUND(300,1800,2,2300,2,2800,2):: CALL COINC(#1,#3,SK*5,HIT)
700 CALL PATTERN(#1,112):: CALL COLOR(#1,7):: IF HIT THEN 760 :: LB=LB-10
710 CALL COLOR(#1,16):: CALL PATTERN(#1,104):: DISPLAY AT(24,25):LB :: IF LB<10 THEN 800
720 NEXT ATT
730 CALL DELSPRITE(#1):: CALL MAGNIFY(4):: CALL PATTERN(#3,116)
740 CALL SOUND(1700,-7,0):: FOR D=1 TO 30 :: CALL SCREEN(7):: CALL SCREEN(10):: NEXT D :: CALL SCREEN(2)
750 CALL DELSPRITE(#3):: CALL MAGNIFY(SK+2):: LB=LB-50 :: DISPLAY AT(24,25):LB :: IF LB<=0 THEN 770 ELSE 420
760 CALL PATTERN(#3,116):: CALL COLOR(#3,7,#1,1):: LB=LB-10 :: GOTO 560
770 CALL DELSPRITE(#1,#3):: CALL SOUND(1000,110,0,130,0,150,0)
780 DISPLAY AT(8,1):"YOUR SHIP HAS BEEN DAMAGED": :"BEYOND REPAIR. BECAUSE OF": :"YOUR FAILURE YOU HAVE BEEN"
790 DISPLAY AT(14,1):"DEMOTED TO PRIVATE!!!" :: GOTO 860
800 CALL DELSPRITE(#1,#3):: FOR L=1 TO 6 :: CALL SOUND(60,2000,0):: NEXT L
810 DISPLAY AT(8,1):"YOUR POOR SHOOTING HAS": :"CAUSED YOU TO RUN OUT OF": :"ENERGY UNITS. RETURN TO BASE"
820 DISPLAY AT(14,1):"AT ONCE!! YOU WILL BE SENT": :"BACK TO THE TRAINING CENTER!" :: GOTO 860
830 CALL DELSPRITE(#1):: FOR L=2000 TO 4500 STEP 250 :: CALL SOUND(100,L,0):: NEXT L
840 DISPLAY AT(8,1):"YOU GOT 'EM ALL CAPTAIN!!!": :"YOUR MISSION WAS A SUCCESS.": :"YOU ARE HEREBY PROMOTED"
850 DISPLAY AT(14,1):"TO FLEET COMMANDER!!"
860 FOR D=1 TO 700 :: NEXT D :: DISPLAY AT(19,1)BEEP:"PLAY AGAIN? (Y OR N)"
870 CALL KEY(0,KEY,STAT):: IF KEY=89 OR KEY=121 THEN 310
880 IF KEY<>78 AND KEY<>110 THEN 870 ELSE CALL CLEAR :: END
890 CALL JOYST(2,C,R):: CALL MOTION(#1,-R*4,C*4):: RETURN
900 CALL KEY(0,KEY,STAT):: IF STAT=0 THEN CALL MOTION(#1,0,0):: RETURN
910 IF KEY=69 THEN 920 ELSE IF KEY=68 THEN 930 ELSE IF KEY=88 THEN 940 ELSE IF KEY=83 THEN 950 :: RETURN
920 CALL MOTION(#1,-16,0):: RETURN
930 CALL MOTION(#1,0,16):: RETURN
940 CALL MOTION(#1,16,0):: RETURN
950 CALL MOTION(#1,0,-16):: RETURN
```

Computer Chess

The game of chess has fascinated men and women for hundreds of years. People from all walks of life and all ages have enjoyed the challenges and entertainment it provides. The universal popularity of chess is undoubtedly due to its resemblance to life: Mastery of chess requires many of the same elements necessary to mastery of one's life—logical thought, long-range planning, the ability to recognize and act on sudden opportunities, persistence, patience, concentration, steady nerves, confidence, objectivity and, of course, lots of experience! Yes, to do well in chess does require all these things, but interestingly enough, practicing the game greatly helps develop and nourish these *same* characteristics and abilities! "Learning to play" is, in reality, one and the same as "playing to learn."

And yet, for all its challenges and self-improvement attributes, the game is enjoyable at all levels of skill—from raw novice to international master. Over the years, chess has provided me with hundreds of hours of engrossing entertainment and many cherished friendships.

With the advent of strong chess-playing computer programs, chess has entered an important new stage of development. People can learn chess much more rapidly than before *without* the often deflating experience of losing many games *in public*. This is especially true for children; losing badly to adults or other children can often drive them from the game. Having a ready and discreet opponent does indeed have its advantages. . .

The TI *Video Chess* Command Cartridge is one of the programs now available. It is a unique implementation since it is contained in 30K of ROM (no time-consuming cassette loading), can run on a "bare-bones" TI-99/4A (no disk drives or other peripherals are needed), uses a keyboard overlay to simplify commands, and has built-in chess clocks. This last feature is useful for users who wish to eventually play in tournaments where the use of chess clocks is mandatory. Playing under tournament conditions is now possible in the privacy on one's own home!

In this initial section, I'll look briefly at the main features [illegible] some suggestions for using the [illegible] program. You can (1) play chess against the computer, (2) play against another human opponent, (3) set up a problem for the computer to solve, or (4) have the program play as many as nine (!) opponents simultaneously. In addition, games or positions may be stored on cassette—an especially useful feature for postal players, or players without enough time to finish their games in one sitting.

When playing against the computer, you can control the playing characteristics of the program by choosing the experience level (beginner, novice, or intermediate), the time allotted to the computer for each move (30 seconds to 200 seconds), and the style of play (normal, defensive, or aggressive). The program also allows you to take back a move, ask for advice, have your move evaluated, or even switch sides!

In the problem mode, you can ask the program to solve a checkmate in two, three, or four moves. This is, of course, a potentially valuable learning tool, but the program's versatility doesn't stop there: You can also set up *any* position and have the computer play a normal game starting from the given position.

Based on many years of tournament experience, I would estimate the maximum strength of the program to be slightly less than the average player in a typical chess tournament. This is superior to probably 90 percent of the world's chess players! And presumably, stronger versions of the program will be available in the future. To put this in perspective, the strongest chess-playing program in the world, running on the enormous and fast CYBER or CRAY computers, still does not play at the level of a human chess master. (It will, however, defeat 99 percent of the world's chess players!)

As an educational tool, the *Video Chess* program is excellent. A beginning player can make rapid improvements in his game by adjusting the strength of the program as his own playing strength increases. If you're a new player, you should have at least one good book on chess that is designed for beginners. (There are many good ones on the market.) Then as you learn new ideas and techniques from reading, you can try them out against the computer immediately. For example, it is important for every player to master the basic checkmates: king and queen vs. king; rook and king vs. king; two bishops and king vs. king. All good chess manuals discuss these in detail. After reading about how to mate with [illegible] immediately try it out using the program [illegible] progress much faster when a [illegible] ready to play!

The weakest part of the program is in the problem mode when asking for a mate in two, three, or four moves. For example, when I gave the program Problem No. 1A (below), it worked for two and one half hours without coming to a conclusion. I finally turned it off. And I have had similar

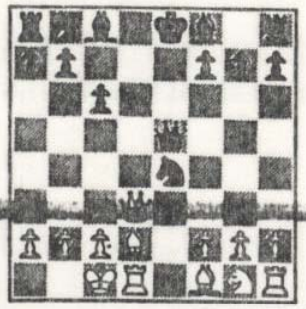
disappointing results with rather easy mates in Problem No. 1B. Fortunately, this defect is not terribly important, and may be alleviated in future versions of the program. The best use for this problem-solving mode seems to be in setting up positions from which the computer will commence playing as in a normal game. (Note: You can do this to learn the basic checkmates mentioned previously.)

Problem No. 1A

White: Pawns: A2, B2, C2, F2, G2, H2
Knights: G1
Bishops: D2, F1
Rooks: H1
Queen: D3
King: C1

Black: Pawns: A7, B7, C6, G7, H7
Knights: B8, E4
Bishops: C8, F7
Rooks: A8, H8
Queen: E5
King: E1

White to move and checkmate in three moves.



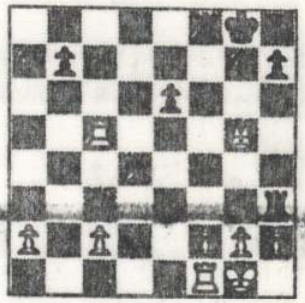
Finally, I will leave you with two problems to solve. Problem No. 1A comes from a game between two grandmasters about sixty years ago. Problem No. 1B is a famous position—especially memorable because after Black made the beautiful winning move, spectators showered the playing stage with gold coins. As it turned out, however they were *not* showing their admiration. . .but rather, paying off their bets *in disgust!*

Problem No. 1B

White: Pawns A2, C2, F2, G2, H2
Knights: None
Bishops: None
Rooks: C5, F1
Queen: G5
King: G1

Black: Pawns: A7, B7, E6, G7, H7
Knights: D4
Bishops: None
Rooks: H3, F8
Queen: C3
King: G8

Black to move and win
(Black has a single crushing move).



Computer Chess PART TWO

Ever wondered where your computer got the "intelligence" to beat you in a game of chess? It's all in the program, you say? But then where did chess-playing computer programs come from? You might suppose that the impetus for the development of these programs came from chess players themselves. But in fact, this was not the case at all. It was researchers in the field of artificial intelligence (psychologists and computer scientists) whom we have to thank for those embarrassing checkmates. . .

The goal of these reseachers was to determine the nature of intelligence itself: What precisely it was, and consequently, what it was not. This was no easy task. They hoped to shed some light on this problem by getting computers to do things that if performed by a human would require "intelligence." It didn't take long to figure out that chess was a natural: It presumably required highly intelligent behavior, and yet, it was "contained" enough so that initial programs designed just to play "legal" games would not be prohibitively large. As these programs were developed, it soon became obvious that to progress from legal games to good —or even just reasonable—play required close attention to basic theory and concepts as understood by humans. For example, the number of possible positions after only the first ten moves in a game is a number having over a *hundred* zeros in it! Hence, looking at all possible positions is clearly impossible.

As a consequence of this need for a higher level of understanding of the game, strong chess players had to be consulted. One of these was international master David Levy of Scotland. Levy is perhaps best known for his $10,000 bet (made in August 1968) that even within a decade, there still wouldn't be a computer program that could defeat him in a match. In the years since his bet (which he won easily), Levy has been a frequent visitor at computer conferences, where he lectures and plays simultaneous exhibitions against several of the current programs. Incidentally, he also acted as a consultant to Texas Instruments in the development of the *Video Chess* program.

Levy has therefore provided a valuable link between the artificial intelligence commmunity and the large community of chess players. He, perhaps more than anyone else, has been in the position to measure the rate of computer chess progress. In his view (and mine as well), the rather recent advent of microprocessor chess playing machines will make chess popular and accessible as never before. The revolution has just begun!

As indicated above, chess playing programs do not attempt to find a move by searching all posssible combinations of moves. Rather, chess programs combine chess theory and concepts together with brute force searching techniques to choose a move. Therefore, they are limited by how well the program "understands" chess theory and can "think" like a human player, and by speed and memory considerations. The speed and available memory determine how far ahead the program can be examined and evaluated in a given amount of time. The number of moves the program can look ahead in a given position is called its *search horizon* (Levy's term).

For these reasons, even though they play relatively strong chess, chess playing programs have certain characteristic weaknesses which can often be exploited. For example, a program may sacrifice a bishop or a knight on one side of the board to win a rook (with a knight usually) in a corner on the other side. This will leave the knight trapped after it captures the rook. To any human chess player, it would be

evident that the knight was permanently trapped and would eventually be lost—leaving the player with only a rook (5 units) to show for the loss of two minor pieces (a total of 6 units). However, the computer would merely consider the situation a gain of two units (lose a bishop or knight and gain a rook) as long as the stranded knight could not be captured within the number of moves in its search horizon. The limited search horizon leads to other situations where short term expedients are followed to the detriment of position.

Future improvements in speed will extend the search horizon of chess programs and thereby increase their playing strength even further. In my opinion, without considerable improvement in the longer range strategic capabilities of these programs, they will not be able to reach the level of world-class human players. However, we players in the other 99.9 percent had better watch out!

As an experiment, I recently pitted my *Video Chess* (a TI Command Cartridge) program against the Boris machine with the Morphy cartridge. Boris-Morphy is reputedly the strongest comercially available microprocessor chess playing machine. The match consisted of playing the *Video Chess* program at its highest level (Intermediate, 200 seconds per move) against the Boris-Morphy machine at three different levels from high to low. Although the Boris-Morphy program won all three games, the *Video Chess* program did obtain a winning position against the two lower levels (but could not find the knock-out punch). The top level of Boris-Morphy seems clearly stronger than *Video Chess*. All in all, the results were not bad, and since the top current level of *Video Chess* is called "Intermediate," we may look forward to further strengthening of the program.

The two problems I'll leave you with are from games by famous chess players. The first position is from a game of "speed " chess played in 1912 between American Edward Lasker (who died recently at age 96!) and former English champion Sir George Thomas. The rules were, I believe, that neither player could allow his own clock to get more than five minutes ahead of his opponent's clock. To find such a pretty mating combination at that speed is impressive. The second position was played by the great American champion Harry Nelson Pillsbury near the turn of the century in an exhibition where he played blindfolded against 22 different opponents simultaneously! Blindfold play is not as difficult as you might think—try it against your *Video Chess* program sometime—but to play 22 such games successfully is phenomenal. In recent times George Koltanowski has played blindfolded against more than 50 opponents simultaneously. But Pillsury's achievement is magnified by the fact that he could perform well in blind simultaneous play against *masters*!

Problem No. 2A

White: Pawns: A2, B2, C2, D4, F2, G2, H2
Knights: E4, E5
Bishops: D3
Rooks: A1, H1
Queen: H5
King: E1

Black: Pawns: A7, B6, C7, D7, E6, G7, H7
Knights: B8
Bishops: B7, F6
Rooks: A8, F8
Queen: E7
King: G8

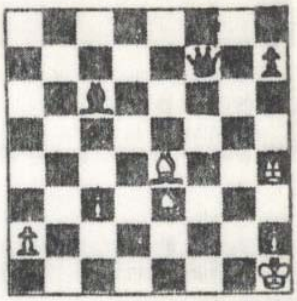
White to move and mate in several moves. Can you find the fewest necessary?

Problem No. 2B

White: Pawns: A2, C3, H2
Knights: None
Bishops: E3, E4
Rooks: None
Queen: H4
King: H1

Black: Pawns A7, B6, C5, H7
Knights: None
Bishops: C6
Rooks: None
Queen: F7
King: H8

Black to move and mate in three moves.

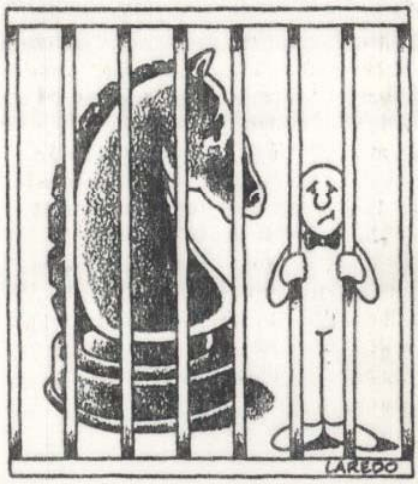


Computer Chess
PART THREE

We have discussed the relationship between chess programs and artificial intelligence, and examined some general characteristics of chess playing programs—both strengths and weaknesses. In this article, I'm going to illustrate some of these characteristics through an actual game played between the TI *Video Chess* program and myself. The game was played with the program set on intermediate level, normal mode, with 200 seconds per move allowed.

White: J. Wolfe	**Black: TI-99/4 with Video Chess**
1. E2 - E4	G7 - G6
2. D2 - D4	F8 - G7

These first two moves constitute the Pirc-Robatsch defense to the opening move E2 - E4. You may have noticed in your play that the program often makes the first several moves quickly and then slows down. This is because certain standard opening sequences are stored



in the program and played automatically in the appropriate situation. As soon as these run out, or as soon as the position is no longer standard, the program reverts to its main programming and hence slows down.

3. B1 - C3	C7 - C6

The main purpose of the opening part of the game is to bring out the pieces and to get a reasonable foothold in the central part of the board. (More precisely, the *center* is the square region whose corners are C3, C6, F6, and F3. The squares D4, D5, E4, and E5 are especially crucial.) Long experience has shown that the success of future maneuvers depends on an adequate control of this area. The last moves for each side fit well into this plan. White brings out a knight that bears down on the center while Black prepares to play D7 - D5 establishing his own foothold there.

4. F1 - C4	B7 - B5
5. C4 - B3	

White develops a piece and temporarily prevents D7 - D5. Black responds by driving back White's bishop and preparing a later pawn advance on the *queen-side* (i.e., the left-hand portion of the board).

5. . . .	D7 - D6

This is a weak move because of the following tactic.

6. C3 - B5

Black cannot capture the knight because White then plays 7. B3 - D5 and captures the rook at A8 coming out with a two unit gain in material. (Recall that a rook is worth 5 units and a bishop 3 units. These units represent the relative strength of the two pieces.)

You might be wondering why the program missed such a short sequence of moves. Well, the reason is fairly complex. The program has two basic features: The first is a *static evaluation feature* which takes a given position and evaluates it to decide which side is better and by how much. This is done by assigning numerical values to certain features of the position and summing these values to get a numerical value for the position. For example, being a pawn ahead in material might be worth, say, 75 points, while not being able to castle (ever) might be worth *minus* 15 points. The program does this for both sides, and the side with the largest score is judged to have the best position. In this evaluation scheme, material advantage is given the largest positive weight by far.

The second basic feature of the program is a *searching procedure*. When combined with the static evaluation program, it allows the program to evaluate the consequences of various moves and to pick what it deduces to be the optimum one. Unfortunately, time and memory considerations limit the number of moves the program can look ahead (i.e., its *search horizon*) and can also limit the number of moves that are considered in response to a contemplated move.

Thus, in examining the position after 5. . . . D7 - D6, White has 38 legal moves. In deciding which moves to consider first as possible replies by White, the program will *not* begin with moves that result in immediate material loss by White. This again is due to the heavy weight assigned to material superiority. Thus the continuation 6. C3 - B5 might not even be reached in the search within the time limit. Sacrifices of material are difficult for all but the most advanced and powerful programs to either make or predict.

6. . . .	D6 - D5

This is a good move and is the other side of the argument above. The program finds the only possible way to regain the lost pawn. Here the emphasis on material is helpful to the program.

7. B5 - C3	D5 - E4
8. C3 - E4	G7 - D4
9. G1 - F3	

Thus, Black has not lost a pawn after all. However, Black's position now has two unpleasant features: First, his pawns at A7 and C6 are weakened since they cannot be protected by pawns if attacked, but must be protected by pieces. This can tie down Black's pieces and will make the pawns vulnerable, especially in the later part of the game when fewer pieces remain. Second, to regain the pawn, Black has exposed his bishop to attack. Thus White can develop a piece (G1 - F3) and at the same time force Black to waste a move either guarding or retreating his bishop. Note that White has three pieces developed and no pawn weaknesses, while Black has only one developed and definite pawn weaknesses. White already has a distinct advantage.

9. . . .	C6 - C5

This is another weak move. Black cannot retreat the bishop to G7 or F6 because of the B3 - F7 check winning the queen, but D4 - B6 is possible—preserving material equality. There is some evidence from this game and others I have played that the search horizon of *Video Chess* is about *two* moves in complicated positions. This would explain why C6 - C5 (so as not to waste a move retreating) was considered best.

10. C2 - C3	D4 - F2 check

Now looking ahead two moves, the program *apparently* can see that if D4 -G7, then B3 - F7 check and White wins the black queen on the next move. However, later when I set up the position after D4 - G7 and asked the program to play White, it played D1 - D8 winning only two pawns (the one on F7 and then the one on C5), leaving White two units ahead. Since giving up a bishop for a pawn also leaves Black two units behind without having to trade queens, the move 10. . . . D4 - F2 was chosen. Thus it appears that the program made the right move for the wrong reason!

11. E1 - F2	G8 - F6

Again the program does not see the third move in the coming sequence.

12. E4 - F6 check	E7 - F6
13. D1 - D8 check	E8 - D8
14. B3 - D5	

Thus White wins another piece.

14. . . .	B8 - C6
15. D5 - C6	A8 - B8

White is so far ahead in material that winning is simple. Accordingly, I will relate the rest of the game with little comment.

16. B2 - B4

This move allows White to play C1 - F4 without an annoying check at the B2 by the black rook.

16.	. . .	C5 - B4
17.	C1 - F4	B8 - B6
18.	A1 - D1 check	D1 - E2
19.	F4 - D6 check	E7 - D8

19. . . . E7 - E6 leads to a quick mate after 20. H1 - E1 check E6 - F5; 21. D1 - D5 check F5 - G4; 22. H2 - H3 mate.

20.	D6 - C5 check	D8 - C7
21.	C5 - B6 check	A7 - B6
22.	C6 - D5	B4 - C3
23.	H1 - E1	C7 - B8
24.	D5 - F7	C8 - G4
25.	E1 - E7	G4 - F3
26.	G2 - F3	F6 - F5
27.	D1 - D7	B8 - C8
28.	F7 - E6	B6 - B5
29.	D7 - A7 check	C8 - D8
30.	E7 - D7 check	D8 - E8
31.	A7 - A8 checkmate	

Currently, the most powerful chess programs can look ahead about *six* moves in fairly complicated positions. Advancements in hardware should extend the capability to *nine* moves. This is about twice as many moves as chess masters can look ahead in complicated positions. Such programs will be virtually impossible to trap in simple tactical sequences and, in fact, the human player will most likely be victim. To defeat such a program will require superior application of chess theory and strategy, as well as avoidance of open tactical situations where an eight or nine move look-ahead program would be at its best.

Computer Chess
PART FOUR

The computer chip has already revolutionized the game and toy industry, and even bigger changes are ahead. Chess playing machines, a specialized branch of this new technology, are now widespread. There are several companies making what are essentially simple microcomputers, completely devoted to playing chess (at least eight companies at last count). This is, of course, in addition to numerous packages of chess software that run on personal computers.

Competition between chess-playing machines has resulted in a continuing strengthening and evolution in performance to the point where there is now a world *micro*computer chess championship held each year. This now complements the annual world computer championship which has been held for several years, and which features powerful programs, typically requiring large, fast computers.

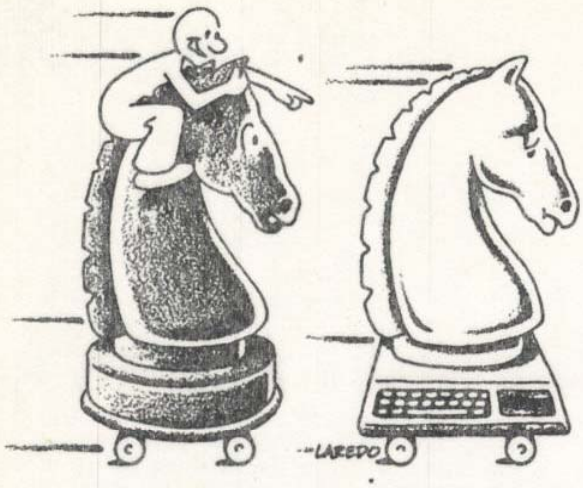
In September 1981, the second annual World Microcomputer Championship was held in Hamburg, Germany. Four machines competed in the commercial division and eight in the experimental group. The Chess Champion Mark V (Sci Scys, Hong Kong) won the commercial group while the Champion Sensory Challenger (Fidelity, U.S.A.) was a close second and defeated the Mark V in their individual series 2½-1½. In the experimental group, Fidelity Experimental (USA) was first, with Princhess (Sweden) second and a two-way tie for third place between Pilidor Experimental (England) and the Phoenix/Novag Experimental (USA/Hong Kong).

Later in November, the twelfth annual North American Computer Championships were held in Los Angeles. Sixteen programs were entered—from microcomputer programs like Philidor mentioned above, to powerful "move crunchers" like Belle of Bell Labs, and Cray Blitz using the powerful Cray computer that carries out 80 million instructions per second! The winner was the impressive program, Belle (also the world computer champion!) which features, in addition to the basic program, special hardware developed expecially for chess. This program and hardware can examine 23 million (!) chess positions in a three minute period—almost *seven times* as many as its nearest competitor. Finishing in a tie for second were Cray Blitz, Nuchess and Bebe. Even though Cray Blitz is backed up by a computer a hundred times faster than Belle's, it can only examine only about a million positions in a three minute period. This demonstrates the great advantage of having special hardware!

Just in case you're curious about how well these programs play chess, I give you here the crucial last round game between Belle and Cray Blitz for the championship.

White: Cray Blitz	Black: Belle
1. E2 - E4	E7 - E5
(Normally gives a lead in development and a powerful piece play.)	
8. D1 - F3	A8 - B8
9. B5 - C6 check	A5 - C6
10. F3 - C6 check	F6 - E7
11. D2 - D3	F8 - E7
12. G5 - E4	C8 - B7
13. C6 - A4	D8 - C7
(13. . . 0 - 0 may be better)	
14. B8 - C3	B7 - C6
A4 - C4 was better for White and would then have the	
19. F2 - F4?	D7 - D5
(19. C1 - E3 was better for White who would then have the advantage.)	
20. D5 - A5	C7 - D3
21. A5 - A7	0 - 0
22. A7 - E7	D3 - E4
23. E7 - E6 check	G8 - H8
24. F4 - E5	F6 - E5
25. F1 - F8 check	B8 - F8
26. B2 - B3	E4 - F3 check
29. B6 - D8 check	F8 - G7
30. D8 - F6 check	G7 - H7
31. D3 - F1	E5 - F4
(D3 - F1 is forced to avoid checkmate.)	
32. A2 - A3	E4 - E3
33. C1 - E3	F1 - A1

and Black won in a few more moves.



Computer Chess
PART FIVE

In this section we are going to look at some variations of standard chess problems, as well as a few interesting challenges associated with chess but not directly related to playing the game. You'll be able to try all of this on your TI-99/4A computer with the *Video Chess* Command Cartridge.

Diversions

By now, of course, you are already well acquainted with chess problems taken from positions in actual games. But chess literature also abounds in problems that have little or no relevance to practical play, but are nevertheless extremely intriguing. Here are a few:

Problem 5A: This is called the "Knight's Tour." Place a knight on an empty board (on A1 for example) and move the knight 63 consecutive times in such a way as to land on each square exactly once and return to the beginning square on the 64th move.

Problem 5B: Remove the squares H1 and A8 from the chessboard. Is the "Knight's Tour" still possible now? You are required to prove that your answer is correct!

Problem 5C: This problem involves a knowledge of chess plus the ability to make logical deductions. While playing a game of chess, Black became irked at his losing position and petulantly removed his king from the board. At that moment, White was in the middle of making his move; for an instant after removal of the black king, the board was completely empty. After White completed his move, Black cooled down and replaced his king. But then he made the worst possible move on the board and White announced mate in two moves. Your task is to reconstruct the position just before White moved and give the exact sequence of moves leading to the checkmate of the black king. (Yes, the problem has a solution.)

Problem 5D: Place eight queens on an otherwise empty board in such a way that no two queens are attacking each other.

Problem 5E: Find the shortest number of moves necessary to produce a *stalemate* starting position.

The above problems represent only a small sample, but perhaps give some idea of the variety of possiblities. Oh yes, I will provide solutions (for all but Problem 5E, for which the minimal number is not known). It is a much smaller number than one would think on first seeing the problem. Try it and see what you can come up with. . .

Versions

Besides the diversions provided by such puzzles, chess players have also been attracted by variations on the basic game of chess. "Speed Chess" (or five-minute chess) is a version requiring a chess clock. Initially each player is given five minutes of time. Play then proceeds until one side is checkmated, a draw is declared, or until one side runs out of time. (For those of you who are not acquainted with the use of a chess clock, I should explain that the player has his clock running until he makes his move. He then pushes a button stopping his own clock and starting that of his opponent.) Thus each game lasts no more than ten minutes. This version is widely popular at chess clubs and among tournament players.

Another currently popular version, especially with younger players, is called "Siamese Chess." This involves four player divided into teams of two players each and requires two chess sets and (usually) two chess clocks. The partners sit on the same side of the table and play opposite colors. Thus the pieces that one partner captures will be the same color as those his partner will be playing on the adjacent board. As one partner captures a piece from his opponent, he passes it to the other partner. The reason for this is that, in addition to the usual moves of chess, one is allowed to place new pieces on the board anywhere that is not occupied—with the one exception that pawns may not be placed on the back ranks (squared A1 - H1 or A8 - H8) where they could be promoted instantly to a more powerful piece.

The placement of new pieces on the board causes the chess battle to take place at an accelerated pace, and causes unusual and often hilarious positions to occur. To make matters worse, the clocks on each board are set for five minutes as in speed chess! The game ends when either a checkmate occurs in one of the games, both games are drawn, or one side runs out of time.

Although chess is far from being "played out," so much study has been devoted to the opening portion of the game that it is possible to go through the first twenty moves in some openings simply repeating moves that are already known to be good. These are called "book moves" because they can be found in chessbooks dealing with openings. This means that a player may obtain a substantial advantage in the opening stages of a game simply by memorizing several sequences of moves found in opening books. At the grandmaster level, this tendency is so refined that victory often hinges on knowing the latest wrinkle in the theory of some particular opening variation, and springing it on a less prepared opponent—one who must then expend extra time on his clock searching for the best reply to this surprise. To combat this over-refinement of opening theory, a simple variation of chess has been proposed. It is called "Prechess" and is played exactly like ordinary chess except for the first eight moves of the game. These proceed as follows: Both sides line up their pawns in the usual way, but leave the row behind the pawns empty. Then the

first eight moves consist of each side alternately (beginning with White, as usual) placing down one piece at a time on the back row anywhere that is unoccupied. This is done until all eight pieces on each side are placed. Then the game continues in the usual way. Since all opening theory is based on the *standard* starting position (which this is usually not), the printed variations found in the opening books are useless.

This version of chess appeals to many serious players and has the advantage that it can be played using a standard set without any bizarre rule changes; basic chess principles still apply as strongly as ever. By using the problem mode of the *Video Chess* program to set up the initial position, you can play "Prechess" on your computers. Try it some time. Some very unusual and interesting games can result from it.

Solutions to Chess Problems

Problem No. 1A:

1. D3 D8 check E8 - D8
2. D2 - G5 double check

(a)2. . . D8 - E8
3. D1 - D8 checkmate

(b)2. . . D8 - C7
3. G5 - D8 checkmate

Problem No. 1B: 1. . . C3 - G3!!

Black appeared to be in trouble since after the apparently forced retreat of his queen out of danger, White could capture the rook on H3 and be decisively ahead in material. Black had foreseen all this, however, and replied with the crushing move above. White has three ways to capture the black queen (which must be captured else mate on H2 is inevitable)—all unsatisfactory.

(a) 2. H2 - G3 D4 - E2 checkmate.
(b) 2. F2 - G3 D4 - E2 check.
3. G1 - H1 F8 - F1 checkmate.

(c) 2. G5 - G3 D4 - E2 check.
3. G1 - H1 E2 - G3 check.
4. H1 - G1 G3 - E2 check.
5. G1 - H1 H3 - C3

and Black is a full piece ahead with an easy win. In the actual game, White resigned after 1. . . C-G3.

Problem No. 2A

1. H5 - H7 check!! G8 - H7
2. E4 - F6 Double check H7 - H6 (else EF - G8 mate)
3. E5 - G4 check
4. F2 - F4 check

(a) 4. . . G5 - F4
5. G2 - G3 check F4 - G5
6. H2 - H4 checkmate or
5. . . F4 - F3
6. 0 - 0 checkmate.

(b) 4. . . G5 - H4
5. G2 - G3 check H4 - H3
6. D3 - F1 check B7 - G2
7. G4 - F2 checkmate

Problem No. 2B

1. . . . F7 - F1 check
2. E3 - G1 F1 - F3 check!
3. E4 - F3 C6 - F3 checkmate.

Problem No. 5A:

Here is a knight's tour beginning at A1. This solution is my own, found after an appropriate amount of trial, error and frustration!

A1 - C2 - E1 - G2 - H4 - F3 - G1 - H3 - F2 - H1 - G3 - F1 - H2 - G4 - H6 - F7 - H8 - G6 - F8 - H7 - F6 - G8 - E7 - F5 - G7 - H5 - F4 - E6 - G5 - E4 - D6 - E8 - C7 - A8 - B6 - C8 - A7 - C6 - D8 - B7 - A5 - C4 - E5 - D7 - B8 - A6 - B4 - A2 - C3 - D5 - E3 - D1 - B2 - A4 - C5 - D3 - C1 - E2 - D4 - B5 - A3 - B1 - D2 - B3 - A1 Home Again!

Problem No. 5B:

Removing the square H1 and A8 makes the knight's tour *impossible*. The reasoning is as follows: Both these squares are the same color (white) so there remain two more black squares than white. But on a knight's tour the color of the squares visited alternates from move to move so that the total number of dark squares and light squares must be the same if a tour is possible.

Problem No. 5C:

It is clear that White must have at least two pieces to checkmate black. The only legal move in which two pieces of the same color may be moved is castling. Thus White was in the act of castling when Black removed his own king and hence White has a king and a rook. The remaining problem is to move the black king so that after White castles it can be moved to a square where checkmate in two can be forced (counting the first Black move). A little experimenting leads to the conclusion that the black king in on B3 and White was castling on the queenside. The final sequence is 1. 0 - 0 - 0 B3 - A2 2. D1 - D3 A2 - A1 3. D3 - A3 checkmate!

Problem No. 5D:

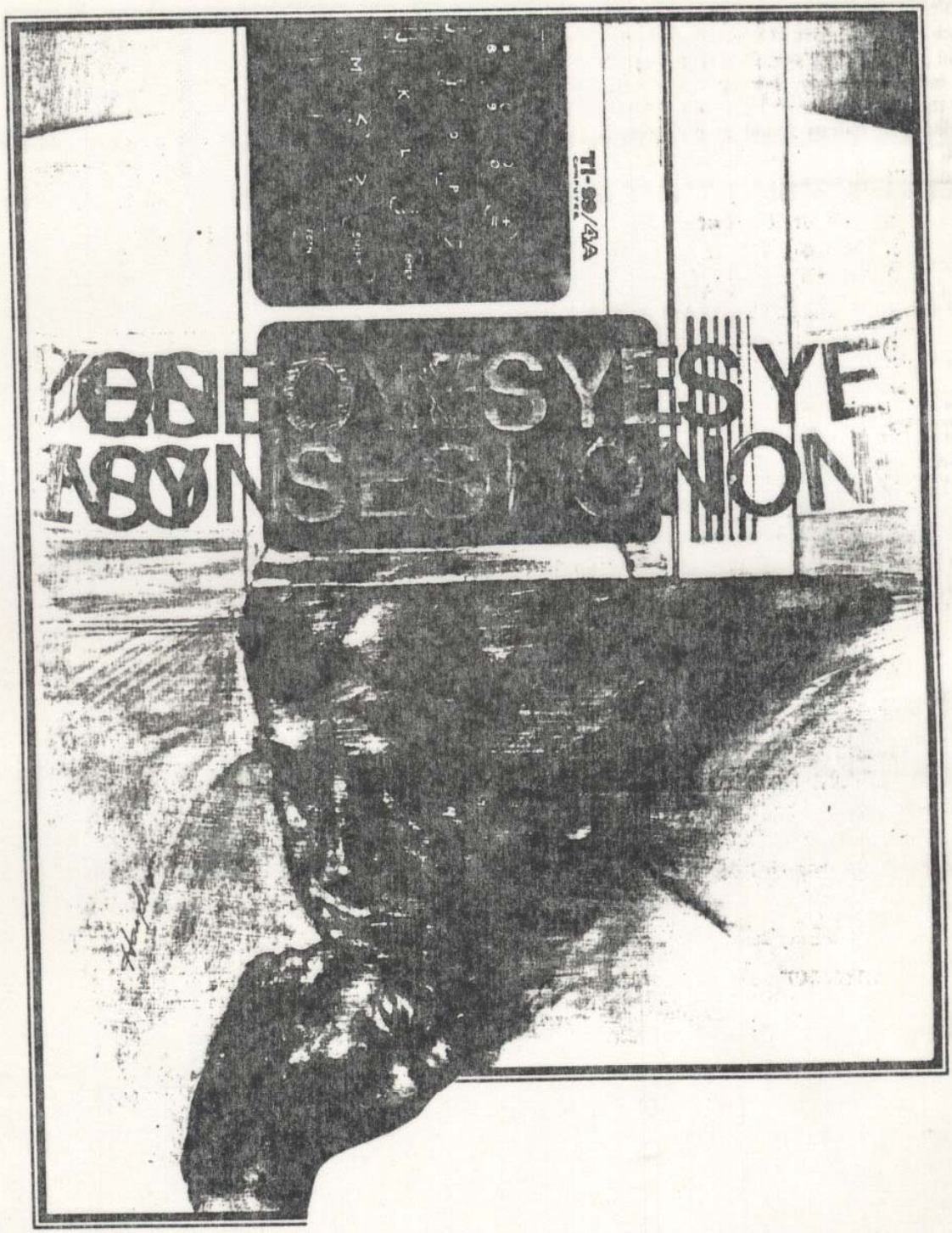
It is easy to convince yourself that this one is impossible but it isn't. One solution is to place the queens on A3, B4, C7, D1, E4, F2, G8, and H6.

8

Applications and Utilities

From bartending, banking, and budget management . . . to big ideas for small businesses.

8
Applications and Utilities



A MICRO BARTENDER

TI BASIC ON THE ROCKS

Entertaining guests can indeed be a chore—especially when you have to help them decide on the choice of drinks, remember how to correctly mix the selected drinks, and simultaneously explain to your curious visitors exactly how you use the exotic computer in your livingroom. Now, this three-part task can be handled much more enjoyably with *Micro Bartender*—a TI BASIC program.

The next time guests arrive just sit them in front of your home computer and let them choose their own mixed drinks. The program will not only provide easy-to-follow recipes, but will also show your guests how the finished drinks should appear—in full color, with proper glass and garnish!

But what's the use of choosing drinks that are impossible to make because you're missing one or more ingrediants? It's definitely slow and frustrating when the only way to find "possible" drinks is by scanning all the ingredients on page after page of recipes. But happily, this tedious process is now a thing of the past. With *Bartender*'s built-in search routine, you can tell the computer what ingredients are actually on hand, and it will tell you what drinks you can, in fact, make. Then, you can look up the details of each recipe and see a graphic representation of the finished drink's appearance.

Cramming nearly a score of drink recipes (plus the associated graphics) into the TI-99/4's 16K of RAM memory was no easy feat. Observant programmers will notice our extensive use of data reconstruction techniques. For those programmers who happen to be non-drinkers—and debugging alone could drive a man to drink—the program logic and control structure is suitable with many other types of reconstructed "recipes." [Only kidding, of course, about "driving a man to drink. . . ."—Ed.]

EXPLANATION OF THE PROGRAM
Micro Bartender

Line Nos.	
200-240	Prints title screen.
250-290	Subroutine to determine color for graphics.
300-1350	Subroutine for graphics.
1360-1650	Defines special characters.
1660-1750	Reads data while title screen is displayed.
1760-1860	Prints screen of two major options.
1870-2220	First option. Prints two menu screens of the list of drinks, receives user's choice.
2230-2250	Clears screen, sets colors of graphics for drink chosen.
2260-2540	Prints name of drink and type of glass.
2550-2580	Prints amounts and ingredients in recipe.
2590-2650	Prints mixing instructions.
2660-2710	Prints cocktail or whiskey sour glass.
2720-2810	Prints garnish and sets colors for garnish.
2820-2850	Prints instructions for stir rod or straws.
2860-3000	Draws the drink.
3010-3020	User may press any key to continue program.
3030-3090	Second option. Prints instructions for ingredient inventory.
3100-3200	Receives user's input Y or N for each ingredient in INV$ array.
3220-3260	Prints message for no drinks possible.
3270-3370	Compares each drink's ingredients with inventory list and prints possible drinks to make.
3380-3400	User presses any key to go back to option screen.
3410-3780	Data for DRINK$ array of attributes for each drink.
3790-3810	Names of ingredients for inventory list.

```
100 REM  ** BARTENDER **
110 REM
120 REM
170 CALL SCREEN(8)
180 CALL CLEAR
190 DIM DRINKS(19,23),INVS(15,1)
200 PRINT "** BARTENDER **":::::::::
210 CALL COLOR(2,7,16)
220 C=7
230 GOSUB 250
240 GOTO 1360
250 CALL COLOR(14,13,C)
260 CALL COLOR(11,2,C)
270 CALL COLOR(16,11,C)
280 CALL COLOR(12,C,C)
290 RETURN
300 REM  NORMAL GLASS
310 CALL HCHAR(4,23,105,5)
320 CALL HCHAR(13,23,104,5)
330 CALL VCHAR(5,22,96,8)
340 CALL VCHAR(5,28,103,8)
350 CALL HCHAR(13,22,106)
360 CALL HCHAR(13,28,107)
370 CALL HCHAR(4,28,108)
380 CALL HCHAR(4,22,109)
390 FOR I=12 TO 7 STEP -1
400 FOR J=27 TO 23 STEP -1
410 CALL HCHAR(I,J,120)
420 NEXT J
430 NEXT I
440 RETURN
450 REM  COCKTAIL GLASS
460 CALL HCHAR(6,21,129)
470 FOR I=1 TO 3
480 CALL HCHAR(6+I,21+I,113)
490 NEXT I
500 FOR I=1 TO 3
510 CALL HCHAR(6+I,20+I,102)
520 NEXT I
530 CALL HCHAR(6,28,128)
540 FOR I=1 TO 3
550 CALL HCHAR(6+I,28-I,112)
560 NEXT I
570 FOR I=1 TO 3
580 CALL HCHAR(6+I,29-I,97)
590 NEXT I
600 CALL HCHAR(10,24,100)
610 CALL HCHAR(10,25,101)
620 CALL VCHAR(11,24,96,3)
630 CALL VCHAR(11,25,103,3)
640 CALL HCHAR(14,23,104,4)
650 CALL HCHAR(5,21,105,8)
660 CALL HCHAR(7,23,120,4)
670 CALL HCHAR(8,24,120,2)
680 RETURN
690 REM  TALL GLASS
700 CALL HCHAR(4,23,105,4)
710 CALL HCHAR(15,23,104,4)
720 CALL VCHAR(5,22,96,10)
730 CALL VCHAR(5,27,103,10)
740 CALL HCHAR(4,22,109)
750 CALL HCHAR(4,27,108)
760 CALL HCHAR(15,22,106)
770 CALL HCHAR(15,27,107)
780 FOR I=14 TO 7 STEP -1
790 FOR J=26 TO 23 STEP -1
800 CALL HCHAR(I,J,120)
810 NEXT J
820 NEXT I
830 RETURN
840 REM  CHAMPAGNE GLASS
850 CALL HCHAR(4,23,105,7)
860 CALL HCHAR(11,23,104,7)
870 CALL VCHAR(5,22,96,6)
880 CALL VCHAR(5,30,103,6)
890 CALL HCHAR(11,22,106)
900 CALL HCHAR(11,30,107)
910 CALL HCHAR(4,30,108)
920 CALL HCHAR(4,22,109)
930 FOR I=10 TO 7 STEP -1
940 FOR J=29 TO 23 STEP -1
950 CALL HCHAR(I,J,120)
960 NEXT J
970 NEXT I
980 RETURN
990 REM  PONY GLASS
1000 CALL VCHAR(5,22,96,5)
1010 CALL VCHAR(5,27,103,5)
1020 CALL HCHAR(9,26,112)
1030 CALL HCHAR(10,25,112)
1040 CALL HCHAR(10,26,97)
1050 CALL HCHAR(9,23,113)
1060 CALL HCHAR(10,24,113)
1070 CALL HCHAR(10,23,102)
1080 CALL HCHAR(11,24,100)
1090 CALL HCHAR(11,25,101)
1100 CALL VCHAR(12,24,96,3)
1110 CALL VCHAR(12,25,103,3)
1120 CALL HCHAR(15,23,104,4)
1130 CALL HCHAR(4,23,105,4)
1140 CALL HCHAR(4,22,109)
1150 CALL HCHAR(4,27,108)
1160 CALL HCHAR(9,24,120,2)
1170 CALL HCHAR(8,23,120,4)
1180 CALL HCHAR(7,23,120,4)
1190 RETURN
1200 REM  SPK OLIVE OR CHERRY
1210 CALL HCHAR(7,24,150)
1220 CALL HCHAR(6,23,98)
1230 RETURN
1240 REM  LEMON TWIST
1250 CALL HCHAR(7,24,152)
1260 CALL HCHAR(7,25,153)
1270 RETURN
1280 REM  ORANGE
1290 CALL COLOR(16,12,8)
1300 CALL HCHAR(5,22,157)
1310 CALL HCHAR(5,23,154)
1320 CALL HCHAR(6,22,155)
1330 CALL HCHAR(6,23,156)
1340 CALL HCHAR(4,21,98)
1350 RETURN
1360 CALL CHAR(152,"7F7F7F7F7F")
1370 CALL CHAR(110,"0303030303030303FFFF")
1380 CALL CHAR(158,"000000000C8CDFFFF")
1390 CALL CHAR(153,"FEFEFEFEFE")
1400 CALL CHAR(112,"0103070F1F3F7FFF")
1410 CALL CHAR(128,"0103070F1F3F7FFF")
1420 CALL CHAR(97,"FFFEFCF8F0E0C080")
1430 CALL CHAR(150,"3C7EFFFFFFFF7E3C")
1440 CALL CHAR(99,"2424242424242424")
1450 CALL CHAR(100,"FFF7F3F1F0F07030")
1460 CALL CHAR(101,"FFFEFCF8F0E0C080")
1470 CALL CHAR(102,"FF7F3F1F0F070301")
1480 CALL CHAR(113,"80C0E0F0F8FCFEFF")
1490 CALL CHAR(129,"80C0E0F0F8FCFEFF")
1500 CALL CHAR(106,"0303030303")
1510 CALL CHAR(107,"C0C0C0C0C0")
1520 CALL CHAR(108,"000000000000C0C0")
1530 CALL CHAR(109,"0000000000000303")
1540 CALL CHAR(104,"FFFFFFFFFFFFFFFF")
1550 CALL CHAR(96,"0303030303030303")
1560 CALL CHAR(103,"C0C0C0C0C0C0C0C0")
1570 CALL CHAR(105,"000000000000FFFF")
1580 GOSUB 460
1590 CALL CHAR(136,"3F7FFFFFCFCFF7F3F")
1600 CALL CHAR(137,"FCFEFFFF3F3FFFFEFC")
1610 CALL CHAR(138,"0000000000000000")
1620 CALL CHAR(139,"0F0F0F0F0F0F0F0F")
1630 CALL CHAR(134,"C0C0C0C0C0C0C0C0")
1640 CALL CHAR(135,"C1C2C4C8D0E0C080")
1650 CALL CHAR(146,"8888888888888888")
1660 FOR I=1 TO 19
1670 CALL COLOR(2,16,7)
1680 FOR J=0 TO 23
1690 READ DRINKS(I,J)
1700 NEXT J
1710 CALL COLOR(2,7,16)
```

```
1720 NEXT I
1730 FOR I=0 TO 15
1740 READ INV$(I,0)
1750 NEXT I
1760 CALL CLEAR
1770 CALL COLOR(2,2,1)
1780 PRINT "DO YOU:":::"(1) WANT TO SEE
     THE RECIPE"
1790 PRINT "  FOR A SPECIFIC DRINK?"
1800 PRINT :::"(2) WANT TO KNOW WHAT YOU
     "
1810 PRINT "  CAN MAKE WITH THE"
1820 PRINT "  INGREDIENTS YOU HAVE?":
     :::::::::
1830 CALL SOUND(150,1397,2)
1840 CALL KEY(0,K,S)
1850 IF (K<49)+(K>50)THEN 1840
1860 ON K-48 GOTO 1870,3030
1870 CALL CLEAR
1880 PRINT "DRINK:":::
1890 PRINT :::"  (1) MARTINI"
1900 PRINT :::"  (2) DRY MARTINI"
1910 PRINT :::"  (3) EXTRA DRY MARTINI"
1920 PRINT :::"  (4) VODKA MARTINI"
1930 PRINT :::"  (5) MANHATTAN"
1940 PRINT :::"  (6) DRY MANHATTAN"
1950 PRINT :::"  (7) SWEET MANHATTAN"
1960 PRINT :::"  (8) PERFECT MANHATTAN"
1970 PRINT :::"  (9) WHISKEY SOUR"
1980 PRINT :::"  (C) CONTINUE"
1990 CALL SOUND(150,1397,2)
2000 CALL KEY(0,K,S)
2010 IF K=67 THEN 2050
2020 IF (K<49)+(K>57)THEN 2000
2030 II=K-48
2040 GOTO 2230
2050 CALL CLEAR
2060 PRINT "DRINK:"
2070 PRINT :::"  (0) WARD EIGHT"
2080 PRINT :::"  (1) DAIQUIRI"
2090 PRINT :::"  (2) BACARDI"
2100 PRINT :::"  (3) SCREWDRIVER"
2110 PRINT :::"  (4) PINK LADY"
2120 PRINT :::"  (5) SALTY DOG"
2130 PRINT :::"  (6) GIN COOLER"
2140 PRINT :::"  (7) TOM COLLINS"
2150 PRINT :::"  (8) BLACK RUSSIAN"
2160 PRINT :::"  (9) OLD-FASHIONED"
2170 PRINT :::"  (R) REPEAT FIRST SCREEN"
2180 CALL SOUND(150,1397,2)
2190 CALL KEY(0,K,S)
2200 IF K=82 THEN 1870
2210 IF (K<48)+(K>57)THEN 2190
2220 II=K-38
2230 CALL CLEAR
2240 C=VAL(DRINK$(II,1))
2250 GOSUB 250
2260 PRINT "  ";DRINK$(II,0)::::::
2270 ON VAL(DRINK$(II,2))GOTO 2280,2310
     ,2340,2370,2420,2450
2280 PRINT "FILL MIXING GLASS"
2290 PRINT "2/3 FULL WITH ICE CUBES"
2300 GOTO 2550
2310 PRINT "FILL HIGHBALL GLASS"
2320 PRINT "FULL WITH ICE CUBES"
2330 GOTO 2550
2340 PRINT "SALT RIM OF HIGHBALL GLASS"
2350 PRINT "FILL WITH ICE"
2360 GOTO 2550
2370 PRINT "FILL TALL FROSTED"
2380 PRINT "GLASS WITH ICE"
2390 PRINT "SQUEEZE 1/4 LIME"
2400 PRINT "OVER ICE-DROP IN"
2410 GOTO 2550
2420 PRINT "FILL OLD-FASHIONED GLASS"
2430 PRINT "WITH ICE"
2440 GOTO 2550
2450 PRINT "1 CUBE OF SUGAR IN"
2460 PRINT "OLD-FASHIONED GLASS"
2470 PRINT "2-3 DROPS ANGOSTURA BITTERS
     "
2480 PRINT "OVER CUBE OF SUGAR"
2490 PRINT "RIM GLASS WITH LEMON TWIST"
2500 PRINT "DROP IN":"1/2 OZ. SODA OR W
     ATER"
2510 PRINT "MUDDLE THOROUGHLY"
2520 PRINT "FILL WITH ICE"
2530 PRINT "1 OZ. BOURBON":"STIR"
2540 GOTO 2720
2550 FOR JJ=8 TO 23
2560 IF DRINKS(II,JJ)="" THEN 2580
2570 PRINT DRINKS(II,JJ);INVS(JJ-8,0)
2580 NEXT JJ
2590 ON VAL(DRINKS(II,3))GOTO 2720,2600
     ,2620,2640
2600 PRINT "STIR AND STRAIN INTO"
2610 GOTO 2660
2620 PRINT "SHAKE AND STRAIN INTO"
2630 GOTO 2660
2640 PRINT "STIR LIGHTLY"
2650 GOTO 2720
2660 ON VAL(DRINKS(II,4))GOTO 2720,2670
     ,2690,2710
2670 PRINT "3 OZ. CHILLED COCKTAIL GLAS
     S"
2680 GOTO 2720
2690 PRINT "5 OZ. CHILLED COCKTAIL GLAS
     S"
2700 GOTO 2720
2710 PRINT "WHISKEY SOUR GLASS"
2720 ON VAL(DRINKS(II,5))GOTO 2800,2730
     ,2760,2790
2730 PRINT "GARNISH WITH SPIKED OLIVE"
2740 CALL COLOR(15,13,C)
2750 GOTO 2800
2760 PRINT "GARNISH WITH SPIKED CHERRY"
2770 CALL COLOR(15,10,C)
2780 GOTO 2800
2790 PRINT "GARNISH WITH LEMON TWIST"
2800 IF DRINKS(II,6)="1" THEN 2820
2810 PRINT "AND ORANGE SLICE"
2820 ON VAL(DRINKS(II,7))GOTO 2860,2830
     ,2850
2830 PRINT "SERVE WITH STIR ROD"
2840 GOTO 2860
2850 PRINT "SERVE WITH TWO STRAWS"
2860 IF DRINKS(II,2)="1" THEN 2890
2870 ON VAL(DRINKS(II,2))-1 GOSUB 310,3
     10,700,850,850
2880 GOTO 2900
2890 ON VAL(DRINKS(II,4))-1 GOSUB 460,4
     60,1000
2900 ON VAL(DRINKS(II,5))GOSUB 290,1210
     ,1210,1250
2910 IF DRINKS(II,6)="1" THEN 2930
2920 GOSUB 1290
2930 ON VAL(DRINKS(II,7))GOTO 2980,2940
     ,2970
2940 CALL VCHAR(3,26,96,4)
2950 CALL HCHAR(4,26,110)
2960 GOTO 2980
2970 CALL VCHAR(3,26,99,4)
2980 IF DRINKS(II,2)<>"3" THEN 3010
2990 CALL COLOR(16,16,8)
3000 CALL HCHAR(4,23,153,5)
3010 CALL KEY(0,K,S)
3020 IF S=0 THEN 3010 ELSE 1760
3030 CALL CLEAR
3040 PRINT "IN THE FOLLOWING LIST,"
3050 PRINT "PRESS ""Y"" IF YOU HAVE"
3060 PRINT "THE INGREDIENT."
3070 PRINT "PRESS ""N"" IF YOU DO NOT."
3080 PRINT :"PRESS ""B"" TO BACK UP":::
     :::
3090 CALL SOUND(150,1397,2)
3100 YS=0
3110 FOR KK=0 TO 15
3120 PRINT "_";INVS(KK,0)
```

```
3130 CALL KEY(0,KEY,S)
3140 IF KEY=66 THEN 3030
3150 IF KEY=78 THEN 3180
3160 IF KEY<>89 THEN 3130
3170 YS=YS+1
3180 CALL HCHAR(23,3,KEY)
3190 INVS(KK,1)=CHR$(KEY)
3200 NEXT KK
3210 DR=0
3220 PRINT ::"YOU CAN MAKE:":::
3230 IF YS>1 THEN 3270
3240 PRINT "NOTHING; SORRY.":::"YOU NEED
     TO GO TO THE LIQUOR"
3250 PRINT "STORE IF YOU'RE THIRSTY."
3260 GOTO 3380
3270 FOR I=1 TO 19
3280 FOR J=8 TO 23
3290 IF DRINKS$(I,J)="" THEN 3310
3300 IF INV$(J-8,1)="N" THEN 3350
3310 NEXT J
3320 PRINT DRINKS$(I,0)
3330 CALL SOUND(150,1397,2)
3340 DR=DR+1
3350 NEXT I
3360 IF DR=0 THEN 3240
3370 PRINT :"THAT'S ALL."
3380 PRINT :"PRESS ANY KEY TO CONTINUE"
3390 CALL KEY(0,K,S)
3400 IF S=0 THEN 3390 ELSE 1760
3410 DATA MARTINI,16,1,2,2,2,1,1
3420 DATA "1 OZ. ....1/3 OZ. .........
     ....."
3430 DATA "DRY MARTINI",16,1,2,2,2,1,1
3440 DATA "1 1/4 OZ. ....1/4 OZ. .....
     ....."
3450 DATA "EXTRA DRY MARTINI",16,1,2,2,
     2,1,1
3460 DATA "1 1/2 OZ. ....2-3 DROPS ...
     ....."
3470 DATA "VODKA MARTINI",16,1,2,2,2,1,
     1
3480 DATA "","1 OZ. ....1/3 OZ. .......
     ....."
3490 DATA MANHATTAN,7,1,2,2,3,1,1
3500 DATA "",1 OZ. ....1/2 OZ. .....
     "2-3 DROPS ....."
3510 DATA "DRY MANHATTAN",7,1,2,2,2,1,1
3520 DATA "",1 OZ. ....1/2 OZ. .......
     .....
3530 DATA "SWEET MANHATTAN",7,1,2,2,3,1
     ,1
3540 DATA "",1 OZ. ....1/2 OZ. .....
     1/4 OZ. ....2-3 DROPS .....
3550 DATA "PERFECT MANHATTAN",7,1,2,2,4
     ,1,1
3560 DATA "",1 OZ. ....1/4 OZ. ....1/4
     OZ. ....2-3 DROPS .....
3570 DATA "WHISKEY SOUR",12,1,3,4,3,2,1
3580 DATA "",1 OZ. ....1 OZ. ....1/
     2 OZ. .....
3590 DATA "WARD EIGHT",7,1,3,4,3,2,1
3600 DATA "",1 OZ. ....1 OZ. .....
     1/2 OZ. .....
3610 DATA "DAIQUIRI",16,1,3,3,1,1,1
3620 DATA "",1 OZ. ....1 OZ. ....1/
     2 OZ. .....
3630 DATA BACARDI,7,1,3,3,1,1,1
3640 DATA "",1 OZ. ....1 OZ. .....
     1/2 OZ. .....
3650 DATA SCREWDRIVER,11,2,1,1,1,1,2
3660 DATA "",1 OZ. ..........FILL W
     ITH .....
3670 DATA "PINK LADY",10,1,3,3,1,1,3
3680 DATA "1 OZ. ..........1/2 OZ. ...
     ....1 1/2 OZ. .....
3690 DATA "SALTY DOG",16,3,4,1,1,1,2
3700 DATA "1 OZ. ..........FILL WITH
     ..... ....."
3710 DATA "GIN COOLER",7,4,4,1,3,2,3
3720 DATA "1 OZ. ..........1 OZ. ....1/
     2 OZ. ....FILL WITH .....
3730 DATA "TOM COLLINS",16,4,4,1,3,2,3
3740 DATA "1 OZ. ..........1 OZ. ....1/2
     OZ. ....FILL WITH .....
3750 DATA "BLACK RUSSIAN",2,5,4,1,1,1,1
3760 DATA "",1 OZ. ..........1/2 OZ. ....
     .....
3770 DATA "OLD-FASHIONED",7,6,1,1,3,2,1
3780 DATA "",1 OZ. ..........2-3 DROPS
     ....1/2 OZ. .....
3790 DATA GIN,VODKA,BOURBON,"DRY VERMOU
     TH","LIGHT RUM","SWEET VERMOUTH",K
     AHLUA
3800 DATA "LEMON JUICE","SIMPLE SYRUP",
     "ANGOSTURA BITTERS",GRENADINE,"GRA
     PEFRUIT JUICE"
3810 DATA "ORANGE JUICE","GINGER ALE",
     SODA,"MILK OR CREAM"
```

THE RULE OF 78

TI BASIC

"Why Mr. Templeton, you can't figure that!" said the lady at the finance company. I had merely asked her the formula for computing the payoff amount on the installment contract on my 1978 Datsun.

This emphatic "can't do" sent me racing off to the library in my soon-to-be-liberated Datsun. And it was there that I discovered the existence of the *Rule of 78*. So, armed with this knowledge, I decided to write a program that applied the Rule to installment contracts and let my TI-99/4A do the figuring for me.

From the name of this article you might have expected some sort of game, but the Rule of 78 is no game. It determines the amount of money required to pay off an installment contract at any given time, or the amount to be re-financed when you trade in before making all the payments. Should you be so unfortunate and have to default, the Rule of 78 determines the balance that becomes due and payable—the amount the finance company would be entitled to recover by repossessing the car. This Rule also is the method recognized by the Internal Revenue Service for computing the portion of the finance charge deductible each year during the life of the contract.

The Rule of 78 defines the fraction of the total finance charge that is on the unused portion. The numerator of the fraction is the sum of the numbers of the remaining payments; the denominator is the sum of the numbers of all payments. The number of the first payment is equal to the number of payments in the contracts—e.g., 48 payments for a four-year contract. The number of each succeeding payment is one less; the last payment is number 1. At the time the Rule got its name, 12-payment contracts were the usual type. The sum of 12, 11,. . ., and 1 is 78, the denominator of the fraction. A more appropriate name in our day would be rule of 1176, which is the sum of 48 through 1.

Many installment contracts allow an *acquisition charge* to be deducted from the finance charge before multiplying it by the fraction. This is almost a prepayment penalty, but not quite—because you usually pay *only a portion* of the acquisition charge. When applicable, the acquisition charge affects the payoff amount of the contract.

The Rule of 78 is also known as the Sum of the Monthly Balances Method and the Sum of the Months Digits Method. According to the *Consumer and Commercial Credit Installment Sales*, a subscription service published by Prentice-Hall, it is widely used in installment contracts. From these volumes, which contain federal and state law on the subject, I discovered that the Rule is required by law in some states and allowed by law in all states. It applies to installment contracts on automobiles, furniture, and appliances, and to some types of loans. Internal Revenue Service Publication 545, *Interest Expense*, explains the Rule and its application to income tax deductions.

Running the Program

The program is shown in the listing at the end of this article. It is written in TI BASIC, but will also run in TI Extended BASIC. Copy the program into your computer and enter the RUN command.

Consult a copy of the contract. First, be sure it mentions the Rule of 78 or one of its aliases in the section on prepayment. Then locate the amounts requested in the initial display. All of the amounts are usually typed in except the acquisition charge; it is printed in the contract. The display is as follows:

```
INSTALLMENT PAYMENTS
AMOUNT FINANCED: $
FINANCE CHARGE: $
ACQUISITION CHARGE: $
AMOUNT OF PAYMENT: $
NUMBER OF PAYMENTS:
FIRST PAYMENT DATE:
```

The prompts of the displays are typical of the names used in contracts. The amount financed is the sum of the price of the merchandise, sales taxes, insurance, etc., less the down payment. The finance charge is the amount added to the amount financed to compute the total of payments. The acquisition charge is printed in the section on prepayment. (It is $25 in many contracts.) It is easy to come up with the amount of payment: That's the amount you pay each month. Typically, you make 12 payments on appliances and 48 on new cars. Enter the date of the first payment expressed as three numbers separated by slashes. The first number represents the month, 1 through 12. The second is the day of the month, 1 through 31. The last number is the year, represented by the last two digits. For example, if the first payment were due December 23, 1984, you would enter 12/23/84.

After you enter the figures, the program lists the options as follows:

```
CHOOSE ONE
  1. CONTRACT SCHEDULE
  2. CONTRACT STATUS
  3. TAX DEDUCTION
  4. NEW CONTRACT
ENTER NUMBER:
```

The Contract Schedule option provides the date, total paid, balance prepay amount, and amount saved by prepaying for the first payment. By pressing ENTER you request the next payment. By repeatedly pressing ENTER you can display these five items for each payment of the contract. On the display for December of each year, the program also displays the tax deduction for the year.

When you specify the Contract Status option, the program requests a date. The program then displays the status of the contract on that date. If the date is during the period of the contract, the status display includes the date, total paid, balance, prepay amount, amount saved

by prepayment, and the tax deduction for the year if the contract is prepaid on that date. The status figures, of course, apply only if all payments have been made up to the requested date.

The Tax Deduction option shows you the allowable income tax deduction for each year of the contract. This same information is provided in the contract schedule displays; because this option gives you *only* the tax deduction, it is much faster. In many cases, prepayment is not possible, but deducting the proper portion of the finance charge is important.

The New Contract option returns to the beginning of the program and requests the inputs previously described. If you were really into installment contracts, you could compute the figures for the contract on your car, then on your TV, etc. Option 4 would enable you to enter figures for each additional contract.

If you select option 1, 2, or 3, the program lists the values you entered at the top of the screen, as follows:

```
AMOUNT FINANCED: $2,545.73
FINANCE CHARGE: $   781.03
ACQUISITION CHARGE:$ 25.00
AMOUNT OF PAYMENT:$ 92.41
NUMBER OF PAYMENT:     36
FIRST PAYMENT:      12/23/80
```

Below this display, the specific display for the selected option appears. For the Contract Schedule option, the following display is repeated for each payment:

```
          CONTRACT SCHEDULE

AFTER PAYMENT ON 12/23/80
TOTAL PAID          $    92.41
BALANCE             $ 3,234.35
PREPAY AMOUNT       $ 2,558.92
SAVE BY PREPAY      $   675.43

DEDUCTION FOR 1980 $42.22
```

For the Contract Status option, the initial display requests the date, as follows:

```
CONTRACT STATUS
ENTER DATE:
```

Enter a date in the format previously described. If you enter a date before the month of the first payment, the following is displayed:

```
STATUS ON 11/30/80
TOO EARLY
```

On the other hand, if you enter a date later than the last day of the month in which you will make the last payment, the following is displayed:

```
STATUS ON 12/1/83
PAID UP
```

When you enter a date during the period of the contract, the following is displayed:

```
STATUS ON 12/31/81:

TOTAL PAID          $ 1,201.33
BALANCE             $ 2,125.43
PREPAY AMOUNT       $ 1,838.23
SAVE BY PREPAY      $   287.20

DEDUCTIBLE IN 81 $     451.61
IF PAID OFF ON 12/31/81
```

For the tax deduction option, the display is as follows:

```
IF YOU PAY ALL PAYMENTS
AS SCHEDULED, YOU MAY
DEDUCT FINANCE CHARGE
AS FOLLOWS:

YEAR                    AMOUNT
1980          $           42.22
1981          $          415.14
1982          $          246.27
1983          $           77.40
```

At the bottom of each screen, the program displays the following message;

```
PRESS ENTER TO CONTINUE
OR 9 TO QUIT
```

For the Contract Schedule option, you get the figures for the next payment when you press ENTER. When all payments have been displayed, pressing ENTER displays the list of options previously described. For options 2 and 3, which have one screen each, pressing ENTER displays the option list.

The accuracy of the figures depends on the accuracy of the computer. Texas Instruments claims ten digits of accuracy for the TI-99/4A. In the case of the contract on my 1978 Datsun, the finance company's figures were not exactly the same as mine. The differences were a penny or two, most likely due to differences in computer accuracy. Of course, I paid the amount *their* computer wanted.

Changing the Program

If you have a printer, you will want to change the program to print the data displayed on the screen and you will probably want to change the format as well. The contract schedule can be printed in tabular form, one line per payment, on an 80-column printer.

The program has a subroutine for each option, but you may not want all the options; if not, you can leave one or two out. The contract schedule subroutine begins on line 680, and ends on line 1540. The contract status subroutine begins on line 1560 and continues through line 2280. The tax deduction subroutine occupies lines 2300 through 2650. Each subroutine is independent of the other two; however, the driver (lines 170 through 660) and the miscellaneous subroutines from line 2670 to the end of the program are required for all subroutines.

Streamlining for TI Extended BASIC

You can run the program in Extended BASIC as it is, or you can streamline it, exploiting some of the features of the more powerful language. The power of the DISPLAY statement of Extended BASIC is particularly valuable in this program.

Line 170 is a DEF statement that defines a rounding function. A format defined by an IMAGE statement automatically rounds fractions, and this function is used to align decimal points in the displays. The function is not needed if you use a specified format.

The subroutine beginning at line 2880 displays a string at a defined point on the screen. When you use a

DISPLAY statement with the AT option, this subroutine is not required. Similarly, the subroutine at line 2940 adds zeros to the right of the decimal point, where required. It also inserts a comma between the hundreds and thousands digit of numbers greater than 999.99. A defined format adds least significant zeros but does not insert the comma. If you want to use a format and give up the comma, omit this subroutine.

To incorporate these changes, modify the program shown in Listing 1 by performing the following steps:

1. Omit line 170 and modify line 180 as follows:
   180 IMAGE" #####.##"
2. Omit lines 490 and 500; modify line 510 as follows:
   510 PRINT USING "AMOUNT FINANCED : :$#####.##":UB
3. Omit lines 520 and 530; modify line 540 as follows:
   540 PRINT USING "FINANCE CHARGE :$ #####.##":FC
4. Omit lines 550 and 560; modify line 570 as follows:
   570 PRINT USING "ACQUISITION CHARGE: $####.##":AC
5. Omit lines 580 and 590; modify line 600 as follows:
   600 PRINT USING "AMOUNT OF PAYMENT: $####.##":PMNT
6. Omit line 630 and modify line 640 as follows:
   640 PRINT USING "FIRST PAYMENT: ##/##/##" ;MO,DA,YR
7. Omit lines 810, 830, and 840; modify line 850 as follows:
   850 DISPLAY AT (14, 17):USING "##/##/##": CMO, DA,CYR
8. Omit lines 880-910; modify line 930 as follows:
   930 DISPLAY AT(15, 17):USING 180:TOTPD
9. Omit lines 950-990; modify line 1000 as follows:
   1000 DISPLAYS AT(16, 17):USING 180:BAL
10. Omit lines 1080-1110; modify line 1130 as follows:
    1130 DISPLAY AT(17, 17):USING 180:PREPAY
11. Omit lines 1140-1170; modify line 1190 as follows:
    1190 DISPLAY AT(18, 17):USING 180:SAV
12. Omit lines 1280-1310 and 1330; modify line 1340 as follows:
    1340 DISPLAY AT(20,1):USING "DEDUCTION FOR 19## $###.##":CYR,ADED
13. Omit lines 1430-1460 and 1480; modify line 1490 as follows:
    1490 DISPLAY AT(20,1):USING "DEDUCTION FOR 19## $####.##":CYR,ADED
14. Omit lines 1830 and 1840; modify line 1850 as follows:
    1850 PRINT USING "TOTAL PAID $ #####.## ":TOTPD
15. Omit lines 1880 and 1890; modify line 1900 as follows:
    1900 PRINT USING "BALANCE $#####.##" :BAL
16. Omit lines 1970 and 1980; modify line 1990 as follows:
    1990 PRINT USING "PREPAY AMOUNT $ #####.## ":PREPAY
17. Omit lines 2000 and 2010; modify line 2020 as follows:
    2020 PRINT USING "SAVE BY PREPAY $ #####.## ":SAV
18. Omit lines 2140 and 2150; modify line 2160 as follows:
    2160 PRINT USING "DEDUCTIBLE IN ## $ #####.##" :SYR,DEDUCT
19. Omit lines 2440 and 2450; modify line 2430 as follows;
    2430 PRINT USING "19## $#####.##" :DYR,DED
20. Omit lines 2590 and 2600; modify line 2580 as follows:
    2580 PRINT USING "19## $#####.##" :DYR,DED
21. Omit lines 2690-2710 and modify line 2720 as follows:
    2720 DISPLAY AT(23,1):"PRESS ENTER TO CONTINUE"
22. Omit lines 2730 and 2740; modify line 2750 as follows:
    2750 DISPLAY AT(24,1):"OR 9 TO QUIT"
23. Remove references to function RND2 in the following lines:

```
1050 SAV = RUL 78 (AFC)
1270 ADED = DEDUCT/DEN*FC
1400 SAV = X
1420 ADED = DEDUCT/DEN*FC
1940 SAV = RUL 78(AFC)
2130 DEDUCT = DEDUCT - RUL78(FC)
2200 SAV = 1/DEN*AFC
2420 DED = RUL78(FC)
2510 DED = NP/DEN*FC
2570 DED = RUL78(FC)
2640 DED = 1/DEN*FC
```

The subroutine beginning at line 2810 is not required if an ACCEPT statement is used to input the character. Omit line 2770 and modify lines 2760 and 2780 as follows:

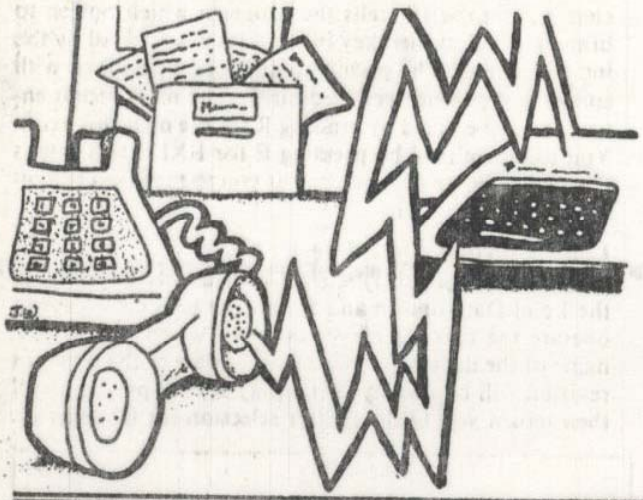
```
2760 ACCEPT AT(24,14):SEL$
2780 IF SEL$ = "9" THEN 2800
```

And don't forget to omit the subroutines (lines 2810-3070). Extended BASIC allows further compression by putting several statements on the same line and by using statements in IF-THEN-ELSE statements. However, the changes I have suggested provide a significant reduction in the size of the program.

With this program in your computer, you have all the secrets of the Rule of 78 at your disposal. Your computer will tell you everything you ever wanted to know about an installment contract, but didn't ask because you would not have been told.

```
100 REM  *********************
110 REM  *      PAYMENTS:      *
120 REM  *     RULE OF 78      *
130 REM  *********************
140 REM
150 REM
160 REM     DEFINE FUNCTIONS
170 DEF  RND2(X)=INT(X*100+.5)/100
180 DEF  INC=28-LEN(XS)
190 DEF  RUL78(D)=P*Q/2/DEN*D
200 REM   INPUT DATA  *********
210 CALL  CLEAR
220 PRINT  "      INSTALLMENT PAYMENTS":::
230 INPUT  "AMOUNT FINANCED: $":UB
240 INPUT  "FINANCE CHARGE: $":FC
250 INPUT  "ACQUISITION CHARGE: $":AC
260 AFC=FC-AC
270 INPUT  "AMOUNT OF PAYMENT: $":PMNT
280 INPUT  "NUMBER OF PAYMENTS: ":NP
290 INPUT  "FIRST PAYMENT DATE: ":DATE$
300 M=POS(DATE$,"/",1)
310 MO$=SEG$(DATE$,1,M-1)
320 M=POS(DATE$,"/",M+1)
330 L=M-N-1
340 DA$=SEG$(DATE$,N+1,L)
350 YR$=SEG$(DATE$,M+1,2)
360 MO=VAL(MO$)
370 DA=VAL(DA$)
380 YR=VAL(YR$)
390 DEN=NP*(NP+1)/2
400 PRINT :::"CHOOSE ONE:":::
410 PRINT  " 1 CONTRACT SCHEDULE"
420 PRINT  " 2 CONTRACT STATUS"
430 PRINT  " 3 TAX DEDUCTION"
440 PRINT  " 4 NEW CONTRACT":::
450 INPUT  "ENTER NUMBER: ":SEL
460 IF  SEL=4 THEN 210
470 IF  (SEL<1)+(SEL>3)<0  THEN 400
480 CALL  CLEAR
490 X$=STR$(UB)
500 GOSUB  2940
510 PRINT  "AMOUNT FINANCED: $";TAB(INC
    );X$
520 X$=STR$(FC)
530 GOSUB  2940
540 PRINT  "FINANCE CHARGE: $";TAB(INC)
    ;X$
550 X$=STR$(AC)
560 GOSUB  2940
570 PRINT  "ACQUISITION CHARGE:";TAB(20
    );"$";TAB(INC);X$
580 X$=STR$(PMNT)
590 GOSUB  2940
600 PRINT  "AMOUNT OF PAYMENT:";TAB(20)
    ;"$";TAB(INC);X$
610 X$=STR$(NP)
620 PRINT  "NUMBER OF PAYMENTS:";TAB(26
    );X$
630 X$=DATE$
640 PRINT  "FIRST PAYMENT:";TAB(INC);DA
    TE$
650 ON SEL GOSUB 680,1560,2300
660 GOTO 400
670 REM  DISPLAY SCHEDULE **
680 PRINT :::"  CONTRACT SCHEDULE":::
690 PRINT  "AFTER PAYMENT ON"
700 PRINT  "TOTAL PAID          $"
710 PRINT  "BALANCE             $"
720 PRINT  "PREPAY AMOUNT       $"
730 PRINT  "SAVE BY PREPAY      $":::::::
740 CALL  CLEAR
750 CALL  HCHAR(14,21,32,9)
760 CMO=CMO+1
770 IF  CMO>12  THEN  CMO=1
780 CYR=CYR+(CMO=1)
790 GOSUB  2940
800 CALL  HCHAR(14,21,32,9)
810 FOR I=NP TO 1 STEP -1
820 X$=STR$(CMO)&"/"&DA$&"/"&STR$(CYR)
830 C=INC+1
840 R=14
850 GOSUB  2880
860 CMO=CMO+1
870 TOTPD=TOTPD+PMNT
880 X$=STR$(TOTPD)
890 GOSUB  2940
900 C=INC+1
910 R=15
920 CALL  HCHAR(15,19,32,11)
930 GOSUB  2880
940 BAL=BAL-PMNT
950 X$=STR$(BAL)
960 GOSUB  2940
970 C=INC+1
980 R=16
990 CALL  HCHAR(16,19,32,11)
1000 GOSUB  2880
1010 P=I-2
1020 IF  P=1  THEN  1390
1030 IF  P<0  THEN  1370
1040 Q=P+1
1050 SAV=RND2(RUL78(AFC))
1060 IF  SAV<1  THEN  1370
1070 PREPAY=BAL-SAV
1080 R=17
1090 X$=STR$(PREPAY)
1100 GOSUB  2940
1110 C=INC+1
1120 CALL  HCHAR(17,19,32,11)
1130 GOSUB  2880
1140 R=18
1150 X$=STR$(SAV)
1160 GOSUB  2940
1170 C=INC+1
1180 CALL  HCHAR(18,19,32,11)
1190 GOSUB  2880
1200 N=N+1
1210 DEDUCT=DEDUCT+I
1220 IF  CMO>12  THEN  1420
1230 GOSUB  2670
1240 NEXT  I
1250 IF  DEDUCT=0  THEN  1360
1260 Q=DEDUCT
1270 ADED=RND2(DEDUCT/DEN*FC)
1280 C=2
1290 R=20
1300 X$=STR$(ADED)
1310 GOSUB  2940
1320 CALL  HCHAR(20,23,32,9)
1330 X$="DEDUCTION FOR 19"&STR$(CYR)&"
     $"&X$
1340 GOSUB  2880
1350 GOSUB  2670
1360 RETURN
1370 SAV=0
1380 GOTO  1070
1390 X=1/DEN*AFC
1400 SAV=RND2(X)
1410 GOTO  1060
1420 ADED=RND2(DEDUCT/DEN*FC)
1430 X$=STR$(ADED)
1440 GOSUB  2940
1450 C=2
1460 R=20
1470 CALL  HCHAR(20,23,32,9)
1480 X$="DEDUCTION FOR 19"&STR$(CYR)&"
     $"&X$
1490 GOSUB  2880
1500 DEDUCT=0
1510 N=0
1520 CMO=1
1530 CYR=CYR+1
1540 GOTO  1250
1550 REM  DISPLAY STATUS **
1560 PRINT :::"CONTRACT STATUS":::
1570 INPUT  "ENTER DATE: ":SDATE$
1580 CALL  HCHAR(23,1,32,32)
1590 PRINT  "STATUS ON ";SDATE$;":":::
```

```
1600 N=POS(SDATE$,"/",1)
1610 SMO$=SEG$(SDATE$,1,N-1)
1620 M=POS(SDATE$,"/",N+1)
1630 L=M-N-1
1640 SDA$=SEG$(SDATE$,N+1,L)
1650 SYR$=SEG$(SDATE$,M+1,2)
1660 SMO=VAL(SMO$)
1670 SDA=VAL(SDA$)
1680 SYR=VAL(SYR$)
1690 TMO=MO+NP-INT((MO+NP)/12)*12
1700 TMO=TMO+(TMO=0)*-12
1710 TYR=YR+INT(NP/12)-(TMO<MO)
1720 IF SYR>TYR THEN 2240
1730 IF SYR<YR THEN 2270
1740 IF (SYR=TYR)+(SMO>=TMO)=-2 THEN 22
     40
1750 IF (SYR=YR)+(SMO<MO)=-2 THEN 2270
1760 N=SYR-YR
1770 N=N*12
1780 X=SMO-MO
1790 N=N+X
1800 IF SDA<DA THEN 1820
1810 N=N+1
1820 TOTPD=N*PMNT
1830 X$=STR$(TOTPD)
1840 GOSUB 2940
1850 PRINT "TOTAL PAID $";TAB(INC);
     X$
1860 BAL=UB+FC
1870 BAL=BAL-TOTPD
1880 X$=STR$(BAL)
1890 GOSUB 2940
1900 PRINT "BALANCE $";TAB(INC);
     X$
1910 P=NP-N-1
1920 IF P=1 THEN 2200
1930 Q=P+1
1940 SAV=RND2(RUL78(AFC))
1950 IF SAV<1 THEN 2220
1960 PREPAY=BAL-SAV
1970 X$=STR$(PREPAY)
1980 GOSUB 2940
1990 PRINT "PREPAY AMOUNT $";TAB(INC);
     X$
2000 X$=STR$(SAV)
2010 GOSUB 2940
2020 PRINT "SAVE BY PREPAY $ ";TAB(INC)
     ;X$
2030 DEDUCT=FC-SAV
2040 IF SYR=YR THEN 2140
2050 F=13-MO
2060 AYR=YR+1
2070 IF SYR=AYR THEN 2110
2080 F=F+12
2090 AYR=AYR+1
2100 GOTO 2070
2110 P=F
2120 Q=NP+(NP-F+1)
2130 DEDUCT=DEDUCT-RND2(RUL78(FC))
2140 X$=STR$(DEDUCT)
2150 GOSUB 2940
2160 PRINT "DEDUCTIBLE IN ";SYR$;" $";T
     AB(INC);X$
2170 PRINT "IF PAID OFF ON ";SDATE$:::
2180 GOSUB 2670
2190 RETURN
2200 SAV=RND2(1/DEN*AFC)
2210 GOTO 1960
2220 SAV=0
2230 GOTO 1960
2240 PRINT "PAID UP"::
2250 GOSUB 2670
2260 RETURN
2270 PRINT "TOO EARLY"::
2280 GOTO 2250
2290 REM TAX DEDUCTION ****
2300 PRINT :::"IF YOU PAY ALL PAYMENTS"
2310 PRINT "AS SCHEDULED, YOU MAY"
2320 PRINT "DEDUCT FINANCE CHARGE"
2330 PRINT "AS FOLLOWS:"::
2340 PRINT "YEAR                    AMOUNT
     "::
2350 ND=NP
2360 DYR=YR
2370 P=13-MO
2380 IF P=1 THEN 2510
2390 P=(P<=ND)*-P+(P>ND)*-ND
2400 N=ND-P+1
2410 Q=ND+N
2420 DED=RND2(RUL78(FC))
2430 X$=STR$(DED)
2440 GOSUB 2940
2450 PRINT "19";STR$(DYR);"          $";
     TAB(INC);X$
2460 ND=ND-P
2470 DYR=DYR+1
2480 IF ND<12 THEN 2530
2490 P=12
2500 GOTO 2400
2510 DED=RND2(NP/DEN*FC)
2520 GOTO 2430
2530 IF ND=0 THEN 2610
2540 IF ND=1 THEN 2640
2550 P=ND
2560 Q=ND+1
2570 DED=RND2(RUL78(FC))
2580 X$=STR$(DED)
2590 GOSUB 2940
2600 PRINT "19";STR$(DYR);"          $";
     TAB(INC);X$
2610 PRINT :::
2620 GOSUB 2670
2630 RETURN
2640 DED=RND2(1/DEN*FC)
2650 GOTO 2580
2660 REM NEW SCREEN *******
2670 CALL GCHAR(23,25,X)
2680 IF X<>32 THEN 2760
2690 C=2
2700 R=23
2710 X$="PRESS ENTER TO CONTINUE"
2720 GOSUB 2880
2730 R=24
2740 X$="OR 9 TO QUIT"
2750 GOSUB 2880
2760 C=16
2770 GOSUB 2820
2780 IF SEL=57 THEN 2800
2790 RETURN
2800 STOP
2810 REM INPUT SUBROUTINE *
2820 CALL HCHAR(24,C,30)
2830 CALL KEY(0,SEL,STAT)
2840 IF STAT=0 THEN 2830
2850 CALL HCHAR(24,C,32)
2860 RETURN
2870 REM PRINT SUBROUTINE *
2880 FOR J=1 TO LEN(X$)
2890 X=ASC(SEG$(X$,J,1))
2900 CALL HCHAR(R,C+J,X)
2910 NEXT J
2920 RETURN
2930 REM EDIT ROUTINE *****
2940 Q=POS(X$,".",1)
2950 IF Q=0 THEN 2990
2960 IF Q=LEN(X$)-1 THEN 3010
2970 IF LEN(X$)>6 THEN 3030
2980 RETURN
2990 X$=X$&".00"
3000 GOTO 2970
3010 X$=X$&"0"
3020 GOTO 2970
3030 A=LEN(X$)-6
3040 Y$=SEG$(X$,1,A)
3050 Z$=SEG$(X$,A+1,6)
3060 X$=Y$&","&Z$
3070 RETURN
3080 END
```



The Electronic Home Secretary

TI BASIC

Now that you have a personal computer, you've probably been looking for ways to use it around the house. When writing software for home applications, it's often possible to create a *general* program that functions in a variety of household situations. The program accompanying this article follows this design philosophy. With it, you can create a personal phone and address directory, time events (such as elapsed telephone connect time), have your computer dial or redial any number in your directory, and set up an inventory of household possessions for insurance and maintenance purposes. All this in standard 16K TI BASIC—with some room to spare for customizing the program according to your preference.

GENERAL DESCRIPTION OF THE PROGRAM

Data Entry

When the program is first RUN, the screen options give the user a choice of updating or using a previous data file saved on cassette or disk, or creating an entirely new data file for one of two options: (1) the phone and address directory, or (2) the household inventory. Both of these options also provide sub-options: For example, the program can draw on the data files to dial (by the dual-tone method) an appropriate phone number, or sum the total cost in the inventory, and then print hardcopy listings of either. The category names for the file organization are provided in the DATA statements 220 and 230.

The input data is stored in the arrays A1$, A2$, A3$, A4$, and A5$. A dimension of 60 is assigned to each of the arrays, and a maximum string length of 190 characters is allowed for each complete entry. Line 710 checks the validity of each data set. At this stage, the program also checks for dimension overflow and memory overflow (lines 480 and 810), and appropriate warning messages are displayed. These features prevent you from accidentally keying in excess data—a situation that would result in an error and program termination. Additionally, the cost category (A2$) in option 2 is designed to accept only numerical input so that you can conveniently carry out numerical operations on the data—for example, total the cost of possessions. And keep in mind that you can, of course, change the categories by altering the data in lines 220 and 230.

Sort Routine

An efficient sort subroutine is presented in the program at line 2410. The routine employs a tree sort procedure which needs approximately $2*N*(\text{Log}_2\ N-1)$ comparisons to sort N entries. Since various versions of sorting routines have been previously published and are readily available, I won't discuss the mathematical details of the sorting procedure. [See reference 2, for example, or any elementary book on numerical analysis.—Ed.] Here, the sorting is based on the entries in the arrays A1$ (i.e., names or items in the default categories). The remaining arrays are appropriately rearranged to be consistent with the original data. The procedure is carried out without the use of any intermediate arrays, thereby saving on the core usage. Completely sorting and rearranging 50 entries takes about 4 minutes.

Data Deletion and Alteration

The subroutine at line 1010 updates any existing data set. You can access any particular entry by its serial number or by its name (or a segment of its name). A search routine (line 1790) retrieves the data set with the specified name, or the next higher one if the name match is not exact. As previously described, the program validates the altered data for allowable string length and memory overflow. At this stage, you have the option of moving up or down in the list, searching for a different entry, or finishing the editing session. Any alteration of the entry title (i.e., A1$) causes the variable FLAG2 to be set equal to unity. Before the directory can be displayed, the data set is resorted.

Display of the Directory

The program allows you to display the data directory in two formats. The first format (at line 1420) provides a concise, quick-reference listing of the complete directory. This includes name and phone number for the Phonebook option, and item and cost for the Inventory option.

In the second format, you can display all the data contained in any single entry. Access to individual entries is either by its serial number in the directory, or by a string search as discussed in the previous section.

Additionally, you can get a hardcopy listing of the entire directory (line 4280) through an RS232-compatible printer, or the TI thermal printer. The screen printing

routine at line 4150 was used to get a hardcopy print-out of screen displays for this article. This portion (lines 4150-4260) can be deleted without affecting the operation of the program.

Computerized Phone Dialing

Now let's look at Touch-Tone dialing with the TI-99/4A. Since the telephone company prohibits direct connections to the phone line of any user equipment not approved by the FCC, the method we will have to use involves simple proximity: Placing the microphone from the phone handset in the front of the monitor speaker dials the phone without any direct connection to the phone lines.

Briefly, the Touch-Tone system of telephone dialing operates by sending a specific pair of audio frequency digit. The switching circuits at the telephone facility decode the tones and actuate the appropriate circuits to make the connection. The tone pairs consist of a low frequency group (697-941 Hz) and a high frequency group (1209-1477 Hz) as shown in Figure 1. For example, to dial the number 5, we have to send the audio tones at 770Hz and 1336 Hz simultaneously for a sufficiently long time to be recognized by the switching circuits. There should also be a sufficient gap between digits for each digit to register individually. Although a 40 millisecond signal duration followed by a 40 millisecond silence should theoretically be adequate, a 150-200 millisecond signal duration and a gap of about 100-150 milliseconds is required for reliable operation with this system.

With the CALL SOUND (*duration, frequency 1, volume 1, frequency 2, volume 2*) command of TI BASIC, the TI-99/4A can generate the dual tones of Figure 1. In doing this, however, an interesting problem arises: If we examine the monitor's output on an oscilloscope, we can observe that the so called "pure tone" from the computer is, in fact, a *square wave* and not a *sine wave*. By Fourier analysis, the square wave can be decomposed into its constituent sine waves. (Interested readers can refer to any elementary book of calculus for the details of the analysis.) To be specific, the output from CALL SOUND (100,500,1) is a square wave of 500 Hz for a 100 millisecond duration at the volume level 1. This is a combination of sine waves at 500 Hz, 1500 Hz, 2500 Hz, and so on. This can pose a problem when we try to dial the first two members (i.e., 698 Hz and 770 Hz) of the low frequency group. The third harmonics of these frequencies, namely, 2091 Hz and 2310 Hz, are also recognized by the switching circuits, resulting in the rejection of the signal. The third harmonics of 852 Hz and 941 Hz seem to be outside the frequency response of the switching circuits and pose no problem.

There are several ways we can overcome this problem when dialing the digits 1 thru 6. One very simple and inexpensive way is to use a passive low-pass filter with a cut-off frequency of about 1.5 KHz in the audio line to the monitor, thereby attenuating the higher frequencies. Figure 2 shows a block diagram for the installation. The circuit for the filter (which I built for less than five dollars) is shown in Figure 3.

HOW TO USE THE PROGRAM

Initial Set-Up

With the choice of N (for NO) for the Load Data option in Display 1, the program has you select either the Phone Directory or Household Inventory option. (If your choice was Y, and you loaded a file, one of the data elements on the file tells the program which option to branch to.) You then key in the data file, guided by the input prompts. The phone number can be entered with spaces and parentheses, if desired. The most recent entry can be re-entered by pressing R for the name (or item). You can terminate by pressing E for EXIT; this causes the data to be sorted and returns you to the master selection list (Display 3).

Load Previous Data File

To load a previously stored data file, we select 2 for the Load Data option and follow the screen displays to operate the cassette player or disk. When loaded, the name of the data file, its size and the date of the previous revision will be displayed (Display 2); the program will then return you to the master selection list (Display 3).

Low Frequency Group \ High Frequency Group	1209 Hz	1336 Hz	1477 Hz
697 Hz	1	2	3
770 Hz	4	5	6
852 Hz	7	8	9
941 Hz	*	0	#

Figure 1. Basic Frequencies for the Two-Tone System of Telephone Dialing

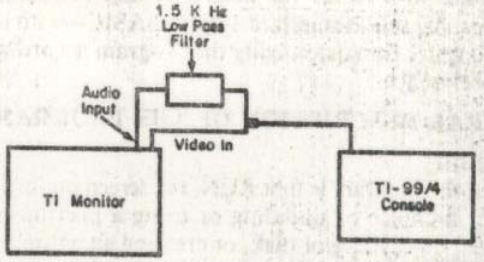


Figure 2. Schematic Layout of Filter Location

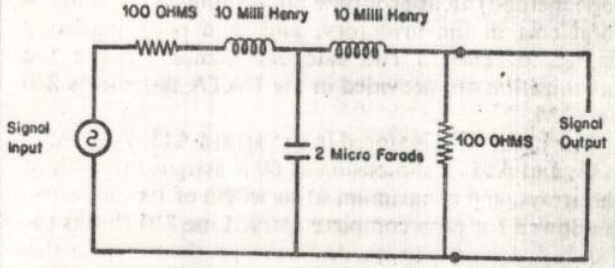


Figure 3. Circuit Diagram of the Filter

Note: On many touch-tone phone systems this filter will not be needed for correct dialing. We suggest you first try without it—Ed.

```
   LOAD DATA? (Y/N)      N
PHONE BOOK? (Y/N)        Y
   ENTER
      E TO EXIT
      R TO REENTER
NAME:DOE
PHONE: 987 6543
STREET:4321 NORTH SOUTH ST
CITY & ZIP:HOLLYWOOD; CA99888
MISC:JOHN; DATE OF BIRTH JAN
   1 1921; WIFE MARY; CHILDREN
   JOE ; SUSAN; WEDD ANNIV FEB
   25;
```

DISPLAY 1 INITIAL SET-UP

Note:

↵ = pressing ENTER, after the user's response

```
   LOAD DATA (Y/N)       Y
ENTER
1. CS1
2. DISK 1
3. OTHER

 * REWIND CASSETTE TAPE     CS1
   THEN PRESS ENTER

 * PRESS CASSETTE PLAY      CS1
   THEN PRESS ENTER

INVENTORY – 1
LSIZE(3800)=1628
LAST UPDATE: MARCH 26 81
```

DISPLAY 2 LOAD PREVIOUS DATA FILE* (FOR OPTION 1)

* OPTION 2: ENTER FILE NAME
OPTION 3: ENTER DEVICE NAME:

```
PRESS
1 – TO ADD MORE DATA
2 – TO ALTER THE DATA
3 – TO DISPLAY THE DIRECTORY
4 – TO DISPLAY ONE ENTRY
5 – TO USE THE DATA
6 – TO STORE DATA FILE
7 – FOR PRINTER LISTING
8 – TO END PROGRAM

*** UPDATE DIRECTORY ***
```

DISPLAY 3 MASTER SELECTION LIST

```
WHICH ONE; DOE
ENTER
   NEW DATA AT CURSOR
   'D' TO DELETE THE ITEM
   'ENTER' FOR NO CHANGES

DOE?
987 6543;  (424) 987 6543
4321 NORTH SOUTH ST?
HOLLYWOOD; CA99888?
JOHN; DATE OF BIRTH JAN 1 19
21; WIFE MARY; CHILDREN JOE
$ SUSAN: WEDD ANNIV FEB 25;?
```

DISPLAY 4 DATA ALTERATION

```
1. ARPACI JOE          321 1234
2. DOE           (424) 987 6543
3. DOE MARY      (424) 789 3456
4. MOORE N.          578 657 8901
5. NORTON P.             356 4473
6. OHSHIMA               368 8714
7. SASTRY M.             765 2345
8. SHIELD B.         654–789 4532
9. SHYAMALA              206 6808
0. SUBBAIAH      (213) 356 4473
1. WONG V.               256 3902
PRESS ANY KEY TO CONTINUE
```

DISPLAY 5 SHORT FORM DIRECTORY

```
   WHICH ONE?  DOE
DOE
(424) 987 6543
4321 NORTH SOUTH ST
HOLLYWOOD; CA99888
JOHN; DATE OF BIRTH JAN 1 19
21; WIFE MARY; CHILDREN JOE
& SUSAN; WEDD ANNIV FEB 25;
PRESS
   E  TO LIST UP
   X  TO LIST DOWN
   S  TO SEARCH MORE
PRESS ANY KEY TO CONTINUE
```

DISPLAY 6 SINGLE ITEM DISPLAY

```
   DOE
(424) 987 6543
4321 NORTH SOUTH ST
HOLLYWOOD; CA99888
JOHN; DATE OF BIRTH JAN 1 19
21; WIFE MARY; CHILDREN JOE
$ SUSAN; WEDD ANNIV FEB 25;
          1(424) 987 6543
PRESS
   R  TO REDIAL
   S  TO START STOPWATCH
   N  FOR NEW NUMBER
PRESS ANY KEY TO CONTINUE
PRESS
   R  TO REDIAL
   S  TO START STOPWATCH
   N  FOR NEW NUMBER
PRESS ANY KEY TO CONTINUE   S
HOLD DOWN
R  TO DIAL AGAIN
ANY KEY TO CONTINUE
                      00:55
```

DISPLAY 7 PHONE DIALING AND STOPWATCH

```
TOTAL COST OF ALL THE ITEMS

   $$$$$$$$$$$$$$$$$$
   $                $
   $                $
   $                $
   $     7450.6     $
   $                $
   $                $
   $                $
   $$$$$$$$$$$$$$$$$$

PRESS ANY KEY TO CONTINUE
```

DISPLAY 8 TOTAL COST OF INVENTORY

```
ENTER   1. CS1
        2. DSK1
        3. OTHER

        YOUR CHOICE?
TODAY'S DATE: MARCH 7 1981
DIR. NAME: PHONE BOOK – 1

 * REWIND CASSETTE TAPE     CS1
   THEN PRESS ENTER
 * PRESS CASSETTE RECORD    CS1
   THEN PRESS ENTER
 * PRESS CASSETTE STOP      CS1
   THEN PRESS ENTER
```

DISPLAY 9 SAVE DATA FILE

Master Selection List and Its Functions

The master selection list (Display 3) provides access to the program's various options. A banner (***UPDATE DIRECTORY***) will be displayed if there has been any alteration of the data file since the last update. This should act as a constant reminder to save the revised version of the data on a cassette or disk. The different options of the master selection list are as follows:

Option 1: Select this to add any new entry to the data file. This leads to the data entry of Display 1.

Option 2: This leads to Display 4. You can access any individual entry by its serial number in the directory (from display 5) or by a string search. Here, entering a null string (i.e., just pressing the ENTER key) for any category will leave it unaltered.

Option 3: This displays a short form of the directory as in Display 5. The display stops when the screen is filled. Pressing any key causes the remaining data to be displayed, or returns you to the master selection list if no more data is to be displayed.

Option 4: This produces a complete listing of a single entry (Display 6), selected by its serial number in the directory, or by a string search as in Display 4.

Option 5: This allows the program to use the data files when dialing/redialing in the Phonebook option, or to obtain the total purchase cost of the inventory in the Household Inventory option. If you are in the Phonebook option, the program will advance to Display 6. If you approve the display by pressing any key other than E, X, and S, the computer dials the displayed phone number. In the beginning, you may have to adjust the volume control of your TV set or monitor for proper operation. The digits will be displayed one by one as they are dialed. If the total number of characters in the phone number is greater than or equal to 10, the routine recognizes it as a long distance call, and dials 1 at the beginning (Display 7). After getting familiar with the operation, you may want to reduce the time periods assigned in the CALL SOUND statements in lines 3540, 3580, 3590. You can redial the number by pressing R, start the stopwatch by pressing S (and quickly releasing the key), or select a new number using the choice N. Any other key (including a prolonged pressing of S) terminates the dialing session and the master selection screen will be displayed.

With the selection of S, the stopwatch routine on line 3700 is activated. The elapsed time is displayed at the lower right-hand corner (Display 7). You can control the accuracy of the stopwatch by adjusting the time delay constants of the DATA statement in line 3320. Holding down R starts the dialing procedure all over again; pressing any other key returns you to the master selection list (Display 3).

In the Household Inventory option, choice 5 of the master selection list will cause the program to calculate the total purchase cost (Display 8) for all the items in the data file. There's no adjustment here for inflation. This, however, could easily be done. For example, you could key in the consumer price index into the data file at the time of an item's purchase and scale the purchase cost with the current index when evaluating the SUM (in the routine on line 3150). I felt, however, that this procedure would be rather involved for day-to-day use.

Option 6: This permits storing the data file on either cassette or disk. The computer asks (Display 9) for the title of the data file and the date of revision for future reference. This information will be displayed when you re-load the data for another session.

Option 7: This produces a hardcopy listing (with nine complete entries per page) on either the TI thermal printer, or a printer connected to the RS232 interface. The computer first asks you to verify that either the thermal printer or the RS232 interface is connected in order to avoid the File-Error termination. As a precaution, always SAVE the updated file on cassette or disk (option 6) *prior* to printing.

SUMMARY AND FINAL REMARKS

This program is capable of performing a wide variety of functions. We have seen how to use it to maintain a computerized phone directory and dial your phone automatically, as well as to maintain very flexible data files for day-to-day use in the home. Typical applications include an inventory of household valuables, a record of credit cards and bank accounts, lists of author/subject references for research, recipe files, etc. Some of the individual subroutines (in particular, the sorting routine and the stopwatch routine) should also be useful in many other applications. The program, as presented here, is contained within the standard 16K TI BASIC. A version in Extended BASIC to access the additional 32K RAM should give the program an even broader scope.

References

Floyd, R. W. "Algorithm 245, TREESORT 3." *Communications of the A. C. M.*, December 1964, p. 701.

Blinchikoff, H. J. and Zverev, A. I. *Filtering in the Time and Frequency Domain*. John Wiley and Sons, 1976, Chapter 4.

Luff, P. P. "The Electronic Telephone." *Scientific American*, March 1978, pp. 58-64.

Renbarger, J. "A Telephone-Dialing Microcomputer." *BYTE*, June 1980, pp. 140-170.

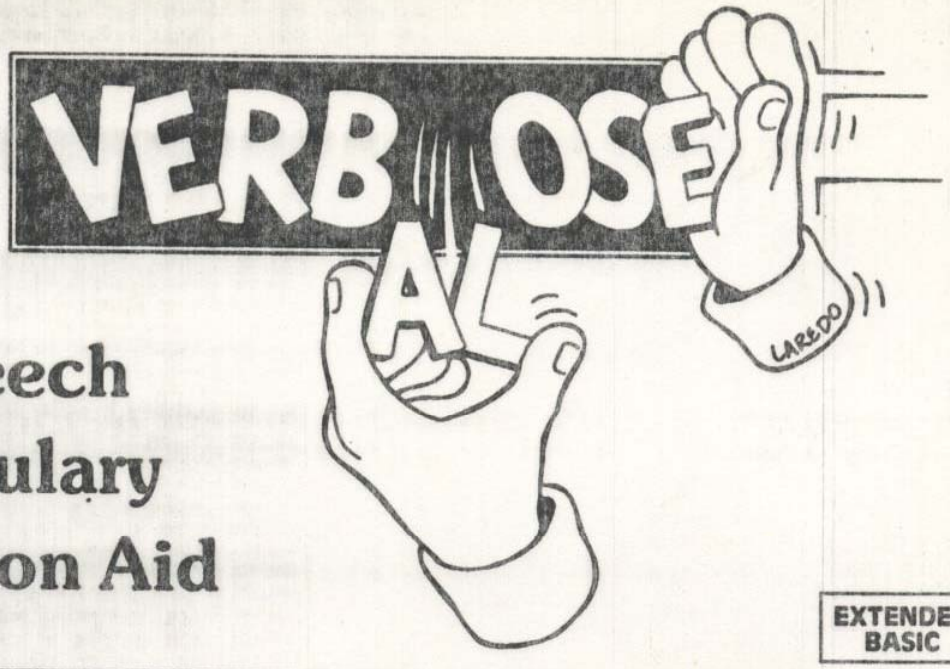

```
100 REM   *********************
110 REM   *  THE HOME SECRETARY  *
120 REM   *********************
130 REM
140 REM
170 DIM A1$(60),A2$(60),A3$(60),A4$(60
    ),A5$(60)
180 DIM CAT$(6)
190 DIM P1(3),P2(3)
200 DATA 697,770,852,1210,1340,1481
210 READ P1(1),P1(2),P1(3),P2(1),P2(2)
    ,P2(3)
```

```
220 DATA NAME:,PHONE:,STREET:,CITY&ZIP
    :,MISC:
230 DATA *ITEM:          *,COST:,SHOP:,WHEN
    :,MISC:
240 CALL CLEAR
250 LSIZE=0
260 OPT=1
270 READ CAT$(1),CAT$(2),CAT$(3),CAT$(
    4),CAT$(5)
280 PRINT "LOAD DATA? (Y/N)"
290 GOSUB 3120
300 IF KEY<>89 THEN 330
```

```
310 GOSUB 1900
320 GOTO 410
330 REM  NEW SET UP
340 PRINT "PHONE BOOK? (Y/N)"
350 GOSUB 3120
360 IF KEY=89 THEN 390
370 OPT=2
380 READ CAT$(1),CAT$(2),CAT$(3),CAT$(
    4),CAT$(5)
390 N=0
400 GOSUB 430
410 GOSUB 850
420 GOTO 410
430 REM  KEY INPUT FOR DATA SET UP
440 PRINT :"  ENTER":"          E TO EXIT
    ":"  R TO REENTER"
450 FLAG1=1
460 FLAG2=1
470 I=N+1
480 IF I<=60 THEN 520
490 PRINT :::"  *** ARRAY FULL (N=60)**
    *":::
500 GOSUB 3100
510 RETURN
520 PRINT
530 INPUT CAT$(1):A1$(I)
540 IF A1$(I)="E" THEN 750
550 IF A1$(I)="END" THEN 750
560 IF A1$(I)="" THEN 530
570 IF A1$(I)<>"R" THEN 620
580 I=I-1
590 N=I-1
600 PRINT :"  *** REENTER LAST SET ***
    "
610 GOTO 520
620 IF OPT<>1 THEN 650
630 INPUT CAT$(2):A2$(I)
640 GOTO 670
650 INPUT CAT$(2):T
660 A2$(I)=STR$(T)
670 INPUT CAT$(3):A3$(I)
680 INPUT CAT$(4):A4$(I)
690 INPUT CAT$(5):A5$(I)
700 GOSUB 770
710 IF T>190 THEN 600
720 GOSUB 800
730 N=I
740 GOTO 470
750 GOSUB 2410
760 RETURN
770 REM  MEMORY CHECK
780 T=LEN(A1$(I)&A2$(I)&A3$(I)&A4$(I)&
    A5$(I))
790 RETURN
800 LSIZE=LSIZE+T
810 IF LSIZE<5300 THEN 840
820 CALL SOUND(200,800,4)
830 PRINT :"** WARNING ** MEMORY GETTI
    NG FULL":"  (LSIZE=";STR$(LSIZE)&"
    )"
840 RETURN
850 SC=4
860 GOSUB 3060
870 PRINT "PRESS:::"1 - TO ADD MORE D
    ATA":::"2 - TO ALTER THE DATA"
880 PRINT :"3 - TO DISPLAY THE DIRECTO
    RY":::"4 - TO DISPLAY ONE ENTRY":::
890 PRINT :"5 - TO USE THE DATA":::"6 -
    TO STORE DATA FILE":::"7 - FOR PRI
    NTER LISTING"
900 GOSUB 3060
910 IF KEY<49 THEN 900
920 IF KEY>55 THEN 900
930 ON KEY-48 GOSUB 990,1010,1430,1560
    ,3150,2140,4280,4450
940 RETURN
950 IF KEY>56 THEN 850
960 GOSUB 3060
970 ON (KEY-48)GOSUB 990,1010,1430,156
    0,3150,2140,4280,4450
980 RETURN
990 GOSUB 430
1000 RETURN
1010 REM  DATA ALTERATION
1020 INPUT "WHICH ONE? ":M$
1030 IF M$="" THEN 1410
1040 PRINT :::"  ENTER":::"  NEW DATA
     AT CURSOR":::"  'D' TO DELETE TH
     E ITEM":::"  'ENTER' FOR NO CHAN
     GES":::
1050 GOSUB 1800
1060 PRINT :::
1070 FLAG1=1
1080 T1$="?  "
1090 I=M
1100 GOSUB 770
1110 T=-T
1120 GOSUB 800
1130 INPUT A1$(M)&T1$:TMP$
1140 IF TMP$="" THEN 1220
1150 IF TMP$<>"D" THEN 1200
1160 A1$(M)="  "
1170 GOSUB 2410
1180 N=N-1
1190 RETURN
1200 A1$(M)=TMP$
1210 FLAG2=1
1220 INPUT A2$(M)&T1$:TMP$
1230 IF TMP$="" THEN 1250
1240 A2$(M)=TMP$
1250 INPUT A3$(M)&T1$:TMP$
1260 IF TMP$="" THEN 1280
1270 A3$(M)=TMP$
1280 INPUT A4$(M)&T1$:TMP$
1290 IF TMP$="" THEN 1310
1300 A4$(M)=TMP$
1310 INPUT A5$(M)&T1$:TMP$
1320 IF TMP$="" THEN 1340
1330 A5$(M)=TMP$
1340 GOSUB 770
1350 IF T<192 THEN 1380
1360 PRINT :"** REENTER LAST SET **"
1370 GOSUB 1130
1380 GOSUB 800
1390 GOSUB 1650
1400 ON T GOTO 1130,1020,1410
1410 RETURN
1420 REM  DISPLAY ENTIRE DIRECTORY
1430 IF FLAG2=0 THEN 1450
1440 GOSUB 2410
1450 FOR I=1 TO N
1460 M=29-LEN(A2$(I))
1470 T$=STR$(I)&".  "
1480 PRINT TAB(4-LEN(T$));T$;A1$(I);TAB
     (M);A2$(I)
1490 IF I=20 THEN 1520
1500 IF I=40 THEN 1520
1510 GOTO 1530
1520 GOSUB 3100
1530 NEXT I
1540 GOSUB 3100
1550 RETURN
1560 REM  SINGLE ITEM LISTING
1570 INPUT "WHICH ONE? ":M$
1580 IF M$="" THEN 1640
1590 GOSUB 1800
1600 GOSUB 3060
1610 PRINT A1$(M)::A2$(M)::A3$(M)::A4$(
     M)::A5$(M)::
1620 GOSUB 1650
1630 ON T GOTO 1600,1640
1640 RETURN
1650 GOSUB 3100
1660 T=3
1670 IF KEY<>69 THEN 1720
1680 T=1
1690 IF M=1 THEN 1780
1700
```

```
1710 M=M-1
1720 IF KEY<>88 THEN 1760
1730 T=1
1740 IF M=N THEN 1780
1750 M=M+1
1760 IF KEY<>83 THEN 1780
1770 T=2
1780 RETURN
1790 REM SEARCH ROUTINE FOR SINGLE ITE
     M LISTING
1800 IF ABS(ASC(M$)-53)>4 THEN 1850
1810 M=VAL(M$)
1820 IF M<=N THEN 1840
1830 M=N
1840 RETURN
1850 FOR I=1 TO N
1860 M=I
1870 IF M$<=A1$(I)THEN 1890
1880 NEXT I
1890 RETURN
1900 REM LOAD DATA
1910 PRINT "ENTER":::"1. CS1":::"2. DISK1
     ":::"3. OTHER"
     INPUT DEV
1930 IF DEV<>1 THEN 1960
1940 DEV$="CS1"
1950 GOTO 2010
1960 IF DEV<>2 THEN 2000
1970 INPUT "ENTER FILE NAME:":FIL$
1980 DEV$="DSK1."&FIL$
1990 GOTO 2010
2000 INPUT "ENTER DEVICE NAME:":DEV$
2010 OPEN #2:DEV$,INTERNAL,INPUT ,FIXED
     192
2020 INPUT #2:OPT,N,FIL$,DATE$,LSIZE
2030 IF OPT=1 THEN 2050
2040 READ CAT$(1),CAT$(2),CAT$(3),CAT$(
     4),CAT$(5)
2050 PRINT :::FIL$:::"LSIZE(3800)=";LSIZE
     :::"LAST UPDATE: ";DATE$:::
2060 FOR I=1 TO N
2070 INPUT #2:A1$(I),A2$(I),A3$(I),A4$(
     I),A5$(I)
2080 NEXT I
2090 IF DEV=1 THEN 2120
2100 FOR TD=1 TO 1000
2110 NEXT TD
2120 CLOSE #2
2130 RETURN
2140 REM SAVE DIRECTORY
2150 IF FLAG2=0 THEN 2170
2160 GOSUB 2410
2170 PRINT "ENTER 1. CS1"
2180 PRINT "          2. DSK1"
2190 PRINT "          3. OTHER":::::
2200 INPUT "YOUR CHOICE?":ANS
2210 IF (ANS<1)+(ANS>3)THEN 2170
2220 ON ANS GOTO 2230,2250,2310
2230 DEV$="CS1"
2240 GOTO 2320
2250 INPUT "ENTER FILE NAME.":NAM$
2260 IF LEN(NAM$)<11 THEN 2290
2270 PRINT "ENTER NO MORE THAN TEN
     LETTERS PLEASE."
2280 GOTO 2170
2290 DEV$="DSK1."&NAM$
2300 GOTO 2320
2310 INPUT "ENTER DEVICE NAME.":DEV$
2320 INPUT "ENTER DATE.":DATE$
2330 OPEN #3:DEV$,INTERNAL,OUTPUT,FIXED
     192
2340 PRINT #3:OPT,N,FIL$,DATE$,LSIZE
2350 FOR I=1 TO N
2360 PRINT #3:A1$(I),A2$(I),A3$(I),A4$(
     I),A5$(I)
2370 NEXT I
2380 CLOSE #3
2390 FLAG1=0
2400 RETURN
2410 REM SORTING ROUTINE
2420 FLAG2=0
2430 CALL SOUND(100,800,6)
2440 PRINT ::::"  *** SORTING DATA ****
     *":::
2450 IF (N-1)THEN 2470
2460 RETURN
2470 FOR I=1 TO N
2480 NEXT I
2490 N2=INT(N/2)
2500 N21=N2+2
2510 ICT=1
2520 I=2
2530 N1=N21-I
2540 NN=N
2550 IK=N1
2560 GOSUB 2880
2570 JK=2*IK
2580 IF JK>NN THEN 2660
2590 IF JK=NN THEN 2620
2600 IF A1$(JK+1)<=A1$(JK)THEN 2620
2610 JK=JK+1
2620 IF A1$(JK)<=C$ THEN 2660
2630 GOSUB 2940
2640 IK=JK
2650 GOTO 2570
2660 GOSUB 3000
2670 ON ICT GOTO 2680,2780
2680 IF I>=N2 THEN 2710
2690 I=I+1
2700 GOTO 2530
2710 ICT=2
2720 NP2=N+2
2730 I=2
2740 N1=NP2-I
2750 NN=N1
2760 IK=1
2770 GOTO 2560
2780 IK=1
2790 GOSUB 2880
2800 IK=N1
2810 GOSUB 2940
2820 IK=N1
2830 GOSUB 3000
2840 IF I>=N THEN 2870
2850 I=I+1
2860 GOTO 2740
2870 RETURN
2880 C$=A1$(IK)
2890 M$=A2$(IK)
2900 T$=A3$(IK)
2910 TMP$=A4$(IK)
2920 TI$=A5$(IK)
2930 RETURN
2940 A1$(IK)=A1$(JK)
2950 A2$(IK)=A2$(JK)
2960 A3$(IK)=A3$(JK)
2970 A4$(IK)=A4$(JK)
2980 A5$(IK)=A5$(JK)
2990 RETURN
3000 A1$(IK)=C$
3010 A2$(IK)=M$
3020 A3$(IK)=T$
3030 A4$(IK)=TMP$
3040 A5$(IK)=TI$
3050 RETURN
3060 REM CLEAR & SET SCREEN
3070 CALL CLEAR
3080 CALL SCREEN(SC)
3090 RETURN
3100 REM KEY RETURN
3110 PRINT :::"PRESS ANY KEY TO CONTINUE
     "
3120 CALL KEY(0,KEY,STATUS)
3130 IF STATUS=0 THEN 3120
3140 RETURN
3150 REM UTILITY PROGRAMS
3160 IF OPT=1 THEN 3290
3170 SUM=0
```

```
3180 FOR I=1 TO N
3190 SUM=SUM+VAL(A2$(I))
3200 NEXT I
3210 PRINT "TOTAL COST OF ALL THE ITEMS
     ":::::
3220 PRINT :::::TAB(10);SUM:::::::::
3230 CALL HCHAR(11,7,36,18)
3240 CALL HCHAR(19,7,36,18)
3250 CALL VCHAR(12,7,36,7)
3260 CALL VCHAR(12,24,36,7)
3270 GOSUB 3100
3280 RETURN
3290 REM  DIAL PHONE
3300 REM  CLOCK TIME DELAYS FOLLOW
3310 RESTORE 3320
3320 DATA 2,57,55,53,51
3330 READ D1,D2,D3,D4,D5
3340 PRINT :::"YOU WANT ME TO DIAL":::
3350 GOSUB 1570
3360 IF MS="" THEN 3610
3370 CALL CLEAR
3380 H=23
3390 V=8
3400 T$=A2$(M)
3410 L=LEN(T$)
3420 PRINT A1$(M):A2$(M):A3$(M):A4$(M):
     A5$(M)::::
3430 IF L<10 THEN 3460
3440 L=L+1
3450 T$="1"&T$
3460 FOR I=1 TO L
3470 TMP$=SEG$(T$,I,1)
3480 CALL HCHAR(H,V,ASC(TMP$),1)
3490 V=V+1
3500 IF ASC(TMP$)<48 THEN 3600
3510 IF ASC(TMP$)>57 THEN 3600
3520 T=VAL(TMP$)
3530 IF T<>0 THEN 3560
3540 CALL SOUND(300,941.0,1336,2)
3550 GOTO 3590
3560 I=INT((T-1)/3)+1
3570 IJ=T-3*(I-1)
3580 CALL SOUND(300,P1(I),0,P2(IJ),2)
3590 CALL SOUND(250,44000,29)
3600 NEXT I
3610 PRINT :::"PRESS ":"R TO REDIA
     L":" S TO START STOPWATCH"::
                N FOR NEW NUMBER"
3620 GOSUB 3100
3630 IF KEY=82 THEN 3370
3640 IF KEY=78 THEN 3350
3650 IF KEY<>83 THEN 3690
3660 PRINT :"HOLD DOWN ":"R TO DIAL AGA
     IN":"ANY KEY TO CONTINUE":::
3670 GOSUB 3700
3680 IF T=82 THEN 3290
3690 RETURN
3700 REM  STOP WATCH
3710 H=23
3720 V=23
3730 JS=1
3740 S=4
3750 DELAY=D1
3760 FOR J1=0 TO 16
3770 A1=48+J1
3780 FOR J2=0 TO 9
3790 A2=48+J2
3800 FOR J3=0 TO 5
3810 A3=48+J3
3820 FOR J4=JS TO 9
3830 A4=48+J4
3840 GOSUB 4020
3850 IF STATUS<>0 THEN 4010
3860 S=4
3870 DELAY=D2
3880 NEXT J4
3890 JS=0
3900 DELAY=D3
3910 NEXT J3
3920 DELAY=D4
3930 S=11
3940 NEXT J2
3950 A2=48
3960 DELAY=D5
3970 S=10
3980 GOSUB 4020
3990 NEXT J1
4000 GOTO 3760
4010 RETURN
4020 REM  COMPENSATED CLOCK
4030 CALL HCHAR(H,V+2,32,1)
4040 CALL KEY(0,T,STATUS)
4050 FOR I=1 TO DELAY
4060 IF STATUS<>0 THEN 4010
4070 NEXT I
4080 CALL SCREEN(S)
4090 CALL HCHAR(H,V,A1,1)
4100 CALL HCHAR(H,(V+1),A2,1)
4110 CALL HCHAR(H,V+2,58,1)
4120 CALL HCHAR(H,V+3,A3,1)
4130 CALL HCHAR(H,V+4,A4,1)
4140 RETURN
4150 REM  SCREEN PRINTING
4160 IF RP=0 THEN 4270
4170 OPEN #1:"RS232"
4180 FOR I=1 TO 20
4190 PRINT #1:TS
4200 PTS=""
4210 FOR I=1 TO 32
4220 CALL GCHAR(I,J,CH)
4230 TS=TS&CHR$(CH)
4240 NEXT J
4250 NEXT I
4260 CLOSE #1
4270 RETURN
4280 REM  COMPLETE LISTING ON PRINTER
4290 INPUT "ENTER DEVICE NAME.":DEVS
4300 PRINT "IS ";DEVS;" READY?(Y/N)"
4310 GOSUB 3120
4320 IF KEY<>89 THEN 4440
4330 OPEN #3:DEVS,OUTPUT
4340 T=INT(N/9)
4350 FOR J=0 TO T
4360 FOR M=1 TO 9
4370 I=9*J+M
4380 PRINT #3:TAB(5);I;".");TAB(10);A1$(
     I);TAB(10);A2$(I);TAB(10);A3$(I);T
     AB(10);A4$(I);TAB(10);A5$(I)::
4390 IF I=N THEN 4430
4400 NEXT M
4410 GOSUB 3100
4420 NEXT J
4430 CLOSE #3
4440 RETURN
4450 PRINT "DO YOU WISH TO HALT THE
     PROGRAM AND LOSE ALL DATA INMEMOR
     Y?(Y/N)"
4460 CALL KEY(0,K,S)
4470 IF S=0 THEN 4460
4480 IF K=ASC("N") THEN 850
4490 IF K<>ASC("Y") THEN 4440
4500 END
```



A Speech Vocabulary Expansion Aid

EXTENDED BASIC

Verbose is a program that was written in an evolutionary manner. One thing just lead to another. The story goes something like this:

One day I decided to make a program speak a simple sentence. After all, the TI Speech Synthesizer must have something to say. Well, anyway, I came up with a simple sentence—don't remember what it was now—in a program which I entered and ran.

Wow—almost half of the words in the sentence were not in the resident vocabulary! It was clearly time for me to read the manual that came with the unit. Surprise. I found it had a vocabulary limited to three or four hundred words. That was not enough for me. Further research was definitely called for.

Reading the TI Extended BASIC manual, I found a program on page 206 that allowed adding standard suffixes to resident vocabulary words (e.g.,-ed, -ing, -s). After playing with this suffix program awhile, I realized it would be possible to *concatenate* two resident vocabulary words to produce a totally new word: therefore, meanwhile, or update. I wrote a routine to do this. Once this concatenation routine was working, it seemed like a speech tool starting to evolve.

It would be nice, I thought, to have the results of the concatenation routine printed in the form of DATA statements. I could then write these DATA statements containing the new word's speech data into other programs that needed to speak the new word. So, I generated a routine to do this, and added it to the concatenation routine.

All of these routines, including a method of building a vocabulary file on disk, were combined into a nice, neat, simple-to-use program. The result is Listing 4. As you can see from the listing, I originally called the program *Word Builder*. When I decided to write an article on it, however, the name seemed too mundane. So in a fit of cleverness, I renamed the program *Verbose*. My wife and my friends just shook their heads and groaned. . .

A TV picture is worth a thousand words, right? Well, perhaps not quite, so I have combined some text with screen images to guide you through the operation of *Verbose*.

Before you start the *Verbose* program, make sure you have either the *TI Extended BASIC* or *TI Speech Editor* Command Cartridge plugged in. *Verbose* uses the SPGET and SAY subroutines that are available in these modules. OK, now you're ready to load *Verbose* and type RUN.

```
+ + + WORD BUILDER + + +
ENTER NUMBER OF YOUR CHOICE
1 - JOIN TWO WORDS
2 - PRINT SPEECH DATA
3 - STORE NEW WORD ON DISK
4 - EXIT
?
```

Here we are at the main menu screen. Let's create a new word by joining two words. The new word that we will generate will be *rewrite* and will be made from vocabulary words *read* and *right*. Type 1 and press the ENTER key.

```
ENTER FIRST WORD JOIN
?
```

We are asked for the first word that will be used in the joining. Type READ and press ENTER.

```
ENTER SECOND WORD TO JOIN
?
```

Now type the word RIGHT and press ENTER.

```
ENTER THE SPELLING OF THE NEW WORD
?
```

Type in REWRITE and then press ENTER.

```
TRUNCATE HOW MANY BYTES?
```

OK, don't panic here! *Verbose* just wants to know how much of the first word (READ) to truncate before it com-

bines it with the second word (RIGHT). We don't know how much, so we make a wild guess of, say, 34. What we want is to truncate the AD from READ and combine that sound with RIGHT. As soon as you press ENTER this time, the TI-99/4A will say the new word for you.

```
SAY AGAIN? (Y OR N)
```

Here you can answer the question with Y as many times as you like to check the sound of the new word. After hearing enough of it, enter N.

```
SAY AGAIN? (Y OR N) N
1-CHANGE SOME MORE
2-BACK TO MAIN MENU
?
```

If you don't think the new word sounded quite right, type 1 and press ENTER.

```
TRUNCATE HOW MANY BYTES?
```

This time type 55 and press ENTER.

Listen to the word as many times as you like. With 55 bytes truncated, it sounds to me close enough to use. When you are satisfied, return to the main menu.

```
+ + + WORD BUILDER + + +
ENTER NUMBER OF YOUR CHOICE
1-JOIN TWO WORDS
2-PRINT SPEECH DATA
3-STORE NEW WORD ON DISK
4-EXIT
?
```

Here we are back at the ranch. Let's print the data for our new word by selecting option 2. Don't forget to press ENTER. (I'm not going to remind you about that anymore 'cause you've got the ENTER key down pat.)

```
ENTER THE WORD WHOSE DATA YOU WANT TO
PRINT - -
?
```

After you enter REWRITE and press the you-know-what, the printer will output what you see in Listing 1. It didn't work? Well your printer must be set up differently from mine. Go to Listing 4 and modify line 870 of *Verbose* (the OPEN statement for the printer) to match your setup. If you don't have a printer, delete lines 870 and 1070. Also modify lines 940, 950, 990, and 1060 by deleting the "#1:" of each print statement. Now, instead of going to the printer, everything will go to the screen of the TI-99/4A. The last change is to enter this line:

```
1070 INPUT F$
```

Now it will stay on the screen (so you can copy it on paper) until you press ENTER.

Look over Listing 2. This is a sample TI BASIC program that shows how the DATA statements for *Verbose* can be used. You will note the DATA statements for the word REWRITE are entered in lines 360-490 of Listing 2. Lines 280-330 build the string E$ which will cause REWRITE to be spoken. The FOR-NEXT loop here is terminated when the last byte is read. The loop counter limit (133) was the number of bytes printed out for REWRITE by *Verbose*. The subroutines SAY and SPGET are explained in the speech synthesizer manual.

```
+ + + WORD BUILDER + + +
ENTER NUMBER OF YOUR CHOICE
1-JOIN TWO WORDS
2-PRINT SPEECH DATA
3-STORE NEW WORD ON DISK
4-EXIT
?
```

It is very tiring to enter all those DATA statements of the previous sample program. For those of you with a disk system, an easier method of saving and using words from *Verbose* is available with option 3. Go ahead and select it now.

```
PUT THE DISK WITH "WORDS"
FILE IN DRIVE ONE.
PRESS ENTER WHEN READY
```

The disk on which you wish to keep your new vocabulary words should now be placed in disk drive 1. The words that will be saved will be appended to a file called WORDS on this diskette. See line 1160 of Listing 4 for the OPEN statement for this file.

```
PUT THE DISK WITH "WORDS"
FILE IN DRIVE ONE.
PRESS ENTER WHEN READY

ENTER THE WORD WHOSE DATA YOU WANT TO
SAVE - -
?
```

Enter the word REWRITE to save. The disk drive will run and then *Verbose* will return to its main menu. Use this option to save a few more words that you choose. Then run the *Spelling Test Game* in Listing 3 using the resultant WORDS file.

The *Spelling Test Game* program will accept up to 20 words for the WORDS file. It then speaks each word, checks the spelling that is input, and keeps score. Any children in your home should find it useful for spelling drill.

Study Listing 3 and notice lines 230-270. The WORDS file has a pair of strings for each word saved. The first string contains the spelling of the word. The second string contains the actual speech data.

As mentioned earlier, the program listing for *Verbose* is Listing 4.

A final note of caution: Once you start that TI-99/4A talking, BEWARE—you may have trouble getting a word in edgewise. . .

Listing 1

```
THE WORD IS ** REWRITE **
LENGTH = 133 BYTES

DATA 96,0,42,161,19,49,92,60,149,149
DATA 78,86,51,117,147,223,26,61,196,197
DATA 69,253,170,93,103,231,176,108,167,10
DATA 158,83,211,151,156,188,40,21,157,106
DATA 180,178,42,89,125,96,0,85,162,101
DATA 33,221,57,28,139,154,142,144,176,116
DATA 172,106,58,92,162,67,137,105,248,82
DATA 142,49,39,169,209,7,179,84,220,175
DATA 218,196,26,103,157,119,235,83,133,156
DATA 233,220,113,110,117,170,88,51,77,58
DATA 238,169,211,240,100,207,186,167,201,69
DATA 196,162,42,205,46,245,41,179,68,87
DATA 97,51,24,105,146,233,22,0,64,1
DATA 93,121,60
```

Listing 2

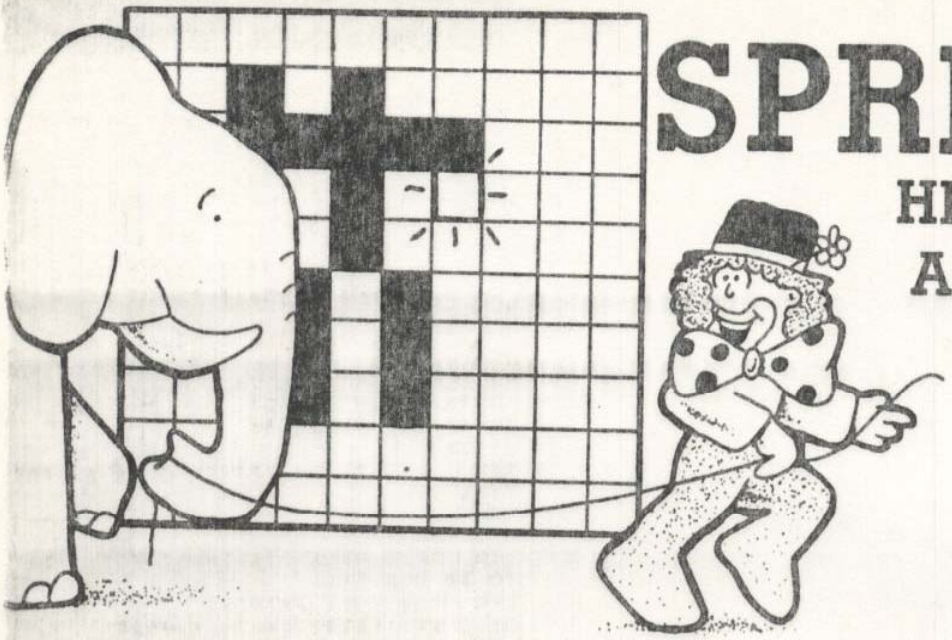
```
100 REM  ++++++++++++++++++++++++++++++++
110 REM  +   SPEAK-- 'THIS              +
120 REM  +   PROGRAM HAD A              +
130 REM  +      REWRITE."                +
140 REM  +                              +
150 REM  +                              +
160 REM  +THIS SAMPLE SHOWS             +
170 REM  +HOW THE OUTPUT OF             +
180 REM  +"VERBOSE" IS USED             +
190 REM  +IN TI BASIC....               +
200 REM  +                              +
210 REM  ++++++++++++++++++++++++++++++++
220 REM
230 REM
240 CALL SPGET("THIS",A$)
250 CALL SPGET("PROGRAM",B$)
260 CALL SPGET("HAD",C$)
270 CALL SPGET("A",D$)
280 RESTORE 360
290 E$=""
300 FOR I=1 TO 133
310 READ X
320 E$=E$&CHR$(X)
330 NEXT I
340 CALL SAY("",A$,"",B$,"",C$,"",D$,"",E$)
350 REM  ** REWRITE **
360 DATA 96,0,42,161,19,49,92,60,149,1
    49
370 DATA 78,86,51,117,147,223,26,61,19
    6,197
380 DATA 69,253,170,93,103,231,176,108
    ,167,10
390 DATA 158,83,211,151,156,188,40,21,
    157,106
400 DATA 180,178,42,89,125,96,0,85,162
    ,101
410 DATA 33,221,57,28,139,154,142,144,
    176,116
420 DATA 172,106,58,92,162,67,137,105,
    248,82
430 DATA 142,49,39,169,209,7,179,84,22
    0,175
440 DATA 218,196,26,103,157,119,235,83
    ,133,156
450 DATA 233,220,113,110,117,170,88,51
    ,77,58
460 DATA 238,169,211,240,100,207,186,1
    67,201,69
470 DATA 196,162,42,205,46,245,41,179,
    68,87
480 DATA 97,51,24,105,146,233,22,0,64,
    1
490 DATA 93,121,60
```

Listing 3

```
100 REM  ++++++++++++++++++++++++++++++++
110 REM  +   SPELLING TEST GAME         +
120 REM  ++++++++++++++++++++++++++++++++
130 REM
140 REM
150 REM  ++++++++++++++++++++++++++++++++
160 REM  USES "DSK1.WORDS" FILE AS SO
    URCE OF WORDS TO
170 REM  GUESS IN GAME.
180 DIM WORD$(20),F$(20)
190 CALL CLEAR
200 PRINT "PUT DISK WITH ""WORDS"" FIL
    E IN DRIVE ONE"
210 INPUT "PRESS ENTER WHEN READY":X$
220 OPEN #1:"DSK1.WORDS",INTERNAL,INPU
    T,VARIABLE 254
230 FOR I=1 TO 20
240 IF EOF(1)<>0 THEN 300
250 INPUT #1:WORD$(I)
260 INPUT #1:F$(I)
270 NEXT I
280 LAST=I
290 GOTO 310
300 LAST=I-1
310 CLOSE #1
320 REM
330 CALL CLEAR
340 SCORE=0
350 PRINT "THERE ARE ";LAST;"WORDS":::
360 PRINT "SEE IF YOU CAN SPELL THEM
    ALL CORRECTLY.  GOOD LUCK!"
370 FOR M=1 TO 700
380 NEXT M
390 FOR J=1 TO LAST
400 CALL CLEAR
410 PRINT "WORD #";J:::
420 CALL SAY("",F$(J))
430 INPUT "SPELL IT-":X$
440 IF X$=WORD$(J) THEN 470
450 CALL SAY("UHOH")
460 SCORE=SCORE-1
470 SCORE=SCORE+1
480 PRINT :::"CORRECT SPELLING IS
    >";WORD$(J);"<":::
490 PRINT "YOUR SCORE IS ";SCORE;" OUT
    OF";J
500 FOR Y=1 TO 500
510 NEXT Y
520 NEXT J
530 CALL CLEAR
540 PRINT "YOU GOT ";INT((SCORE/LAST)*
    100);"% CORRECT":::::
550 INPUT "ENTER 'Y' TO TRY AGAIN":
    Z$
560 IF Z$="Y" THEN 330
570 CALL CLEAR
580 END
```

Listing 4

```
100 REM  +++++++++++++++++++++++++++++
110 REM  +          V E R B O S E    +
120 REM  +                           +
130 REM  +++++++++++++++++++++++++++++
140 REM
150 REM
160 REM
170 REM
180 REM
190 REM   SPEECH MAKER  ROUTINE
200 REM  WILL COMBINE WORDS INTO NEW S
    TRINGS.
210 REM
220 CALL CLEAR
230 PRINT "     +++ WORD BUILDER +++":::
    ::
240 PRINT "ENTER NUMBER OF YOUR CHOICE
    ":::
250 PRINT :" 1 - JOIN TWO WORDS":::
260 PRINT " 2 - PRINT SPEECH DATA":::
270 PRINT " 3 - STORE NEW WORD ON DIS
    K":::
280 PRINT " 4 - EXIT":::
290 INPUT CHOICE
300 IF (CHOICE<1)+(CHOICE>4)=-1 THEN 2
    20
310 ON CHOICE GOSUB 550,870,1100,330
320 GOTO 220
330 CALL CLEAR
340 END
350 REM  +++ TRUNCATE FIRSTWORD +++
360 CALL CLEAR
370 INPUT "TRUNCATE HOW MANY BYTES?":B
    YTES
380 MAXBYTES=LEN(B$)-3
390 IF BYTES<MAXBYTES THEN 420
400 PRINT "TOO MANY BYTES....."
410 GOTO 370
420 IF BYTES>-1 THEN 450
430 PRINT "NO NEGATIVE NUMBERS"
440 GOTO 370
450 L=MAXBYTES-BYTES
460 C$=SEG$(B$,1,2)&CHR$(L)&SEG$(B$,4,
    L)
470 RETURN
480 REM  +++ SPEAK NEW WORD +++
490 CALL CLEAR
500 CALL SAY(".",NEWDATA$)
510 INPUT "SAY AGAIN? (Y OR N)":CHOICE
    $
520 IF CHOICE$="Y" THEN 500
530 RETURN
540 REM  +++ JOIN TWO WORDS SUBROUTIN
    E +++
550 CALL CLEAR
560 PRINT "ENTER FIRST WORD TO JOIN"
570 INPUT FIRSTWORD$
580 IF FIRSTWORD$=LASTMADE$ THEN 610
590 CALL SPGET(FIRSTWORD$,B$)
600 GOTO 620
610 B$=LASTDATA$
620 CALL CLEAR
630 PRINT "ENTER SECOND WORD TO JOIN"
640 INPUT SECONDWORD$
650 IF SECONDWORD$=LASTMADE$ THEN 680
660 CALL SPGET(SECONDWORD$,D$)
670 GOTO 690
680 D$=LASTDATA$
690 CALL CLEAR
700 PRINT "ENTER THE SPELLING OF THE
    NEW WORD"
710 INPUT NEWWORD$
720 REM
730 GOSUB 350
740 NEWDATA$=C$&D$
750 GOSUB 480
760 PRINT "     1 - CHANGE SOME MORE":"
    2 - BACK TO MAIN MENU"
770 INPUT CHOICE
780 IF (CHOICE<1)+(CHOICE>2)=-1 THEN 7
    60
790 IF CHOICE=1 THEN 730
800 LASTMADE$=NEWWORD$
810 LASTDATA$=NEWDATA$
820 RETURN
830 REM  +++ PRINT SPEECH DATA SUBROU
    TINE +++
840 REM  ------------------------------
850 REM  THIS OPEN STATEMENT MAY NEED
    TO BE MODIFIED
860 REM  FOR YOUR PRINTER.....
870 OPEN #1:"RS232.DA=8.BA=9600"
880 REM  ------------------------------
890 CALL CLEAR
900 PRINT "ENTER THE WORD WHOSE DATA
    YOU WANT TO PRINT--     ":::
910 GOSUB 1230
920 IF L=0 THEN 1070
930 VALUES$=""
940 PRINT #1:::"THE WORD IS ** ";WORD$
    ;" **":
950 PRINT #1:"LENGTH =";L;"BYTES"::
960 FOR I=1 TO L
970 VALUES$=VALUES$&STR$(ASC(SEG$(F$,I
    ,1)))
980 IF I/10<>INT(I/10)THEN 1020
990 PRINT #1:"DATA ";VALUES$
1000 VALUES$=""
1010 GOTO 1030
1020 VALUES$=VALUES$&","
1030 NEXT I
1040 IF VALUES$="" THEN 1070
1050 VALUES$=SEG$(VALUES$,1,LEN(VALUES$
    )-1)
1060 PRINT #1:"DATA ";VALUES$
1070 CLOSE #1
1080 RETURN
1090 REM  +++ ADD NEW WORD TO VOCABULA
    RY FILE +++
1100 CALL CLEAR
1110 PRINT "PUT THE DISK WITH ""WORDS""
     FILE IN DRIVE ONE."
1120 INPUT "PRESS ENTER WHEN READY ":X$
1130 PRINT :::"ENTER THE WORD WHOSE DAT
    A YOU WANT TO SAVE--     ":::
1140 GOSUB 1230
1150 IF L=0 THEN 1200
1160 OPEN #1:"DSK1.WORDS",INTERNAL,APPE
    ND,VARIABLE 254
1170 PRINT #1:WORD$
1180 PRINT #1:F$
1190 CLOSE #1
1200 RETURN
1210 REM
1220 REM  +++ FIND WORD SUBROUTINE
1230 INPUT WORD$
1240 F$=""
1250 IF WORD$="" THEN 1300
1260 IF WORD$=LASTMADE$ THEN 1290
1270 CALL SPGET(WORD$,F$)
1280 GOTO 1300
1290 F$=LASTDATA$
1300 L=LEN(F$)
1310 RETURN
```



SPRITER

HIGH-SPEED ANIMATION WITH SPRITES

EXTENDED BASIC

TI Extended BASIC lets you fill the screen with rapidly moving sprites of many colors. See for example *Sprite Chase* in "Computer Gaming." Although the smooth and rapid motion possible with sprites is indeed quite impressive and arcade-like, think how much more spectacular these screen displays would be if we animated the moving sprites: After all, why just move a man-shaped sprite when you can also move his arms and legs? Picture the visual impact of a bird-sprite flying across the screen flapping its wings. How about a circus parade with clowns tumbling, animals walking, and elephants moving their trunks? All of this—and more—is possible with sprite animation.

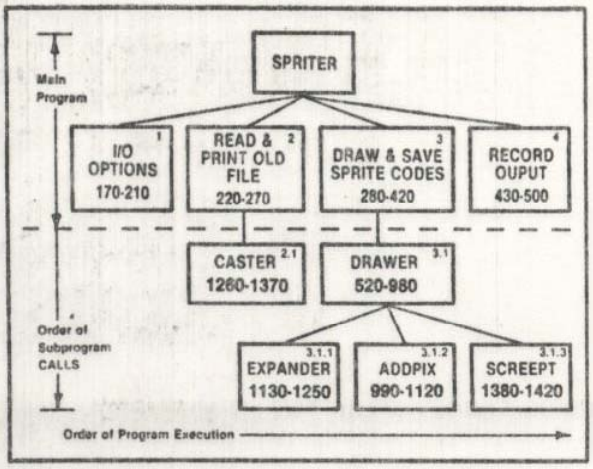
The technique of animation is old and well-known. First we draw a series of figures with each figure in a slightly different position and posture. Then, we rapidly flash the figures one after the other on the screen, and *persistence of vision* goes to work—fooling our eyes and causing us to see figures move as if alive. Now with sprites we can duplicate this movie animation technique on the TI-99/4 and 99/4A through simple commands in Extended BASIC. [See "3-D Animation with the TMS9918A Video Chip."—Ed.]

The usual trouble with computer animation is the tedious series of tasks that you have to do after drawing the figures: You have to figure out, keep track of, and key in those long pattern identifiers. If you have chosen to work with sprites that are four characters large, these codes then become 64 hexadecimal characters long! This situation prompted me to write *Spriter* (Listing 1), a program that does much of the work for you, and leaves you free to concentrate on the fun—drawing the figures for the animation sequence. *Spriter* automatically computes, files, and saves an array of four-character pattern identifiers that define sprites of magnification 3 or 4 (figurines). After you draw each figurine you can output a model of it to the thermal printer (optional) and when you are finished, you can save the whole file on cassette tape or disk.

When you run *Spriter*, it presents you with a 16 × 16 character work area in the screen's character display field. Under your direction, the computer generates an enlarged model of the figurine within the work area. The image is made up of dark and bright character squares, each of which has a counterpart in the figurine. Changes in the display field are automatically converted into changes of the figurine's pattern identifier. The figurines in the computer's memory (RAM) can be stored permanently on tape or disk and later accessed by either *Spriter* or any other program with animation recall capability. See, for example, the animation demonstration program in Listing 2. *Spriter* thus allows you to generate new figurines, transfer any figurine that you have stored on tape into RAM, and rework any figurine that is present in RAM.

How to Run *Spriter*

Instructions are almost self-contained: A series of prompts guides you through much of the program. First, you are asked if you have a thermal printer and if you want to input a file of characters from tape or disk. If so, you are asked the corresponding file name. Then the work area is framed on the screen. If you have chosen to show an existing figurine by reading in an old file, *Spriter* copies that figurine to the work area. When the cursor appears in the upper lefthand corner of the work area, you are ready to draw a new figurine or redraw an existing one. You can move the cursor anywhere within the work area by using the arrow keys for horizontal and vertical motion, and the W, R, C, and Z keys for diagonal motions. When the cursor is moved, it automatically leaves a trace as determined by the polarity keys: bright if the A key was pressed and dark if the F key was previously pressed. When the cursor first appears, the polarity of the trace is dark. Afterwards, by using the motion and polarity keys you can draw and erase portions of the model until you are satisfied with the results. Then press the Q key to exit the drawing mode. A new series of prompts will guide you through the rest of the program.



How *Spriter* Works

Space does not allow a line-by-line description of *Spriter* (see Listing 1). But for those interested in exploring the intricacies of the program, I have provided a road map in the form of a structure diagram (Figure 1). Functions identified within the main program are depicted as boxes above the dashed line; those identified with subprograms are below the dashed line. The order of program execution in this figure is from left to right, and the order of subprogram calls is from top to bottom. The program line numbers to which these various functions refer are listed at the bottom of each box.

The main task of drawing a series of sprite figurines is under the direction of Function 3 (Draw & Save Sprite Codes). The task of initializing the work area and handling individual figurines and their models is directed by subprogram Expander, which constructs this model when given the pattern identifier for the figurine. After this, Drawer directs changes in the model display according to the user's keyboard inputs. Then it calls upon subprogram Addpix to make the corresponding changes in the pattern identifier for the figurine that is being drawn. When the figurine is complete, Drawer will call subprogram Screept to output the model on the thermal printer (if this option is chosen).

A Demonstration Program

After you generate cassette or disk data files of figurine pattern-identifiers with *Spriter*, you are ready to incorporate these into an animation sequence within a program. The short demonstration program (Listing 2) is a very simple example. This program is in effect a continuous loop projector that sequences through a series of sprite figurines to produce animation of the sprite that is moved across the screen. After the program reads the pattern identifiers from cassette tape or disk, it goes into the animation loop. You can stop the looping by pressing SHIFT C (on the 99/4) or FCTN 4 (on the 99/4A).

Keep in mind that this program is just a very simple demonstration of the sprite animation technique. You can use it to study the figurines files created by *Spriter*, and perhaps as a starting point for writing more elaborate sprite animation programs that are more apt to your specific applications [We've also included an additional program (Listing 3) that incorporates DATA statements for those wishing to get a feel for the animation process before working with *Spriter*.—Ed.]

Listing 1

```
100 REM *********************
110 REM *      SPRITER      *
120 REM *********************
130 REM
140 REM
150 REM
160 CALL CHARSET :: FOR I=96 TO 143 :: CALL CHAR(I,"0"):: NEXT I
170 INPUT "DO YOU HAVE A THERMAL PRINTER(Y/N)?":TPS
180 DIM CHA$(50),ID$(50)
190 INPUT "DO YOU WANT TO INPUT A FILE OF CHARACTERS FROM TAPE OR DISK (Y/N)?":ANS :: IF ANS<>"Y" THEN 240
200 DISPLAY AT(24,1):"FILE NAME" :: ACCEPT AT(24,11)SIZE(10)VALIDATE(UALPHA,DIGIT):NAM$ :: IF POS(NAM$,".",1)<>0 THEN 200
210 PRINT "ENTER '1' FOR TAPE '2' FOR DISK" :: INPUT "(1/2)?":ANS
220 IF ANS="1" THEN @FILE$="CS1" ELSE IF ANS="2" THEN @FILE$="DSK1."&NAM$ ELSE GOTO 210
230 GOSUB 1260 :: GOTO 250
240 IF ANS<>"N" THEN 190 ELSE NS=0 :: GOTO 290
250 IF TPS="N" THEN 280
260 OPEN #1:"TP.U.E.S",OUTPUT :: FOR J=0 TO NS
270 PRINT #1:J,ID$(J)):: NEXT J :: CLOSE #1
280 NS=NS+1 :: C$=CHA$(0)
290 FOR I=NS TO 1000
300 GOSUB 520
310 DISPLAY AT(2,1):ID$(NS):: DISPLAY AT(22,1):C$
320 DISPLAY AT(3,1):"PRESS ANY KEY TO CONTINUE."
330 CALL KEY(0,K,S):: IF S=0 THEN 330
340 CALL CLEAR :: INPUT "ENTER COLOR CODE FOR SPRITE.":COL
350 CALL CHAR(96,C$):: CALL SPRITE(#1,96,COL,30,30,0,-30):: CALL MAGNIFY(4)
360 DISPLAY AT(10,3):"PRESS ANY KEY TO CONTINUE."
370 CALL KEY(0,K,S):: IF S=0 THEN 370 ELSE CALL DELSPRITE(ALL)
380 INPUT "DO YOU WANT TO SAVE THE CHARACTER CODE OF THIS SPRITE(Y/N)?":ANS
390 IF ANS="Y" THEN CHAS(NS)=C$
400 INPUT "DO YOU WANT TO CONTINUE(Y/N)?":ANS :: IF ANS="N" THEN 430 ELSE IF ANS<>"Y" THEN 400
410 NS=NS+1
420 NEXT I :: END
430 INPUT "DO YOU WISH TO SAVE RESULTS ON TAPE OR DISK(Y/N)?":ANS
440 IF ANS="N" THEN 510 ELSE IF ANS<>"Y" THEN 430
450 DISPLAY AT(24,1):"ENTER FILE NAME" :: ACCEPT AT(24,16)SIZE(10)VALIDATE(UALPHA,DIGIT):NAM$ :: IF POS(NAM$,".",1)<>0 THEN 450
460 PRINT "ENTER '1' FOR TAPE '2' FOR DISK" :: INPUT "(1/2)?":ANS
470 IF ANS="1" THEN @FILE$="CS1" ELSE IF ANS="2" THEN @FILE$="DSK1."&NAM$ ELSE GOTO 460
480 OPEN #1:@FILE$,INTERNAL,OUTPUT,FIXED 128
490 PRINT #1:NAM$,NS
```

```
500 FOR K=0 TO NS :: PRINT #1:ID$(K),C
    HA$(K):: NEXT K :: CLOSE #1
510 END
520 REM SUB DRAWER(TP$,C$,NS,ANS,CHA$(
    ),ID$())
530 CALL CHAR(33,RPT$("F",16))
540 IF C$="" THEN 590
550 INPUT "DO YOU WANT TO INITIALIZE W
    ITH A PREVIOUSLY DEFINED CHARACTER
    (Y/N)?":ANS
560 IF ANS="N" THEN C$="" :: GOTO 590
    ELSE IF ANS<>"Y" THEN 550
570 INPUT "ENTER INDEX OF CHARACTER DE
    SIRED,ANY "-" VALUE FOR MOST RECEN
    TLY DEFINED":NOS
580 IF NOS<0 THEN 590 ELSE C$=CHA$(NOS
    ):: NXX=NOS :: GOTO 600
590 NXX=NS-1
600 M=16 :: IF LEN(C$)=0 THEN C$=RPT$(
    "0",64):: F=0 ELSE F=1
610 IF LEN(C$)=16 THEN C$=C$&RPT$("0",
    48)
620 N=1 :: C1$=SEG$(C$,1,16):: C2$=SEG
    $(C$,17,16):: C3$=SEG$(C$,33,16)::
     C4$=SEG$(C$,49,16)
630 PRINT "USE ARROW KEYS AND 'W,R,C,Z
    ' TO MOVE CURSOR,OR TO CHANGE POLA
    RITY USE 'F' FOR DARK AND 'A' FOR LIG
    HT."
640 CALL KEY(0,K,S):: IF S=0 THEN 640
650 CALL CLEAR :: CALL HCHAR(4,4,30,M+
    2):: CALL HCHAR(M+5,4,30,M+2)
660 CALL VCHAR(5,4,30,M):: CALL VCHAR(
    5,M+5,30,M):: X,Y=5
670 IF ANS="Y" THEN GOSUB 1130
680 IF NXX>=0 THEN DISPLAY AT(2,1):ID$
    (NXX):: DISPLAY AT(22,1):C$
690 CALL HCHAR(X,Y,30,1):: CT$=C$ :: G
    OSUB 990 :: C$=CT$
700 CALL KEY(1,K,S)
710 IF S=0 THEN 700 ELSE IF N=1 THEN C
    ALL HCHAR(X,Y,33,1)ELSE CALL HCHAR
    (X,Y,32,1)
720 IF K=1 THEN N=0
730 IF K=2 THEN N=1
740 IF K=5 AND X>5 THEN X=X-1
750 IF K=0 AND X<M+4 THEN X=X+1
760 IF K=2 AND Y>5 THEN Y=Y-1
770 IF K=3 AND Y<M+4 THEN Y=Y+1
780 IF K=4 AND X>5 THEN IF Y>5 THEN X=
    X-1 :: Y=Y-1
790 IF K=6 AND X>5 THEN IF Y<M+4 THEN
    X=X-1 :: Y=Y+1
800 IF K=15 AND X<M+4 THEN IF Y>5 THEN
     X=X+1 :: Y=Y-1
810 IF K=14 AND X<M+4 THEN IF Y<M+4 TH
    EN X=X+1 :: Y=Y+1
820 IF K=18 THEN 910
830 IF X>4 AND X<13 THEN IF Y>4 AND Y<
    13 THEN P=1 ELSE IF Y>4 THEN P=3 E
    LSE P=2 ELSE IF Y<13 THEN P=4 ELSE P=
    ND Y<13 THEN P=2 ELSE P=4
840 IF P=1 THEN X0=X-5 :: Y0=Y-5 :: CH
    $=SEG$(C$,1,16)
850 IF P=2 THEN X0=X-13 :: Y0=Y-5 :: C
    H$=SEG$(C$,17,16)
860 IF P=3 THEN X0=X-5 :: Y0=Y-13 :: C
    H$=SEG$(C$,33,16)
870 IF P=4 THEN X0=X-13 :: Y0=Y-13 ::
    CH$=SEG$(C$,49,16)
880 CT$=CH$ :: GOSUB 990 :: CH$=CT$
890 IF P=1 THEN C1$=CH$ ELSE IF P=2 TH
    EN C2$=CH$ ELSE IF P=3 THEN C3$=CH
    $ ELSE C4$=CH$
900 CALL HCHAR(X,Y,30,1):: C$=C1$&C2$&
    C3$&C4$ :: GOTO 700
910 DISPLAY AT(22,1):"ENTER SPRITE NAM
    E." :: DISPLAY AT(23,1):"" :: DISP
    LAY AT(24,1):""
920 ACCEPT AT(23,1):ID$(NS)
930 IF TP$="N" THEN GOTO 980
940 DISPLAY AT(22,1):"WANT TO COPY ON
    T.P.(Y/N)?" :: ACCEPT AT(23,1):ANS
950 IF ANS="N" THEN GOTO 980 ELSE IF A
    NS<>"Y" THEN 940
960 DISPLAY AT(2,1):ID$(NS):: DISPLAY
    AT(22,1):C$
970 CALL SCREEPT
980 RETURN
990 REM SUB ADDPIX(X,Y,N,C$)
1000 IF Y0<4 THEN ZT=2*X0+1 :: YT0=3-Y0
     ELSE ZT=2*X0+2 :: YT0=7-Y0
1010 A2$=SEG$(CT$,ZT,1)
1020 IF ZT>1 THEN A1$=SEG$(CT$,1,ZT-1)
1030 IF ZT<16 THEN A3$=SEG$(CT$,ZT+1,16
    -ZT)
1040 NH=ASC(A2$):: IF NH<=57 THEN NH=A
    -48 ELSE NH=NH-55
1050 ZZ=INT(NH/(2^YT0))-2*INT(NH/(2^(YT
    0+1)))
1060 IF ZZ=0 AND N=1 THEN NH=NH+2^YT0
1070 IF ZZ=1 AND N=0 THEN NH=NH-2^YT0
1080 IF NH<=9 THEN A2$=STR$(NH)ELSE A2$
    =CHR$(NH+55)
1090 IF ZT=16 THEN CT$=A1$&A2$
1100 IF ZT=1 THEN CT$=A2$&A3$
1110 IF ZT<>16 AND ZT<>1 THEN CT$=A1$&A
    2$&A3$
1120 RETURN
1130 REM SUB EXPANDER(C$,X0,Y0)
1140 DEF B(A)=INT(NHF/(2^A))-2*INT(NHF/
    (2^(A+1)))
1150 FOR IW=0 TO 15 :: FOR JW=0 TO 15
1160 IF IW>7 THEN IW0=IW-8 ELSE IW0=IW
1170 IF IW>7 THEN IW0=IW-8 ELSE IW0=IW
1180 IF IW<8 THEN IF JW<8 THEN LW=1 ELS
    E LW=3 ELSE IF JW<8 THEN LW=2 ELSE
     LW=4
1190 IF IW0<4 THEN ZW=2*IW0+1 :: YW=3-J
    W0 ELSE ZW=2*IW0+2 :: YW=7-JW0
1200 SA2$=SEG$(S$,ZW,1)
1210 SA2$=SEG$(C$,ZW+16*(LW-1),1)
1220 NHF=ASC(SA2$):: IF NHF<=57 THEN NH
    F=NHF-48 ELSE NHF=NHF-55
1230 IF B(YW)=1 THEN CALL HCHAR(X+IW,Y+
    JW,33,1)
1240 NEXT JW :: NEXT IW
1250 RETURN
1260 REM SUB CASTER(@FILE$,N,I$(),C$())
1270 OPEN #2:@FILE$,INTERNAL,INPUT ,FIX
     ED 128 :: GOTO 1280
1280 INPUT #2:NAM$,NS
1290 FOR I=0 TO NS
1300 INPUT #2:ID$(I),CHA$(I):: NEXT I
     : CLOSE #2
1310 N3=23 :: N1=0 :: IF NS<=24 THEN N2
    =NS ELSE N2=23
1320 FOR I=N1 TO N2 :: IF I>NS THEN 137
    0
1330 PRINT I;ID$(I):: NEXT I
1340 PRINT "PRESS ANY KEY TO CONTINUE."
1350 CALL KEY(0,K,S):: IF S=0 THEN 1350
1360 IF NS>N3 THEN N1=N1+24 :: N2=N2+24
     :: N3=N3+24 :: GOTO 1320
1370 RETURN
1380 SUB SCREEPT
1390 OPEN #255:"TP.U.E.S",OUTPUT :: FOR
     X=1 TO 24 :: S$=""
1400 FOR Y=1 TO 32 :: CALL GCHAR(X,Y,Z)
     :: S$=S$&CHR$(Z)
1410 NEXT Y :: PRINT #255:S$ :: NEXT X
     :: CLOSE #255
1420 SUBEND
```

Listing 2

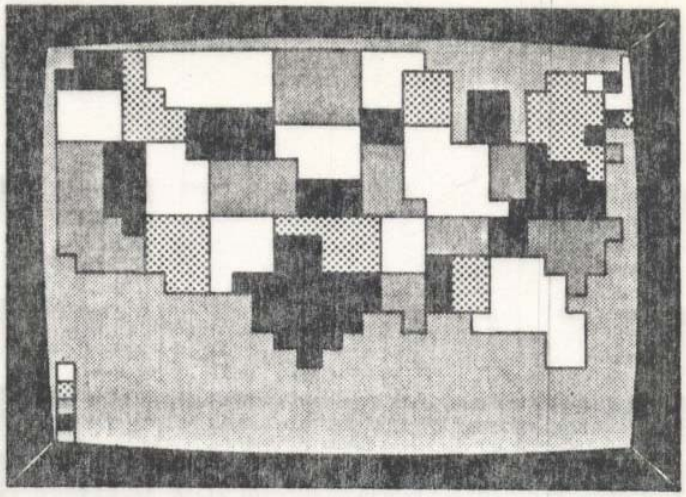
```
100 REM ********************
110 REM *    SPRITE DEMO     *
120 REM ********************
130 REM
140 REM
150 REM
160 REM
170 REM DEMONSTRATION OF SPRITE ANIMAT
    ION USING CASSETTE OR DISK DATA FI
    LE.
180 CALL CLEAR
190 DIM CHA$(17),ID$(17)
200 CALL CASTER(NS,ID$(),CHA$())
210 FOR I=0 TO NS :: CALL CHAR(136-4*I
    ,CHA$(I))
220 NEXT I
230 CALL CLEAR
240 CALL SPRITE(#1,136,2,30,30,0,-10):
    : CALL MAGNIFY(4)
250 FOR I=0 TO NS :: CALL PATTERN(#1,1
    36-4*I):: CALL DELAY :: NEXT I ::
    GOTO 250
260 END
270 SUB CASTER(N,I$(),C$())
280 REM
290 REM ********************
300 REM SET FILES$='CS1' FOR
310 REM TAPE FILES; USE YOUR
320 REM FILE NAME:
330 REM 'DSK1.FILENAME'
340 REM FOR DISK FILES.
350 REM ********************
360 REM
370 OPEN #2:FILES$,INTERNAL,INPUT ,FIXE
    D 128
380 INPUT #2:NAMS$,N
390 FOR I=0 TO N
400 INPUT #2:I$(I),C$(I):: NEXT I :: C
    LOSE #2
410 SUBEND
420 SUB DELAY
430 FOR I=0 TO 15 :: NEXT I
440 SUBEND
```

Listing 3

```
100 REM ********************
110 REM *   SPRITE DEMO 2    *
120 REM ********************
130 REM
140 REM
150 REM DEMONSTRATION OF SPRITE ANIMAT
    ION USING DATA STATEMENTS
160 CALL CLEAR
170 DIM I$(17),C$(17)
180 GOSUB 250 !CASTER
190 FOR I=0 TO N :: CALL CHAR(136-4*I,
    C$(I))
200 NEXT I
210 CALL CLEAR
220 CALL SPRITE(#1,136,2,30,30,0,-10):
    : CALL MAGNIFY(4)
230 FOR I=0 TO N :: CALL PATTERN(#1,13
    6-4*I):: GOSUB 300 :: NEXT I :: GO
    TO 230
240 END
250 REM SUBROUTINE CASTER
260 READ NAM$,N
270 FOR I=0 TO N
280 READ I$(I),C$(I):: NEXT I
290 RETURN
300 REM SUBROUTINE DELAY
310 FOR J=0 TO 15 :: NEXT J
320 RETURN
330 DATA MAN6N81,12
340 DATA MAN#1,000060909060F0F0F1E060F0
    F1908040800000000000000000020508
    00000000
350 DATA MAN#2,030404030707 2F1303030 70
    6060207000080800008090D0A0808080808
    0808000
360 DATA MAN#2.5,0609090060F0F172606060
    E1EA2420103000000000000080408000000
    0000000
370 DATA MAN#3,0007090306 0F0F172F06060
    60F0908180000000000000000000000000
    0804080
380 DATA MAN#4,00001824 1C0C1C2C4E16060
    7060202060000000000000000000040A0000
    00000000
390 DATA MAN#5,0000000000000000387FDE966
    2428100010000000000000205080000000
    00080000
400 DATA MAN#6,0006142414 0C0C0C0C1C1E1
    E1E1D0C1000000000000000000000000
    00080800
410 DATA MAN#6.5,0000002020184C7C0C0C0
    E0606090E040000000000000000000000
    00008080
420 DATA MAN#7,0000000000000000408040 2F1
    E376640C000000000000000000040800000
    0804020
430 DATA MAN#8,0000001 10A0603010101030
    3062A12060000008448 59A0C0C0808000
    00000000
440 DATA MAN#1,0006090906 0F0F0F1E060F0
    F1908040800000000000000000020508
    00000000
450 DATA MAN#3,0007090306 0F0F172F06060
    60F0908180000000000000000000000000
    0804080
460 DATA MAN#2.5,0609090060F0F172606060
    E1EA2420103000000000000080408000000
    0000000
```

COLOR MAPPING and the TI-99/4A

TI BASIC



One of the principal features of the new technology exhibited by low-cost home computers is their graphic capabilities. But these small computers' graphic capabilities in the area of mapping is often overlooked. Statistical mapping is not new; cartographers have used the methods described in this article for decades, and sophisticated mapping programs that run on large mainframe computers have been available (from Harvard University and elsewhere) for a number of years. Their application for the small computer field, however, especially in the classroom setting, should be further explored and documented.

The program described in this article, *United States Choropleth Map,* was written for the TI-99/4A. No peripherals are needed, except for a cassette recorder to store the program. Therefore, anyone with the console can get started immediately and experience the excitement of computer mapping. The program should benefit a large number of users: For example, classroom teachers, from the upper elementary grades through college levels in geography, can utilize it; sales and marketing managers, and others interested in the spatial distribution of goods and services may also find it especially useful; political scientists can easily see the national election results displayed almost instantaneously.

Choropleth Mapping

Simply defined, choropleth mapping has been likened to a spatial table. Enumeration units—which can be census tracts, counties, states, or other small area geography—are symbolized by different area patterns, depending on the values they represent. Typically, the original data are divided into a number of data classes (map classes). The individual enumeration units will be symbolized according to the map class into which their data value falls. Enumeration units are put into classes because it is usually impractical, or not feasible, to apply an area pattern for *each* data value.

Classing, of course, is similar to a sieve; individual values "fall" into each group depending on the class limits. This results in a generalization, and the final map is a *simplification* of the original data. Nonetheless, choropleth mapping has a number of advantages over a simple table of values. It provides a third, or spatial dimension to a rather dull list of values in tabular format. In the bibliography, I've listed several good books that discuss the methods and rationale of this form of mapping.

Symbolization on choropleth maps takes on several different forms. In the case of *black and white* mapping, the enumeration units are symbolized by area *patterns* to differentiate each class from all others. Different shades of grey, ranging from near-black to near-white, are often used. *Color* symbolization includes two forms: (1) different hues (such as green, red, blue, etc.) for the various classes, or (2) different values (shades) of the *same* color. The present program uses the second method.

Main Features of the Program

Figure 1 illustrates the main components of the program's logic, and Figure 2 lists the most important variables. I wrote the program with flexibility in mind: New subroutines can be incorporated as different versions are developed. Lines 170 to 260 of the program are used for an opening screen, which displays the program name.

The first section, Program Instructions, provides the option list and incorporates directions for data input. The present version accommodates only data from the keyboard. (You may wish to add program statements to read data from a file system). The data is input by entering the values to be mapped for each state, by the alphabetical order of the states.

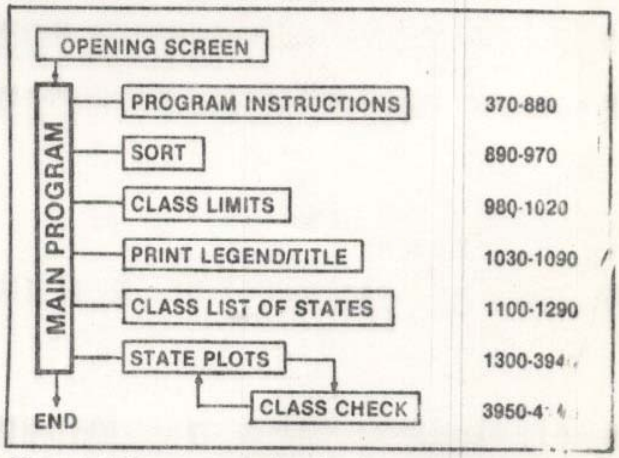


Figure 1 -Main program logic, showing subroutines, of United States Choropleth Map.

SC	-Map background color
MC	-Map color (blue or green)
C(1-5)	-Map class colors
V(1-50)	-Values of each state to be mapped
TT$	-Map title
X1	-Limit for class 1
X2	-Limit for class 2
X3	-Limit for class 3
X4	-Limit for class 4
K(1-5)	-Character sets
S(1-5)	-ASCII character identifiers
SN$(1-50)	-States' names
NN	-State's number

Figure 2--Principal variables used in mapping program.

After the data are entered, the main program directs the flow to a simple bubble sort subroutine, where the data values are sorted into ascending order. The data values are then classed, and the class limits are selected in the Class Limits section. There are a number of ways in which data may be classed. This program will class the data values into *quintiles*—that is, into *five* classes each having the same *number* of values. As the data set has been arrayed in ascending order, the values of the class limits are computed rather easily.

Program flow is next directed to printing. With the TI-99/4A and the BASIC language supplied with the standard computer, printed ASCII characters must be displayed *before* the color graphic blocks are called on the screen. Otherwise, scrolling will move the color graphics off the screen. The Print Legend and Title subroutine displays the classed values and user-chosen title on the lower portion of the screen.

State Plots is next. Each state is assigned an ordinal number based on its alphabetical rank (1-50). As each state is encountered, flow is directed to a subroutine, Class Check, in which the state's ordinal number is used to determine which color the state should be.

Outlines of the states are not variable, but the color (symbolization) varies, of course, depending on the class in which each state falls. Flow continues until each state has been displayed on the screen. A color graphic block is displayed adjacent to the printed legend values at the bottom of the screen. The program ends with a GO TO statement (line 3940); the screen will display the map until the user presses SHIFT C or FCTN = to BREAK program.

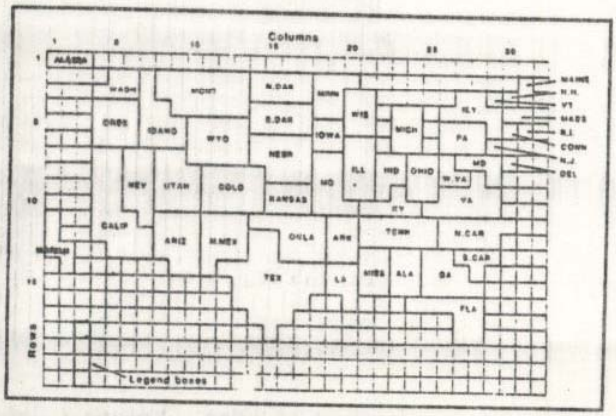


Figure 3 - The graphic blocks used to identify the shapes of the states in the choropleth map program. Each block's color is generated with the CALL COLOR command.

Mapping on the TI-99/4A

The color graphic capabilities of the TI-99/4A include a screen which is divided into 32 columns and 24 rows, each block of which is addressable by a row and column identifier in the CALL HCHAR and VCHAR commands. Any of 16 colors (including transparent) can be specified. Further resolution is possible by using the CALL CHAR command, with which the user can specify the "on" and "off" condition of 64 dots in each graphic block, through the use of hexadecimal codes. The present program utilizes only the 32 × 24 resolution screen, and does not develop the refinements of the shapes of the states that are possible with the CALL CHAR command. The blocks used to identify the states are illustrated in Figure 3. Although only an approximation is achieved with this resolution, the shapes resemble fairly well the individual states, and relative area is proportional to real geographical areas. Other users may wish to modify these (although I suspect that the 16K RAM will be taxed if they do).

The Choice of Color Symbolization

As mentioned previously one standard, acceptable way to symbolize the areas on choropleth maps is to vary the lightness or darkness of one color, in accordance with the values represented. Classes having higher values are rendered darker, and the lower-valued classes lighter. For this program, the highest class is black, the lowest class white, and the three intermediate classes are in three shades of green or three shades of blue. The TI-99/4A can display 15 different colors, and fortunately there are three different greens and blues, each ranging from light to dark. Symbolizing the color classes in this manner better shows the *total form* of the distribution over the map. The map reader gets a better idea of the continuously changing nature of the *spatial* attributes of the data.

Program Enhancements

You can make any number of useful changes to this program. You may wish to provide alternate ways of classing the data (e.g., quartiles, equal steps, standard deviations, or others), to add new subroutines, or to enter your own classes. A *different* color for each class could be used in the color symbolization. The variable C(1-5) need only be changed to conform to the other color code options used by the TI BASIC. With small changes, data sets could be input from external files rather than from the keyboard. This would be especially useful in classroom settings, where census or other data from previous years (and other geographical data) can be compared with present patterns.

Computer-aided instruction (CAI) which uses inquiry questions generated by the spatial distribution seen on the screen could also be added to this program. Geographical concepts could be brought out in this manner, and students could easily test hypotheses.

One most intriguing enhancement would be to introduce animation (dynamic cartography) to the program. Various data sets could be read (from files) and displayed in fairly fast sequence to produce a dynamic, *changing* image of the geographical distribution. For example, different population densities from 1850 to 1980 would show the steady drift of our population from east

to west; a temporal set of sales (or income) performance data would be of interest to marketing analysts. One advantage of all computer mapping is its ability to show the dynamic qualities of geographical data. This capability is possible on the versatile TI-99/4A.

Bibliography

Robinson, Arthur; Sale, Randall; and Morrison, Joel. *Elements of Cartography*. 4th ed. New York: John Wiley and Sons, 1978.

Lawrence, G.R.P. *Cartographic Methods*. 2nd ed. New York: Methuen and Co., Ltd., 1979.

Harvard Graduate School of Design. Various publications, Laboratory for Computer Graphics and Spatial Analysis. Cambridge, Mass.


```
100 REM  *********************
110 REM  *  CHOROPLETH MAP  *
120 REM  *********************
130 REM
140 REM
150 REM
160 DIM SN$(50),V(50),VV(50)
170 CALL CLEAR
180 CALL SCREEN(12)
190 PRINT TAB(12);"UNITED"
200 PRINT TAB(12);"STATES"
210 PRINT ::::::TAB(8);"CHOROPLETH MAP":
    :::::::::
220 CALL COLOR(9,5,5)
230 CALL VCHAR(3,4,96,17)
240 CALL HCHAR(3,5,96,25)
250 CALL HCHAR(19,5,96,25)
260 CALL VCHAR(4,29,96,15)
270 RESTORE 280
280 DATA ALABAMA,ALASKA,ARIZONA,ARKANS
    AS,CALIFORNIA,COLORADO,CONNECTICUT
    ,DELAWARE,FLORIDA,GEORGIA,HAWAII,I
    DAHO
290 DATA ILLINOIS,INDIANA,IOWA,KANSAS,
    KENTUCKY,LOUISIANA,MAINE,MARYLAND,
    MASSACHUSETTS,MICHIGAN,MINNESOTA
300 DATA MISSISSIPPI,MISSOURI,MONTANA,
    NEBRASKA,NEVADA,NEW HAMPSHIRE,NEW
    JERSEY,NEW MEXICO,NEW YORK
310 DATA NORTH CAROLINA,NORTH DAKOTA,O
    HIO,OKLAHOMA,OREGON,PENNSYLVANIA,R
    HODE ISLAND,SOUTH CAROLINA,SOUTH D
    AKOTA
320 DATA TENNESSEE,TEXAS,UTAH,VERMONT,
    VIRGINIA,WASHINGTON,WEST VIRGINIA,
    WISCONSIN,WYOMING
330 FOR I=1 TO 50
340 READ SN$(I)
350 NEXT I
360 CALL CLEAR
370 PRINT TAB(6);"PROGRAM INSTRUCTIONS
    "
380 PRINT TAB(14);"***"
390 PRINT "CHOOSE MAP BACKGROUND COLOR
    ":::
400 PRINT TAB(8);"1-MEDIUM RED"
410 PRINT TAB(8);"2-LIGHT RED"
420 PRINT TAB(8);"3-DARK YELLOW"
430 PRINT TAB(8);"4-LIGHT YELLOW"
440 PRINT TAB(8);"5-GREY"::::
450 CALL KEY(0,KEY,ST)
460 IF (KEY<49)+(KEY>53)=-1 THEN 450
470 IF KEY<>53 THEN 500
480 SC=15
490 GOTO 510
500 SC=KEY-48
510 CALL CLEAR
520 PRINT "CHOOSE MAP COLOR"
530 PRINT :::"1-BLUE"::"2-GREEN":::::
540 CALL KEY(0,KEY,ST)
550 IF KEY=49 THEN 610
560 IF KEY<>50 THEN 540
570 C(2)=4
580 C(3)=3
590 C(4)=13
600 GOTO 640
```

```
610 C(2)=8
620 C(3)=6
630 C(4)=5
640 C(1)=16
650 C(5)=2
660 CALL CLEAR
670 PRINT "DATA ENTRY INSTRUCTIONS"
680 PRINT :"PLEASE ENTER THE VALUES"
690 PRINT "FOR EACH STATE."
700 PRINT :"YOU MAY ENTER A VALUE":"UP
    TO 8 DIGITS."
710 PRINT "DO NOT USE COMMAS.":::
720 FOR I=1 TO 50
730 INPUT SN$(I)&" = ":V(I)
740 VV(I)=V(I)
750 PRINT
760 NEXT I
770 CALL CLEAR
780 PRINT TAB(6);"WHAT IS THE TITLE"
790 PRINT TAB(6);"OF YOUR MAP?"
800 PRINT
810 PRINT TAB(6);"BECAUSE OF SPACE"
820 PRINT TAB(6);"LIMITATIONS, KEEP"
830 PRINT TAB(6);"YOUR TITLE TO LESS"
840 PRINT TAB(6);"THAN 14 SPACES.":::
850 INPUT "TITLE ":TT$
860 CALL CLEAR
870 PRINT TAB(6):"ONE MOMENT PLEASE"
880 PRINT :TAB(9);"* SORTING *":::
890 REM  SORT SUBROUTINE
900 FOR N=1 TO 49
910 FOR LT=N+1 TO 50
920 IF VV(N)<=VV(LT)THEN 960
930 LET W=VV(N)
940 LET VV(N)=VV(LT)
950 LET VV(LT)=W
960 NEXT LT
970 NEXT N
980 REM  CLASS LIMITS SUBROUTINE
990 X1=VV(10)+(VV(11)-VV(10))/2
1000 X2=VV(20)+(VV(21)-VV(20))/2
1010 X3=VV(30)+(VV(31)-VV(30))/2
1020 X4=VV(40)+(VV(41)-VV(40))/2
1030 REM  PRINT LEGEND, TITLE
1040 CALL CLEAR
1050 PRINT TAB(2);VV(1);"-";X1
1060 PRINT TAB(2);X1;"-";X2
1070 PRINT TAB(2);X2;"-";X3;TT$
1080 PRINT TAB(2);X3;"-";X4
1090 PRINT TAB(2);X4;"-";VV(50)
1100 REM  ALL STATE PLOTS
1110 K(1)=9
1120 K(2)=10
1130 K(3)=11
1140 K(4)=12
1150 K(5)=13
1160 S(1)=96
1170 S(2)=104
1180 S(3)=112
1190 S(4)=120
1200 S(5)=128
1210 FOR T=1 TO 5
1220 CALL COLOR(K(T),C(T),C(T))
1230 NEXT T
1240 CALL SCREEN(SC)
1250 CALL HCHAR(19,4,96)
```

```
1260 CALL HCHAR(20,4,104)
1270 CALL HCHAR(21,4,112)
1280 CALL HCHAR(22,4,120)
1290 CALL HCHAR(23,4,128)
1300 REM  ALABAMA
1310 NN=1
1320 GOSUB 3960
1330 CALL VCHAR(13,23,S(T),3)
1340 CALL VCHAR(13,24,S(T),3)
1350 REM  ALASKA
1360 NN=2
1370 GOSUB 3960
1380 CALL HCHAR(1,1,S(T),3)
1390 REM  ARIZ
1400 NN=3
1410 GOSUB 3960
1420 CALL VCHAR(11,8,S(T),3)
1430 CALL VCHAR(11,9,S(T),4)
1440 CALL VCHAR(11,10,S(T),4)
1450 REM  ARK
1460 NN=4
1470 GOSUB 3960
1480 CALL VCHAR(11,19,S(T),3)
1490 CALL VCHAR(11,20,S(T),3)
1500 REM  CALIF
1510 NN=5
1520 GOSUB 3960
1530 CALL VCHAR(7,4,S(T),6)
1540 CALL VCHAR(7,5,S(T),7)
1550 CALL VCHAR(11,6,S(T),5)
1560 CALL VCHAR(12,7,S(T),2)
1570 REM  COLO
1580 NN=6
1590 GOSUB 3960
1600 CALL HCHAR(8,11,S(T),4)
1610 CALL HCHAR(9,11,S(T),4)
1620 CALL HCHAR(10,11,S(T),4)
1630 REM  CONN
1640 NN=7
1650 GOSUB 3960
1660 CALL HCHAR(5,30,S(T))
1670 REM  DEL
1680 NN=8
1690 GOSUB 3960
1700 CALL HCHAR(7,30,S(T))
1710 REM  FLA
1720 NN=9
1730 GOSUB 3960
1740 CALL HCHAR(16,24,S(T),5)
1750 CALL HCHAR(17,27,S(T),2)
1760 CALL HCHAR(18,27,S(T),2)
1770 REM  GA
1780 NN=10
1790 GOSUB 3960
1800 CALL HCHAR(13,25,S(T),2)
1810 CALL HCHAR(14,25,S(T),5)
1820 CALL HCHAR(15,25,S(T),3)
1830 REM  HAWAII
1840 NN=11
1850 GOSUB 3960
1860 CALL VCHAR(11,1,ST,3)
1870 CALL VCHAR(13,2,S(T),3)
1880 REM  IDAHO
1890 NN=12
1900 GOSUB 3960
1910 CALL VCHAR(2,7,S(T),5)
1920 CALL VCHAR(4,8,S(T),3)
1930 CALL VCHAR(5,9,S(T),2)
1940 REM  ILL
1950 NN=13
1960 GOSUB 3960
1970 CALL VCHAR(6,20,S(T),4)
1980 CALL VCHAR(6,21,S(T),4)
1990 REM  IND
2000 NN=14
2010 GOSUB 3960
2020 CALL VCHAR(7,22,S(T),3)
2030 CALL VCHAR(7,23,S(T),2)
2040 REM  IOWA
2050 NN=15
2060 GOSUB 3960
2070 CALL HCHAR(5,18,S(T),2)
2080 CALL HCHAR(6,18,S(T),2)
2090 REM  KAN
2100 NN=16
2110 GOSUB 3960
2120 CALL HCHAR(9,15,S(T),3)
2130 CALL HCHAR(10,15,S(T),3)
2140 REM  KY
2150 NN=17
2160 GOSUB 3960
2170 CALL HCHAR(9,23,S(T))
2180 CALL HCHAR(10,21,S(T),4)
2190 REM  LA
2200 NN=18
2210 GOSUB 3960
2220 CALL VCHAR(14,19,S(T),2)
2230 CALL VCHAR(14,20,S(T),3)
2240 REM  MAINE
2250 NN=19
2255 GOSUB 3960
2260 CALL VCHAR(2,31,S(T),2)
2270 REM  MD
2280 NN=20
2290 GOSUB 3960
2300 CALL HCHAR(7,27,S(T),3)
2310 CALL HCHAR(8,29,S(T))
2320 REM  MASS
2330 NN=21
2340 GOSUB 3960
2350 CALL HCHAR(4,30,S(T),2)
2360 REM  MICH
2370 NN=22
2390 GOSUB 3960
2400 CALL VCHAR(4,23,S(T),3)
2410 CALL VCHAR(4,24,S(T),3)
2420 REM  MINN
2430 NN=23
2440 GOSUB 3960
2450 CALL HCHAR(2,18,S(T),3)
2460 CALL HCHAR(3,18,S(T),2)
2470 CALL HCHAR(4,18,S(T),2)
2480 REM  MISS
2490 NN=24
2500 GOSUB 3960
2510 CALL VCHAR(13,21,S(T),3)
2520 CALL VCHAR(13,22,S(T),3)
2530 REM  MO
2540 NN=25
2550 GOSUB 3960
2560 CALL VCHAR(7,18,S(T),4)
2570 CALL VCHAR(7,19,S(T),4)
2580 CALL VCHAR(10,20,S(T))
2590 REM  MONT
2600 NN=26
2610 GOSUB 3960
2620 CALL HCHAR(2,8,S(T),6)
2630 CALL HCHAR(3,8,S(T),6)
2640 CALL HCHAR(4,9,S(T),6)
2650 REM  NEBR
2660 NN=27
2670 GOSUB 3960
2680 CALL HCHAR(6,14,S(T),4)
2690 CALL HCHAR(7,14,S(T),4)
2700 CALL HCHAR(8,15,S(T),3)
2710 REM  NEV
2720 NN=28
2730 GOSUB 3960
2740 CALL VCHAR(7,6,S(T),4)
2750 CALL VCHAR(7,7,S(T),5)
2760 REM  NH
2770 NN=29
2780 GOSUB 3960
2790 CALL HCHAR(3,29,S(T))
2800 REM  NJ
2810 NN=30
```

```
2820 GOSUB 3960
2830 CALL HCHAR(6,29,S(T))
2840 REM  N MEX
2850 NN=31
2860 GOSUB 3960
2870 CALL VCHAR(11,11,S(T),4)
2880 CALL VCHAR(11,12,S(T),3)
2890 CALL VCHAR(11,13,S(T),3)
2900 REM  N YORK
2910 NN=32
2920 GOSUB 3960
2930 CALL HCHAR(3,27,S(T),2)
2940 CALL HCHAR(4,26,S(T),4)
2950 CALL HCHAR(5,29,S(T))
2960 REM  NC
2970 NN=33
2980 GOSUB 3960
2990 CALL HCHAR(11,26,S(T),5)
3000 CALL HCHAR(12,26,S(T),4)
3010 REM  N DAK
3020 NN=34
3030 GOSUB 3960
3040 CALL HCHAR(2,14,S(T),4)
3050 CALL HCHAR(3,14,S(T),4)
3060 REM  OHIO
3070 NN=35
3080 GOSUB 3960
3090 CALL VCHAR(7,24,S(T),3)
3100 CALL VCHAR(7,25,S(T),3)
3110 REM  OKLA
3120 NN=36
3130 GOSUB 3960
3140 CALL HCHAR(11,14,S(T),5)
3150 CALL HCHAR(12,16,S(T),3)
3160 CALL HCHAR(13,16,S(T),3)
3170 REM  ORE
3180 NN=37
3190 GOSUB 3960
3200 CALL HCHAR(4,4,S(T),3)
3210 CALL HCHAR(5,4,S(T),3)
3220 CALL HCHAR(6,4,S(T),3)
3230 REM  PA
3240 NN=38
3250 GOSUB 3960
3260 CALL HCHAR(5,28,S(T),3)
3270 CALL HCHAR(6,26,S(T),3)
3280 REM  RI
3290 NN=39
3300 GOSUB 3960
3310 CALL HCHAR(5,31,S(T))
3320 REM  S CAR
3330 NN=40
3340 GOSUB 3960
3350 CALL HCHAR(13,27,S(T),3)
3360 CALL HCHAR(14,28,S(T))
3370 REM  S DAK
3380 NN=41
3390 GOSUB 3960
3400 CALL HCHAR(4,14,S(T),4)
3410 CALL HCHAR(5,14,S(T),4)
3420 REM  TENN
3430 NN=42
3440 GOSUB 3960
3450 CALL HCHAR(11,21,S(T),5)
3460 CALL HCHAR(12,21,S(T),5)
```

```
3470 REM  TEX
3480 NN=43
3490 GOSUB 3960
3500 CALL HCHAR(12,14,S(T),2)
3510 CALL HCHAR(13,14,S(T),2)
3520 CALL HCHAR(14,12,S(T),7)
3530 CALL HCHAR(15,13,S(T),6)
3540 CALL HCHAR(16,13,S(T),5)
3550 CALL HCHAR(17,14,S(T),3)
3560 CALL HCHAR(18,15,S(T))
3570 REM  UTAH
3580 NN=44
3590 GOSUB 3960
3600 CALL VCHAR(7,8,S(T),4)
3610 CALL VCHAR(7,9,S(T),4)
3620 CALL VCHAR(8,10,S(T),3)
3630 REM  VERMONT
3640 NN=45
3650 GOSUB 3960
3660 CALL HCHAR(3,30,S(T))
3670 REM  VA
3680 NN=46
3690 GOSUB 3960
3700 CALL HCHAR(8,28,S(T))
3710 CALL HCHAR(9,26,S(T),4)
3720 CALL HCHAR(10,25,S(T),5)
3730 REM  WASH
3740 NN=47
3750 GOSUB 3960
3760 CALL HCHAR(2,5,S(T),2)
3770 CALL HCHAR(3,4,S(T),3)
3780 REM  W VA
3790 NN=48
3800 GOSUB 3960
3810 CALL HCHAR(7,26,S(T))
3820 CALL HCHAR(8,26,S(T),2)
3830 REM  WISC
3840 NN=49
3850 GOSUB 3960
3860 CALL VCHAR(3,20,S(T),3)
3870 CALL VCHAR(3,21,S(T),3)
3880 REM  WYO
3890 NN=50
3900 GOSUB 3960
3910 CALL HCHAR(5,10,S(T),4)
3920 CALL HCHAR(6,10,S(T),4)
3930 CALL HCHAR(7,10,S(T),4)
3940 GOTO 3940
3950 REM  CLASS CHECK
3960 IF V(NN)<=X1 THEN 4020
3970 IF V(NN)<=X2 THEN 4040
3980 IF V(NN)<=X3 THEN 4060
3990 IF V(NN)<=X4 THEN 4080
4000 T=5
4010 RETURN
4020 T=1
4030 RETURN
4040 T=2
4050 RETURN
4060 T=3
4070 RETURN
4080 T=4
4090 RETURN
4100 END
```

OVERLAND FLOW

TI BASIC

When rain falls to earth, part of it passes into the soil (unless the surface is impervious, such as concrete or asphalt) and the remainder disappears over a period of time either by evaporation or by runoff *(overland flow)* or by both. In most engineering drainage systems, the amount of water lost by evaporation is negligible; thus, drainage must be provided for all rainfall that does not infiltrate the soil or is not stored temporarily in surface depressions (lakes, swamps, etc.) within the drainage area. Until recently, the Rational Method was used for calculating design discharges for storm drains. This method has various drawbacks and is of limited applicability. For that reason, the method used in this program is the Izzard dimensionless hydrograph. This method has been verified in laboratory tests and gives computed overland flow hydrographs agreeing closely with the measured hydrographs. The result of Izzard's method can be used by engineers in the design of drainage facilities for parking lots, airports and highways, etc. (See sample problem.)

Program Description

Input to the *Overland Flow* Program consists of two elements. The first consists of rainfall data and the second consists of physical characteristics.

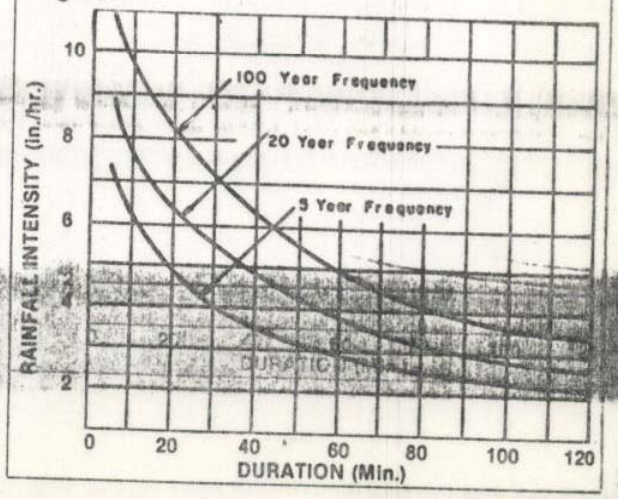
Standard curves (see Fig.1) may be developed to express rainfall intensity-duration relationships with an accuracy sufficient for drainage problems. Rainfall intensity-duration data have been published by the National Weather Service.

The following physical characteristics are needed: length, width and slope of the area of interest, and a coefficient [illegible] surface. The computer program contains a [illegible] surface coefficient can be determined. [illegible] program can be displayed on the [illegible] thermal printer. The program displays [illegible] and the overland flow hydrograph in tabular and/or graphic format. The program can calculate and display two hydrographs at any one time. Thus, it is possible to vary the input data and compare the results.

Definition of Terms.

Depression Storage:	Rainwater retained in puddles, ditches and other depressions in soil surface.
Equilibrium:	Occurs when the intensity of effective rainfall is equal to the outflow discharge. See Figure 2.
Equilibrium Time:	Time in minutes to reach the equilibrium condition. See Figure 2.
Infiltration:	Passage of water through the soil surface into the soil.
Intensity:	Effective rainfall intensity in inches per hour. Effective rainfall is that which occurs after depression storage and infiltration capacities are met. See Figure 2.

Figure 1



Maximum Discharge: The discharge, in cubic feet per second, when equilibrium is reached. See Figure 2.

Roughness Factor: A coefficient that characterizes the resistance to flow of a particular surface type.

Length: Distance, in feet, in the direction of slope, on which overland flow occurs. See Figure 3.

Slope: See Figure 3.

Width: Distance, in feet, perpendicular to the length, on which overland flow occurs. See Figure 3.

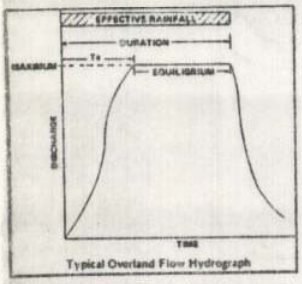


Figure 2

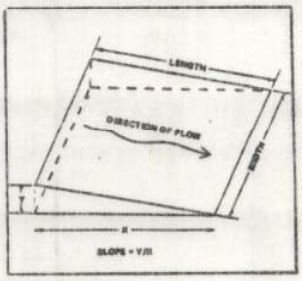


Figure 3

Operating Instructions

Step 1: Insert the cassette into a recorder, type: OLD CS1 and press ENTER. The computer then displays directions for loading the tape.

Step 2: When the cursor appears, type RUN, and press ENTER. When the title screen appears, press any key. Then select the screen or thermal printer as the device for output from the program.

Step 3: After choosing the output device, the computer asks for the input data needed to compute the overland flow hydrograph. Type in the data requested and press ENTER.

Step 4: After all data is entered, the computer will generate a hydrograph and display the menu. Select one of the following options:

1. DISPLAY DATA (GIVEN AND CALCULATED).
2. DISPLAY HYDROGRAPH.
3. COMPUTE ANOTHER HYDROGRAPH AND COMPARE.
4. REDIRECT OUTPUT.
5. ENTER NEW PROBLEM.
6. END PROGRAM.

After completing any of options 1 through 5 the computer returns to the menu.

OPTION 1: DISPLAY DATA (GIVEN AND CALCULATED)—If you select option 1, the computer will display the input data that you entered and the calculated values for equilibrium time and maximum discharge.

OPTION 2: DISPLAY HYDROGRAPH—If you select option 2, the computer asks you if you want the hydrograph displayed in tabular or graphic form or both. The graphic form plots the hydrograph points as percent of maximum discharge versus time. When two hydrographs are plotted, the maximum discharge is the greater of the two hydrograph maximums.

OPTION 3: COMPUTE ANOTHER HYDROGRAPH AND COMPARE—If you select option 3, the computer asks you to enter another set of data in order to calculate another hydrograph. Since the first hydrograph is retained by the computer, this option can be used to vary any of the input data and examine the result (see sample problems). The option can be used as many times as the user wishes. The computer always compares to the original hydrograph computed when the program was initially run. If a subsequent hydrograph is preferred to the original, select option 5 and enter the new hydrograph as the original. Thus, all other hydrographs computed via option 3 will be compared to the new hydrograph.

OPTION 4: REDIRECT OUTPUT—If you select option 4, you change the device to which the output is displayed.

OPTION 5: ENTER NEW PROBLEM—If you select option 5, the program begins again. This option is used to rerun the program without having to type RUN. Also, use this option in conjunction with option 3 to compare several hydrographs and select one that is best suited to the problem.

OPTION 6: END PROGRAM—This option returns the computer to TI BASIC.

EXPLANATION OF THE PROGRAM
Overland Flow

Line Nos.	
160-530	Program initialization: Character assignments and array dimensioning.
540-1080	Data entry.
1090-1490	Calculation of Overland Flow Hydrograph.
1500-1800	Display hydrograph, in tabular form, on video monitor or TI thermal printer.
1810-1990	Display menu and go to portion of program according to option selected.
2000-3200	Display hydrograph, in graphic form, on video monitor or TI thermal printer.
3210-3460	Subroutine to align numbers on display.
3470-3590	Display given and calculated data.
3600-3710	Prepare program to accept and calculate a second hydrograph.
3720-3790	Subroutine to blank and restore screen when displaying information on video monitor.
3930-4220	Scale and label axes of graph.
4230-4250	Common subroutine to check keyboard entry.
4260-4510	Select drive to program output.

Sample Problem 1

A parking lot 300 ft. long in the direction of the slope and 900 ft. wide has a tar and gravel pavement on a slope of .0025. Assuming a uniform rainfall intensity of 2.75 in/hr for 30 minutes, what is the maximum discharge that a gutter should be designed for?

Type RUN, then press ENTER.

Sample Problem 1-cont.

Select the thermal printer as the output device.

Enter the following data:

Intensity	2.75
Duration	30
Length	300
Width	900
Slope	.0025
Roughness Factor	.017

Select option 1.

The gutter should be designed for a maximum discharge of 17.2 cfs.

Sample Problem 2

If the parking lot described in problem 1 is resurfaced with asphalt, what effect will this have?

After problem 1 is complete select option 3.

Enter the same data as for problem 1 with the exception of the roughness factor. Enter .007.

Select options 1 and 2.

```
100 REM  °°°°°°°°°°°°°°°°°°°
110 REM   ° OVERLAND FLOW °
120 REM  °°°°°°°°°°°°°°°°°°°
130 REM
140 REM
150 REM
160 CALL CLEAR
170 CALL SCREEN(15)°
180 GOSUB 3720
190 PRINT TAB(8);"OVERLAND FLOW"::::::::
    ::::::::::
200 PRINT TAB(4);"PRESS ANY KEY TO BEG
    IN";
210 CALL SOUND(150,600,1)
220 GOSUB 3760
230 GOSUB 4230
240 CALL CLEAR
250 CALL SCREEN(8)
260 PRINT TAB(9);"INITIALIZING"::::::::
    ::::::
270 OPTION BASE 1
280 DIM H(2,2,22),IN(2),DU(2),LE(2),WI
    (2),SL(2),CR(2),TE(2),QW(2),QE(2),
    TPP(2,2)
290 DEF RD(X)=INT(100*X+.5)/100
300 DEF RD2(X)=INT(1E3*X+.5)/1E3
310 DEF RD4(X)=INT(1E4*X+.5)/1E4
320 DEF RD6(X)=INT(1E6*X+.5)/1E6
330 RESTORE 580
340 FOR I=1 TO 42
350 READ CODE,CH$
360 CALL CHAR(CODE,CH$)
370 NEXT I
380 DATA 99,E0808080818181FF,100,E0808
    0808080808080,1
    02,0000000006161FF
390 DATA 103,0000006161616161FF,104,6060
    6060,105,0606060606,106,0000000006060
    6060,107,0000241818242418
400 DATA 107,0000000000606060606,111,6060
    6060,112,0000606060,113,0000000006060
    6060,86,0018242418
410 DATA 119,0000000000000000000,121,F0F0
    F0F0F0F0F0F0F0,122,0000000000000000F0
    F0F0F0F0,123,0000000000000000F0F0F0F0
    F0F0F0F0,124,F0F0F0F0F0F0F0F0F0F0F0F0
    F0F0F0F0,125,0000000000000000F0F0F0F0
    F0F0F0F0,126,F0F0F0F0F0F0F0F0F0F0F0F0
    F0F0F0F0,127,0000000000000000F0F0F0F0
    F0F0F0F0,128,F0F0F0F0F0F0F0F0F0F0F0F0
    F0F0F0F0,129,0000000000000000F0F0F0F0
    F0F0F0F0,130,F0F0F0F0F0F0F0F0F0F0F0F0
    F0F0F0F0,131,0000000000000000F0F0F0F0
    F0F0F0F0,132,F0F0F0F0F0F0F0F0F0F0F0F0
    F0F0F0F0,133,0000000000000000F0F0F0F0
    F0F0F0F0,134,F0F0F0F0F0F0F0F0F0F0F0F0
    F0F0F0F0,135,0000000000000000F0F0F0F0
    F0F0F0F0,136,F0F0F0F0F0F0F0F0F0F0F0F0
    F0F0F0F0,137,0000000000000000F0F0F0F0
    F0F0F0F0,138,F0F0F0F0F0F0F0F0F0F0F0F0
    F0F0F0F0,139,0000000000000000F0F0F0F0
    F0F0F0F0,140,F0F0F0F0F0F0F0F0F0F0F0F0
    F0F0F0F0,141,0000000000000000F0F0F0F0
    F0F0F0F0,142,0000000000000000F0F0F0F0,
    143,0000000000F0F0F0F0
```

```
440 DATA 144,0000000000000000,151,0000
    0000000000000,152,0000000000000000,
    153,0000000000000000
450 DATA 154,00000000F0F0F0F0,155,0078
    4040404040404040,156,00F0404040404040,15
    7,00F8404040404040,93,0000
460 DATA 158,0000000000404FF,159,000000
    0404040FF,145,,98,00003C3C3C3C,91,0
    0000000000000006000,92,9162040810204080
    9F
470 GOSUB 4260
480 HY=1
490 FOR I=1 TO 2
500 FOR J=1 TO 22
510 H(1,I,J)=0.0
520 NEXT J
530 NEXT I
540 CALL CLEAR
550 CALL SCREEN(8)
560 PRINT "HYDROGRAPH #";STR$(HY):
570 PRINT "ENTER DATA:"::
580 INPUT "INTENSITY(IN/HR) ":IN(HY)
590 IN(HY)=RD2(IN(HY))
600 IF IN(HY)<1E4 THEN 630
610 GOSUB 4470
620 GOTO 580
630 PRINT
640 INPUT "DURATION(MIN) ":DU(HY)
650 DU(HY)=RD(DU(HY))
660 IF DU(HY)<1E5 THEN 690
670 GOSUB 4470
680 GOTO 640
690 PRINT
700 INPUT "LENGTH(FT) ":LE(HY)
710 LE(HY)=RD(LE(HY))
720 IF LE(HY)<1E6 THEN 750
730 GOSUB 4470
740 GOTO 700
750 PRINT
760 INPUT "WIDTH(FT) ":WI(HY)
770 WI(HY)=RD(WI(HY))
780 IF WI(HY)<1E6 THEN 810
790 GOSUB 4470
800 GOTO 760
810 PRINT
820 INPUT "SLOPE(FT/FT) ":SL(HY)
830 SL(HY)=RD6(SL(HY))
840 IF SL(HY)<=.04 THEN 910
850 CALL SOUND(200,120,1)
860 PRINT :" FOR ACCURATE RESULTS THE"
    :" VALUE FOR SLOPE SHOULD":" BE LE
    SS THAN 0.04.":
870 PRINT " DO YOU WISH TO RE-ENTER":"
    A VALUE FOR SLOPE?(Y/N)":::
880 GOSUB 4230
890 IF (KEY=89)+(KEY=78)=0 THEN 880
900 IF KEY=89 THEN 820
910 PRINT
920 FOR I=1 TO 350
930 NEXT I
940 CALL CLEAR
950 PRINT "ENTER ROUGHNESS FACTOR":
960 PRINT "USING TABLE AS A GUIDE":
    :
970 PRINT "SURFACE TYPE FACTOR
    ":::
980 PRINT "-------------------
    -------"
990 PRINT "SMOOTH ASPHALT PAVE..0.0070
    :::"TAR AND SAND PAVE.....0.0075":::
    "CRUSHED SLAG PAVE.....0.0120":::
1000 PRINT "CONCRETE
1010 IMPOTAR & GRAVEL BUILT-UP ROOF=.01
1020 PRINT "CLOSELY CLIPPED GRASS=0.02
1030 CR(HY)=RD(CR(HY))
1040 INPUT "ROUGHNESS FACTOR=";CR(HY
     )
1050 CR(HY)=RD4(CR(HY))
```

```
1040 IF CR(HY)<1 THEN 1070
1050 GOSUB 4470
1060 GOTO 1020
1070 FOR I=1 TO 250
1080 NEXT I
1090 CALL CLEAR
1100 CALL SCREEN(3)
1110 PRINT "                    COMPUTING":::::::::
     :::::
1120 QE(HY)=IN(HY)*LE(HY)/43200
1130 XX=((.0007*IN(HY))+CR(HY))/SL(HY)
     ^(1/3)
1140 DE=XX*LE(HY)*QE(HY)^(1/3)
1150 TE(HY)=DE/(30*QE(HY))
1160 KNT=0
1170 FOR J=1 TO 10
1180 IF (J/10*TE(HY))>DU(HY)THEN 1190 E
     LSE 1210
1190 KNT=J
1200 GOTO 1250
1210 H(HY,2,J)=J/10*TE(HY)
1220 NEXT J
1230 H(HY,2,11)=((DU(HY)-TE(HY))*0.5)+T
     E(HY)
1240 H(HY,2,12)=DU(HY)
1250 QW(HY)=QE(HY)*WI(HY)
1260 RESTORE 1270
1270 DATA .01,.06,.18,.38,.55,.7,.83,.9
     ,.94,1,1,1
1280 FOR J=1 TO 12
1290 IF (J>KNT)+(KNT<>0)=-2 THEN 1340
1300 READ MULT
1310 H(HY,1,J)=MULT*QW(HY)
1320 NEXT J
1330 H(HY,1,12)=QW(HY)
1340 IF KNT=0 THEN 1360
1350 QW(HY)=H(HY,1,KNT-1)
1360 DO=((CR(HY)/SL(HY)^(1/3))*LE(HY)*QE
     (HY)^(1/3)
1370 FOR I=1 TO 9
1380 H(HY,2,I+12)=(DO/(120*QW(HY)/WI(HY
     )))*((1-1/10)^(-2/3))-1)
1390 H(HY,2,I+12)=H(HY,2,I+12)+DU(HY)
1400 H(HY,1,I+12)=(1-I/10)*QW(HY)
1410 NEXT I
1420 H(HY,2,22)=(DO/(120*QW(HY)/WI(HY))
     )*6.36808064+DU(HY)
1430 TPP(HY,1)=0
1440 TPP(HY,2)=0
1450 H(HY,1,22)=.05*QW(HY)
1460 IF KNT<>0 THEN 1810
1470 TPP(HY,1)=.25*(DU(HY)-TE(HY))+TE(H
     Y)
1480 TPP(HY,2)=.75*(DU(HY)-TE(HY))+TE(H
     Y)
1490 GO TO 1810
1500 CALL CLEAR
1510 IF FILE=0 THEN 1530
1520 PRINT "          HYDROGRAPH PRINTING":::
     ::::::::::
1530 FOR I=1 TO HY
1540 PRINT #FILE:TAB(9);"HYDROGRAPH #";
     STR$(I);
1550 IF FILE=0 THEN 1570
1560 PRINT #FILE:
1570 PRINT #FILE:"  TIME(MIN)","DISCHAR
     GE(CFS)":
1580 IF FILE=0 THEN 1600
1590 PRINT #FILE:"  ----------","----------
     ":
1600 FOR J=1 TO 22
1610 IF J=11 THEN 1710
1620 IF H(I,2,J)=0 THEN 1710
1630 IL=4
1640 N=H(I,2,J)
1650 FL=1
1660 GOSUB 3210
1670 N=H(I,1,J)
1680 FL=3
1690 GOSUB 3210
1700 PRINT #FILE:TAB(4);ES;TAB(16);DS
1710 NEXT J
1720 IF FILE<>0 THEN 1770
1730 PRINT " PRESS ANY KEY TO CONTINUE"
     :
1740 GOSUB 4230
1750 CALL CLEAR
1760 GO TO 1780
1770 PRINT #FILE:::::
1780 NEXT I
1790 IF TST=1 THEN 1810
1800 GO TO 2030
1810 CALL CLEAR
1820 GOSUB 3720
1830 CALL SCREEN(15)
1840 PRINT " PRESS";TAB(9);"TO::
     ::TAB(9)::--::  1  DISPLAY DAT
     ::
1850 PRINT "  2  DISPLAY HYDROGRAPH"
     ::
1860 PRINT "  3  COMPUTE AMOTHER":TA
     B(10);"HYDROGRAPH AND":TAB(10);"CO
     MPARE":::
1870 PRINT "  4  REDIRECT OUTPUT":::"
     5  ENTER NEW PROBLEM":::"  6
     END PROGRAM":::::
1880 CALL SOUND(200,600,1)
1890 GOSUB 3760
1900 GOSUB 4230
1910 IF (KEY<49)+(KEY>54)=-1 THEN 1920
     ELSE 1940
1920 CALL SOUND(250,110,1)
1930 GO TO 1900
1940 CALL SOUND(100,666,1)
1950 CALL CLEAR
1960 CALL SCREEN(8)
1970 ON (KEY-48)GO TO 3470,3800,3650,20
     00,480,1930
1980 CALL CLEAR
1990 END
2000 GOSUB 4260
2010 GO TO 1810
2020 CALL SOUND(150,600,1)
2030 CALL CLEAR
2040 F2=0
2050 QMAX=QW(1)
2060 IF HY=1 THEN 2090
2070 IF QMAX>=QW(2)THEN 2090
2080 QMAX=QW(2)
2090 TMAX=H(1,2,22)
2100 IF HY=1 THEN 2230
2110 IF TMAX>=H(2,2,22)THEN 2130
2120 TMAX=H(2,2,22)
2130 CALL CLEAR
2140 IF QMAX>=10 THEN 2150 ELSE 2160
2150 QMAX=RD(QMAX)
2160 PRINT #F2:TAB(12);"LEGEND":TAB(12)
     ;"--  X = HYDROGRAPH #1":::"
     0 = HYDROGRAPH #2":::
2170 PRINT #F2:TAB(2);CHR$(64);" = COIN
     CIDENCE OF 1 & 2":::" MAX DISCHARGE
     (CFS)=";RD2(QMAX)::::::::::
2180 IF (FILE=0)+(F2>0)<0 THEN 2210
2190 F2=FILE
2200 GOTO 2160
2210 FOR I=1 TO 700
2220 NEXT I
2230 CALL CLEAR
2240 CALL SCREEN(8)
2250 FOR I=9 TO 16
2260 CALL COLOR(I-8,2,1)
2270 CALL COLOR(I,2,16)
2280 NEXT I
2290 CALL SOUND(150,600,1)
2300 FOR I=1 TO 20
2310 CALL HCHAR(I,9,145,20)
2320 NEXT I
```

```
2330 CALL VCHAR(1,9,161)
2340 CALL VCHAR(2,9,160,4)
2350 CALL VCHAR(6,9,161)
2360 CALL VCHAR(7,9,160,4)
2370 CALL VCHAR(11,9,161)
2380 CALL VCHAR(12,9,160,4)
2390 CALL VCHAR(16,9,161)
2400 CALL VCHAR(17,9,160,4)
2410 CALL VCHAR(20,9,99)
2420 CALL HCHAR(20,10,102,3)
2430 CALL HCHAR(20,13,105)
2440 CALL HCHAR(20,14,102,4)
2450 CALL HCHAR(20,18,103)
2460 CALL HCHAR(20,19,102,4)
2470 CALL HCHAR(20,23,103)
2480 CALL HCHAR(20,24,102,4)
2490 CALL HCHAR(20,28,103)
2500 GOSUB 3930
2510 FOR J=1 TO HY
2520 FOR I=1 TO 22
2530 IF (I=11)+(I=12)+(KNT=0)<>-2 THEN
     2570
2540 P1=TPP(J,I-10)/TMAX*19.99999
2550 P2=QW(J)/QMAX*19.99999
2560 GOSUB 2930
2570 P1=H(J,2,I)/TMAX*19.99999
2580 P2=H(J,1,I)/QMAX*19.99999
2590 GOSUB 2930
2600 NEXT I
2610 NEXT J
2620 IF FILE=0 THEN 2870
2630 NUS$="                    ***GRAPH PRINTING***
     "
2640 FOR I=1 TO LEN(NUS$)
2650 CALL VCHAR(24,I,ASC(SEG$(NUS$,I,1)
     ))
2660 NEXT I
2670 FOR I=1 TO 23
2680 B$=""
2690 FOR J=1 TO 32
2700 CALL GCHAR(I,J,A)
2710 IF (I=9)+(I>20)=-2 THEN 2820
2720 IF (I=20)+(I<9)+(J>28)=-2 THEN 282
     0
2730 GOSUB 4200
2740 IF (A=91)+(A=93)<>-1 THEN 2760
2750 A=145
2760 IF A<>92 THEN 2780
2770 A=57
2780 IF (A<104)+(A>154)+(A=145)=-1 THEN
     2820
2790 A=INT(A/10)+86
2800 IF A<99 THEN 2820
2810 A=98
2820 B$=B$&CHR$(A)
2830 NEXT J
2840 PRINT #FILE:B$
2850 NEXT I
2860 PRINT #FILE:::::
2870 ZZ$="  PRESS ANY KEY TO CONTINUE"
2880 FOR I=1 TO LEN(ZZ$)
2890 CALL VCHAR(24,I,ASC(SEG$(ZZ$,I,1))
     )
2900 NEXT I
2910 GOSUB 4230
2920 GOTO 1810
2930 T1=P1-INT(P1)
2940 T2=P2-INT(P2)
2950 Y=20-INT(P2)
2960 X=9+INT(P1)
3030 IF T2<.5 THEN 3080
3040 PS=1
3050 GOTO 3090
3060 PS=4
3070 GOTO 3090
3080 PS=3
3090 IF J=2 THEN 3120
3100 CHAR=PS+110
3110 GOTO 3180
3120 CALL GCHAR(Y,X,CHAR2)
3130 IF CHAR2=145 THEN 3170
3140 IF (CHAR2<108)+(CHAR2>103)=-2 THEN
     3200
3150 CHAR=CHAR2*10-990+PS
3160 GOTO 3180
3170 CHAR=103+PS
3180 CALL VCHAR(Y,X,CHAR)
3190 CALL SOUND(100,220*J,3)
3200 RETURN
3210 N2=10^FL
3220 N1=ABS(N)+.5/N2
3230 IP=INT(N1)
3240 FP=INT(N2*(N1-IP))+N2
3250 D$=STR$(IP)
3260 L=LEN(D$)
3270 IF (IL=0)+(IP=0)<>-2 THEN 3300
3280 D$="."
3290 L=0
3300 IF IL<L THEN 3400
3310 IF IL<=L THEN 3350
3320 D$="."&D$
3330 L=L+1
3340 GOTO 3310
3350 IF FL<=0 THEN 3390
3360 D$=D$&"."&SEG$(STR$(FP),2,FL)
3370 IF FL=3 THEN 3390
3380 E$=D$
3390 RETURN
3400 D$="*"
3410 FOR I=1 TO (IL+FL)
3420 D$=D$&"*"
3430 NEXT I
3440 IF FL=3 THEN 3460
3450 E$=D$
3460 RETURN
3470 CALL CLEAR
3480 IF FILE=0 THEN 3500
3490 PRINT "                    DATA PRINTING":::::
     :::::
3500 FOR I=1 TO HY
3510 PRINT #FILE:TAB(7);"DATA--HYDROGRA
     PH #";STR$(I)::
3520 PRINT #FILE:"GIVEN:":"---------":" IN
     TENSITY(IN/HR)=";IN(I)::" DURATIO
     N(MIN)=";DU(I)::
3530 PRINT #FILE:" LENGTH(FT)=      ";
     LE(I)::" WIDTH(FT)=      ";WI(I)
     ::" SLOPE(FT/FT)=      ";SL(I)::
3540 PRINT #FILE:" ROUGHNESS FACTOR=  ";
     CR(I)::
3550 PRINT #FILE:"CALCULATED:":"---------
     ---":" EQUILIB. TIME(MIN)=  ";RD(T
     E(I))::
3560 PRINT #FILE:" MAX DISCHARGE(CFS)=
     ";RD2(QW(I))::
3570 IF FILE<>0 THEN 3620
3580 PRINT "  PRESS ANY KEY TO CONTINUE"
     :
3590 GOSUB 4230
3600 CALL CLEAR
3610 GO TO 3630
3620 PRINT #FILE:::::
3630 NEXT I
3700 NEXT I
3710 GO TO 540
3720 FOR I=1 TO 8
```

```
3730 CALL COLOR(1,1,1)
3740 NEXT I
3750 RETURN
3760 FOR I=1 TO 8
3770 CALL COLOR(I,2,1)
3780 NEXT I
3790 RETURN
3800 CALL CLEAR
3810 GOSUB 3720
3820 PRINT TAB(6);"HYDROGRAPH DISPLAY":
     TAB(7);"PRESS FOR":::TAB(9);"1 TAB
     (6);"LE":
3830 PRINT :TAB(9);"2 GRAPH":::TAB(9
     );"3 BOTH":::::::
3840 CALL SOUND(200,600,1)
3850 GOSUB 3760
3860 GOSUB 4230
3870 IF (KEY<49)+(KEY>51)=-1 THEN 3880
     ELSE 3900
3880 CALL SOUND(250,110,1)
3890 GOTO 3860
3900 CALL SOUND(150,666,1)
3910 TST=KEY-48
3920 ON TST GOTO 1500,2030,1500
3930 T13$="100 75 50 25 0"
3940 FOR I=1 TO 3
3950 FOR J=1 TO 3
3960 CALL VCHAR(5+I-4,J+5,ASC(SEG$(T13$
     ,5+I+J-3,1)))
3970 NEXT J
3980 NEXT I
3990 SC=1
4000 IF TMAX<=132.666 THEN 4020
4010 SC=10
4020 FOR I=1 TO 4
4030 TMS=STR$(I*TMAX/(4*SC)+.5)
4040 CALL HCHAR(21,I*5+8,ASC(SEG$(TMS,1
     ,1)))
4050 IF SEG$(TMS,2,1)="." THEN 4070
4060 CALL HCHAR(21,I*5+9,ASC(SEG$(TMS,2
     ,1)))
4070 IF (SC=1)+(I=4)+(TMAX>99.499999999)
     <>-3 THEN 4090
4080 CALL HCHAR(21,30,ASC(SEG$(TMS,3,1)
     ))
4090 NEXT I
4100 T12$=CHR$(91)&CHR$(92)&CHR$(93)&"O
     F MAX DISCHARGE"
```

```
4110 FOR I=1 TO LEN(T12$)
4120 CALL VCHAR(I+1,5,ASC(SEG$(T12$,I,1
     )))
4130 NEXT I
4140 T14$="TIME(MIN)"
4150 IF SC=1 THEN 4170
4160 T14$="TIME(MIN) X 10"
4170 FOR I=1 TO LEN(T14$)
4180 CALL VCHAR(23,I+11,ASC(SEG$(T14$,I
     ,1)))
4190 NEXT I
4200 IF (I<>20)+(J<>9)=-2 THEN 4220
4210 A=A+56
4220 RETURN
4230 CALL KEY(0,KEY,ST)
4240 IF ST<=0 THEN 4230
4250 RETURN
4260 CALL CLEAR
4270 GOSUB 3720
4280 PRINT TAB(4);"OUTPUT DESTINATION":
     TAB(4);"
4290 PRINT "PRESS FOR":::"1
     SCREEN":::"2 THERMAL PRI
     NTER":::::
4300 GOSUB 3760
4310 GOSUB 4230
4320 IF (KEY<49)+(KEY>50)=-1 THEN 4330
     ELSE 4350
4330 CALL SOUND(250,110,1)
4340 GOTO 4310
4350 CALL SOUND(100,666,1)
4360 FILE=0
4370 IF KEY=49 THEN 4460
4380 FILE=1
4390 DVCS="TP.U.S"
4400 IF KEY=50 THEN 4420
4410 DVCS="RS232"
4420 IF DFG$="" THEN 4460
4430 CLOSE #1
4440 OPEN #1:DVCS,OUTPUT
4450 DFG$="1"
4460 RETURN
4470 PRINT :
4480 CALL SOUND(300,110,1)
4490 PRINT " NUMBER OUTSIDE NORMAL RANG
     E":
4500 PRINT " ENTER":
4510 RETURN
```

Programming

PRINTER GRAPHICS

TI BASIC

The special block graphics character sets that are built into some printers can be extremely useful. In the business world, for example, applications might include the production of charts and graphs, the printing of business forms, or even the design of a letterhead.

The following short program demonstrates how DATA statements are used to format selected graphics characters to produce a letterhead. The DATA statements here are for use with the Epson MX-80 printer (without the GRAFTRAX option) but can be easily modified to accommodate any printer with similar block graphics capabilities. Keep in mind that this is a *shell* program; you can plan the DATA statements to direct the printer to produce virtually any design or pattern (within the limits of the resident block graphics set). The actual graphic design (the letterhead) in this example is unimportant; understanding how to plan and implement it is crucial.

DATA statements are read sequentially from left to right, using the READ statement. The Epson MX-80 printer uses numerical codes 160 to 223 (ASCII 32 to 95 with a 1 for the 8th bit) to generate graphics characters already defined within the printer. Each graphics character is made up of one to six squares within a 2×3 matrix as indicated below.

1	2
4	8
16	32

(The numbers within the squares are not important if you have a coding table in front of you. They represent a particular manufacturer's coding of the matrix print head. For example, numerical code 165 would produce the fifth character in the set, and would cause wires 1 and 4 to fire; if we want the 21st character, wires 1, 4, and 16 would be fired.)

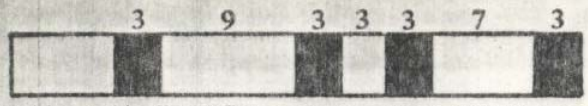
The key part of the program lies in lines 430-470. This controls what will happen when a DATA statement is read. If the first DATA cell is a number greater than 100, that character will be printed. If the first DATA cell is a number greater than 0 but less than 100, the program will read the second DATA cell, and then print it the number of times specified in the first DATA cell. For example, if the first DATA element is 8, and the next is 160 (a blank space), the computer will print a blank space 8 times. This lessens the amount of required DATA when it is necessary to repeat the same character several times. If the first DATA cell read is equal to 0, a Carriage Return will be executed.

Regular text can be printed on the same line with the graphics. In this program a negative number inserted in the DATA statement signals the program to print the text. The value of the negative number designates which message is to be printed out. This can also be used to change the printer's type style, if your printer has that capability. For example, if you want to print a graphics pattern on the left, and a printed message on the right, you would place the negative number just before the zero that causes the Carriage Return. In that case, a message will be printed—according to the directions in line 470—on the same line *before* the Carriage Return.

Every time a character is printed—whether graphics, text, or a control character—it should always be followed by a semicolon. This will insure that the next printed character will be on the same line, until the Carriage Return is executed. The only exception to this is in line 610 where I wanted the Carriage Return to be executed.

The following is an example of the DATA and the graphics line it creates. This line can be found in the 8th print line of the letterhead.

```
270  DATA 7,160,3,223,9,160,3,223,3,160,3,223,7
     160,3,223,-4,0
```



In this graphics line, I first needed to put seven blank spaces between the first character position and the first graphics character. This was done with the first two DATA cells. When the program first READs A, its value is 7. Because this value is less than 100 and greater than 0, the program will READ B, the next DATA cell. The value of B is 160. ASCII (160) is a blank character on the MX-80, so the program will now PRINT B (blank), A (7) times. On the next cycle the value stored in A will be 3, and the value stored in B will be 223. This will cause B (whose value is the ASCII code for a solid 2×3 block) to be printed 3 times. This process is continued until a value less than or equal to zero is encountered.

If the value in A is a negative number, the program will branch off to a subroutine which will PRINT a text message. In the above example, the value −4 caused the message "FOR USERS of TI-99/4" to be printed. These subroutines are extremely versatile; you can change the type style, print a message as I have done, continue the program to do calculations, or run program lines.

Note: All the data necessary for one entire print line is contained in a *single* DATA statement (except for lines 310-320 and lines 330-340). This makes the program a little easier to debug, because you don't have the confusion of counting across statement boundaries to find character positions and their corresponding codes.

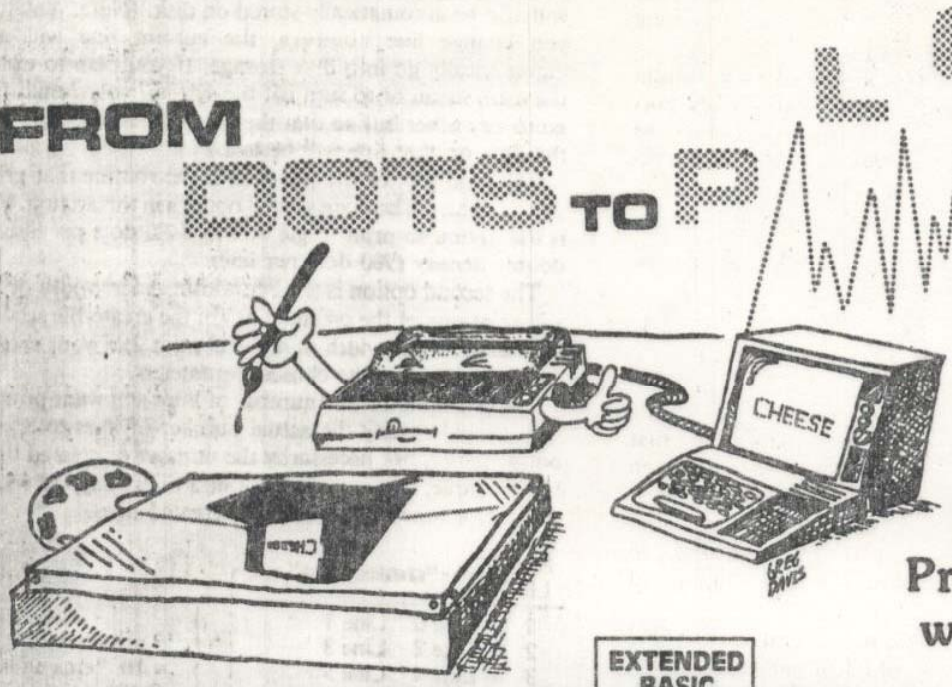
EXPLANATION OF THE PROGRAM

LETTERHEAD

Line Nos.	
200-390	Contains DATA formated to print the *99'er Magazine* letterhead.
400	OPENs a line to the RS-232 interface for output to the printer.
410	Sets the printer for "Double Strike" mode.
420	Sets the printer for "Emphasized" mode.
430	READs the DATA statement and stores the result in A.
440	Tests A; if A is greater than 100, then PRINT the character stored in A.
450	Tests A; if A is greater than 0 and less than 100, then READ B; PRINT B, A times.
460	Tests A; if A is equal to zero, then do a Carriage Return. This marks the end of a line.
470	A equals a negative number at this point; ABS(A) controls the branching of subroutines for special tasks, e.g., PRINT text.
490-500	Subroutine to execute the Carriage Return.
510-520	Subroutine to PRINT the value in A.
530-570	Subroutine to READ B, and PRINT B, A times.
580-710	Subroutine to print normal text instead of graphics.
720-740	End-of-print message on the screen; END of program.

```
100 REM  ••••••••••••••••
110 REM  ••              ••
120 REM  ••    99'ER     ••
130 REM  ••              ••
140 REM  ••LETTERHEAD    ••
150 REM  ••              ••
160 REM  ••••••••••••••••
170 REM
180 REM
190 REM
200 DATA 10,223,2,160,10,223,160,223,2
    21,0
210 DATA 2,223,6,160,2,223,2,160,2,223
    ,6,160,2,223,160,162,223,0
220 DATA 2,223,6,160,2,223,2,160,2,223
    ,6,160,2,223,0
230 DATA 2,223,6,160,2,223,2,160,2,223
    ,6,160,2,223,3,160,8,223,2,160,2,2
    23,2,160,216,2,223,0
240 DATA 10,223,2,160,10,223,3,160,2,2
    23,4,160,2,223,2,160,2,223,192,222
    ,167,2,163,0
250 DATA 7,160,3,223,9,160,3,223,3,160
    ,2,223,4,160,2,223,2,160,3,223,161
    ,0
260 DATA 7,160,3,223,9,160,3,223,3,160
    ,8,223,2,160,3,223,0
270 DATA 7,160,3,223,9,160,3,223,3,160
    ,3,223,7,160,3,223,-4,0
280 DATA 7,160,3,223,9,160,3,223,3,160
    ,3,223,7,160,3,223,-5,0
290 DATA 7,160,3,223,9,160,3,223,3,160
    ,8,223,2,160,3,223,-6,0
300 DATA -7,0
310 DATA 25,160,183,180,182,181,160,19
    2,166,196,2,160,183,2,163,165,160,
    192,166,196,2,160,2,163,211
320 DATA 165,160,162,203,163,160,183,1
    96,160,181,160,183,2,163,165,0
330 DATA 25,160,181,162,160,181,160,18
    9,172,172,181,160,213,208,210,181,
    160,189
340 DATA 2,172,181,160,220,211,208,176
    ,160,192,218,208,160,181,162,196,1
    81,160,215,211,209,180,0
350 DATA 0
360 DATA -1,0
370 DATA -2,0
380 DATA -3,0
390 DATA -8
400 OPEN #1:"RS232.BA=9600.DA=8.PA=N"
410 PRINT #1:CHR$(27);"G"
420 PRINT #1:CHR$(27);"E"
430 READ A
440 IF A>100 THEN 510
450 IF A>0 THEN 530
460 IF A=0 THEN 490
470 ON ABS(A)GOSUB 580,600,620,640,660
    ,680,700,720
480 GOTO 430
490 PRINT #1
500 GOTO 430
510 PRINT #1:CHR$(A);
520 GOTO 430
530 READ B
540 FOR X=1 TO A
550 PRINT #1:CHR$(B);
560 NEXT X
570 GOTO 430
580 PRINT #1:TAB(25);"EMERALD VALLEY P
    UBLISHING CO.";
590 RETURN
600 PRINT #1:TAB(33);"P.O. BOX 5537";
610 RETURN
620 PRINT #1:TAB(29);"EUGENE, OREGON
    97405";
630 RETURN
640 PRINT #1:TAB(41);"FOR USERS OF TI-
    99/4";
650 RETURN
660 PRINT #1:TAB(41);"AND OTHER TMS990
    -BASED";
670 RETURN
680 PRINT #1:TAB(41);"PERSONAL COMPUTE
    R SYSTEMS";
690 RETURN
700 PRINT #1:TAB(63);"TM";
710 RETURN
720 PRINT "ALL DONE WITH LETTER HEAD"
730 CLOSE #1
740 END
```

FROM DOTS TO PLOTS:



Using Bit-Plot Printer Graphics with a TI-99/4A

EXTENDED BASIC

With TI's 99/4 Impact Printer, we can explore the world of *bit plot* printer graphics. The following program will also work with other printers (Epson's MX-80 with the Graftrax-80 option, for instance) when suitable modifications are made to the program.

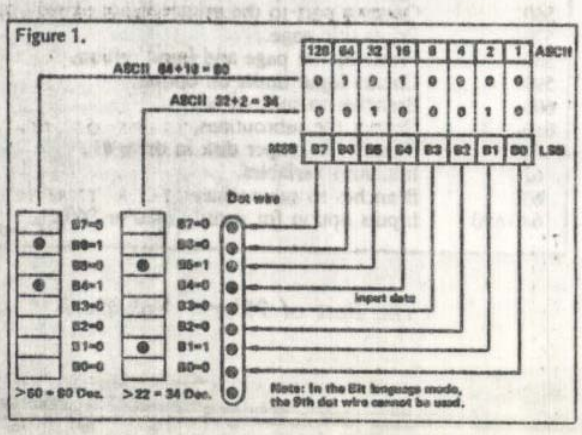
Bit-Plot Graphics

In bit-image mode, the printer produces in one dot-column a character which may have any combination of the eight dots in the printhead. This makes it possible to duplicate exactly the 8 × 8 pixel graphics characters of the TI-99/4 and TI 99/4A by printing 8 columns of up to 8 dots on the printer. The printer dots are turned on in accordance with a binary format. For example, sending CHR$(0) to the printer will produce a blank space, one dot-column wide; CHR$(1) will print only the bottom dot; CHR$(7) will print the bottom three dots, CHR$(255) will print all 8 dots, and so forth (see Figure 1). Under software control you may select either 480 or 960 dot-per-line resolution. This means that to print a full line in the 960 dot mode you would have to print a dot-column character 960 times. The following program, for example, would print a line with only the bottom dot "on" across the entire page width:

```
10 FOR X = 1 TO 960
20 PRINT #1: CHR$(1)
30 NEXT X
40 END
```

In the 960 mode, the line would appear solid with no space between the dots. In the 480 mode, the line would have small but visible spaces between the dots.

Figure 1.



[Note: There are two options in the 960 mode: the first at *half* the speed of the 480 mode, and the second at the *same* speed. This second mode may only be used by high-speed Assembly Language driver routines. BASIC interpreters are too slow in execution to print in this mode. If you try this mode using Extended BASIC you will lose many of the dots on each line. When you use this high-speed mode, there is still another restriction: The same needle may not be struck twice in a row because the needles take 2 microseconds to hit and return to seat. Printing at 480 speed, the print head passes over a dot portion every microsecond. For this reason it is impossible to strike the same needle twice in a row at this high speed. If you attempt a second stike, the printer will automatically toss away the second consecutive dot. The printer will also print bi-directionally in this mode. It should be noted that there is some misalignment between passes of the printhead from opposite directions. This will vary from printer to printer and must be compensated for with computer software.—Ed.]

To leave the TI-99/4's standard text mode and enter the bit-image graphics mode, you must first send CHR$(27); "L" or "K"; CHR$(X); CHR(Y); to the printer. The ESCape "K" code will assign the 480 mode to the printer; "L" will assign it the 960 mode. You must then tell the printer how many graphics columns or characters are to be printed. This is done with CHR$(X);CHR$(Y) where 0<X<255, and 0<Y<3. The number of columns of dots or characters to follow is equal to (Y*256)+X.

The only problem I have encountered with this convention is a difficulty with intermixing graphics and standard characters on the same line without complicated program-

ming and file structures. The simplest method is to store a CHR$(X) in a file or data statement and then print CHR$(X) for each dot column across the page. This may be time consuming and require more disk, tape or data space, but it allows for the least complicated program [and consequently is the simplest way to get you started using this versatile graphics mode.—Ed.]

To program the graphics you must know how to format the OPEN statement. First, you must tell the RS232 port to output 8 bits instead of 7; then tell it to supress the automatic carriage return and linefeed with the .CRLF software switch. The statement in line 560 will read:

```
OPEN #3: "RS232.BA=9600.DA=8.CRLF"
```

[If you have the Epson MX-80 with the Graftrax-80 option, it will read:

```
OPEN #3:
"RS232.BA=9600.DA=8.PA=N.CRLF".—Ed.]
```

The Program

There are three main sections in the program. The first part is a disk initialization subroutine. This routine will open a file on a blank disk with the following parameters: RELATIVE—random access of file records, and INTERNAL,FIXED 24—a fixed record length of 24 to store 12 CHAR$(X) values or 12 dot-columns of information.

It is possible to store up to 3570 such records on one 5¼" single-density disk. The process of initialization is therefore very slow and takes about half an hour. The initialization program will open the file and print CHR$(0) to all records. This helps speed file building in the second section of the program. When a large clear space on the paper is required, you do not need to enter all these zeros.

The second section of the program is a form of "word processor," only here it is designed to handle *numbers* from 0 to 255. The program works with 20 file records at a time (240 character variables). Each group of 240 variables will be called one *created line*. The present line being worked on is displayed at the top of the screen. The next value displayed is the position in that line, from 1 to 240. Below the position indicator, the present CHR$ value at that position is displayed. Below that, the computer asks you for the new value—from 0 to 255—that you want to assign to that position. Several single-keystroke commands are available to help you manipulate the data. If you merely press ENTER without touching any other key, the value of 0 will be assigned to the position indicated. The following is a list of commands and their explanations:

P—prints one line of data (240 dot positions of the line you are presently working on).
L—lists all 240 variables on the screen for inspection.
N—lets you jump to a new line number and position—a process that would take too long with the arrow keys alone (below). Screen prompts will guide you. If you just hit ENTER, the program will default to the current line number or position without changing anything.
E—decrements the line number by 1.
X—increments the line number by 1.
S—decrements the position in a line by 1
D—increments the position in a line by 1.
Z—returns the user to the main menu screen.

After you enter a valid numeric value and press ENTER, the position in the line will automatically increment by 1 to the next record. After you enter the 240th record of the line, the previous line number will increment to the next line, and the position will return to number 1. The previous line will also be automatically stored on disk. (Note: Any time you change line numbers, the current line will also automatically go into disk storage. If you plan to exit to the main menu or to turn off the system, you should first go to any other line so that the data are stored; otherwise the data on that line will be lost.)

The final part of the program is the routine that prints your graphics. There are several options in this section. First is the option to print single density (480 dots per line) or double density (960 dots per line).

The second option is the line width: A line width of 240 will print one of the created lines in the create-file section, (240 dots); a line width of 480 will print 2 of your created lines according to the chosen parameters.

The last option is the number of lines you want printed: You should specify the actual number of lines to appear on the paper, not necessarily the number of created lines. For example, if you want to print 5 lines with 480 width, you will actually be printing 10 created lines.

Print Line	"created lines"			
1	Line 0	Line 1		5 printer lines = 10 "created lines" @ 480 width
2	Line 2	Line 3		
3	Line 4	Line 5		
4	Line 6	Line 7		
5	Line 8	Line 9		
1	Line 0	Line 1	Line 2	3 printer lines = 9 "created lines" @ 720 width
2	Line 3	Line 4	Line 5	
3	Line 6	Line 7	Line 8	

EXPLANATION OF THE PROGRAM
DOTS TO PLOTS

Line Nos.	
100-170	REM.
180-190	Initializes variables and arrays.
210-290	Data statement.
300	Subroutine to read data and display.
310	Subroutine to read data, display and accept input.
320	Reads data only.
330	Initializes colors.
340-380	Checks for proper disk in drive #1.
390-400	Opens file #2 on disk #1.
410	Clears screen with left to right scroll.
420-430	Controls subroutine to check for proper disk in drive #1.
440	Reads multiple data statements and display.
450-470	Inputs record from disk #1.
480-500	Prints record on disk #1.
510-550	Improper disk in drive #1 message; option to try again.
560	Opens a port to the printer if not already open.
570	Prints title page.
580	Prints option page and input option.
590	Checks input limits on option.
600	Branches to subroutines.
610-1120	Creates file subroutines.
610	Checks for proper disk in drive #1.
620	Initializes variables.
630	Branches to subroutines.
640-690	Inputs option for density (480 or 960).

Continued

EXPLANATION Continued

700	Clears screen. Inputs first record.
710-720	Displays record variables.
730	Inputs new value or command.
740-830	Checks for a command input and does necessary logic.
840-860	Checks that all characters in the new value are numeric.
870-920	Assigns new value to array; advances to next line position, and checks for end of line.
940-970	Subroutine to enter new line number and new position.
980-1030	Subroutine to print one line (CHR$(X) = 240).
1040-1110	Subroutine to display array contents on the screen.
1110	Subroutine to convert the input string into ASCII form and store it in the array.
1120	Subroutine to re-convert the array into a string for output to the disk.
1130-1270	Subroutine to print entire graphics page from information stored on disk.
1130	Checks for file on disk.
1140	Inputs density (480, 960).
1150-1160	Inputs width of graphics in dot columns.
1170-1180	Inputs number of lines to be printed.
1190-1270	Prints file from disk onto the printer in the form of bit-image graphics.
1280-1340	Initializes a new disk with all CHR$(0); will destroy any records stored on that file.
1350	Closes files and ends.

```
100 REM ***********
110 REM * DOTS TO PLOTS *
120 REM ***********
130 REM
140 REM
150 REM
160 REM
170 REM
180 OP2=0 :: OP3=0
190 DIM Z(12,20)
200 GOTO 370
210 DATA 1,8,DOTS TO PLOTS,3,9,SUBROUTINES
220 DATA 1,11,MENU,3,3,1.CREATE DATA FIELD,5,3,2.PRINT DATA FIELD,7,3,3.INITIALIZE NEW DISK,9,3,4.EXIT
230 DATA 1,5,CREATE DATA FIELD
240 DATA 1,9,PRINT DATA,3,3,FILE NAME?
250 DATA 23,3,YOUR CHOICE?
260 DATA 20,3,FILE NOT ON DISK
270 DATA 3,5,PLACE DISK IN DSK1,6,5,DISK IS NOT BLANK
280 DATA 24,3,PRESS ENTER TO CONTINUE
290 DATA 12,1,NEW LINE:,13,1,NEW POS.:,12,1,"",13,1,""
300 READ A1,A2,A$ :: DISPLAY AT(A1,A2):A$ :: RETURN
310 READ A1,A2,A$ :: DISPLAY AT(A1,A2):A$ :: ACCEPT AT(A1,A2+LEN(A$)+1):ANS$ :: RETURN
320 READ A1,A2,A$ :: RETURN
330 CALL SCREEN(5):: FOR A=1 TO 8 :: CALL COLOR(A,2,6):: NEXT A :: RETURN
340 OPEN #1:"DSK1.",RELATIVE,INTERNAL,INPUT
350 INPUT #1:AS,A,A,A
360 INPUT #1:A$,:: IF A$="" THEN CLOSE #1 :: GOTO 370 ELSE RETURN
370 IF A$=ANS$ THEN CLOSE #1 :: GOTO 390 ELSE IF A$<>"." THEN 540
380 RESTORE 260 :: GOSUB 300 :: RETURN
390 IF OP2=1 THEN 400 :: OPEN #2:"DSK1."&ANS$,RELATIVE,INTERNAL,FIXED 24 :: OP2=1
400 RETURN
410 CALL COLOR(9,5,5):: CALL VCHAR(1,1,99,96):: CALL VCHAR(1,3,32,672):: RETURN
420 GOSUB 410 :: RESTORE 270 :: GOSUB 300 :: RESTORE 280 :: GOSUB 310
430 RESTORE 240 :: GOSUB 320 :: GOSUB 310 :: GOSUB 340 :: RETURN
440 FOR T1=1 TO T :: READ A1,A2,A$ :: DISPLAY AT(A1,A2):A$ :: NEXT T1 :: RETURN
450 FOR X=1 TO 20
460 INPUT #2,REC L*20+X:@$ :: GOSUB 1110
470 NEXT X :: RETURN
480 FOR X=1 TO 20
490 GOSUB 1120 :: PRINT #2,REC L*20+X:@$
500 NEXT X :: RETURN
510 IF PG=2 THEN GOSUB 390 :: RETURN ELSE DISPLAY AT(6,5):"DISK IS BLANK"
520 DISPLAY AT(7,1):"DO YOU WISH TO TRY AGAIN? (Y/N)." :: ACCEPT AT(8,8)VALIDATE("YN"):ANS$
530 IF ANS$="Y" THEN 340 ELSE 1350
540 IF PG=1 THEN DISPLAY AT(6,1):"FILE NOT PRESENT" :: GOTO 520
550 DISPLAY AT(6,1):"DISK IS NOT BLANK" :: GOTO 520
560 IF OP3=1 THEN RETURN ELSE OPEN #3:"RS232.BA=9600.DA=8.PA=N.CRLF" :: OP3=1 :: RETURN
570 GOSUB 350 :: GOSUB 410 :: RESTORE 210 :: T=2 :: GOSUB 440 :: RESTORE 280 :: GOSUB 310
580 GOSUB 410 :: RESTORE 220 :: T=5 :: GOSUB 440 :: RESTORE 250 :: GOSUB 310
590 IF ASC(ANS$)<48 OR ASC(ANS$)>55 THEN RESTORE 250 :: GOSUB 310 :: GOTO 590
600 ON VAL(ANS$)GOTO 610,1150,1280,1350
610 PG=1 :: GOSUB 410 :: RESTORE 250 :: GOSUB 300 :: GOSUB 420
620 L=0 :: P=1 :: P1=1
630 GOSUB 390 :: GOSUB 640 :: GOTO 700
640 DISPLAY AT(2,1):"WHAT DENSITY PER LINE?"
650 DISPLAY AT(3,1):"1. 480" :: "2. 960"
660 ACCEPT AT(5,1)VALIDATE("12"):DS :: D=VAL(DS)
670 IF D=1 THEN DS="K" ELSE DS="L"
680 D=D*2
690 RETURN
700 GOSUB 410 :: GOSUB 450
710 DISPLAY AT(1,1):"LINE #:" :: "POSITION:" :: "OLD VALUE:" :: "NEW VALUE:"
720 DISPLAY AT(1,9):L :: DISPLAY AT(3,11):((P1-1)*12+P :: DISPLAY AT(5,12):Z(P,P1)
730 ACCEPT AT(7,12):NV$
740 IF NV$<>"S" THEN 770 ELSE IF P>1 THEN P=P-1 :: GOTO 720
750 P1=P1-1 :: P=12 :: IF L=0 THEN 720 ELSE IF P1<1 THEN GOSUB 480 :: L=L-1 :: P1=20 :: GOSUB 450 :: GOTO 720
760 GOSUB 450 :: GOTO 720
770 IF NV$="D" THEN IF P<12 THEN P=P+1 :: GOTO 720 ELSE P1=P1+1 :: P=1 :: IF P1>20 THEN GOSUB 480 :: L=L+1 :: P1=1 :: GOSUB 450 :: GOTO 720
780 IF NV$="E" AND L>0 THEN GOSUB 480 :: L=L-1 :: GOSUB 450 :: P1=1 :: P=1 :: GOTO 720
```

```
 700 IF NV$="X" AND L<100 THEN GOSUB 480
     0 :: L=L+1 :: GOSUB 450 :: P1=1 ::
     P=1 :: GOTO 720
 800 IF NV$="P" THEN GOSUB 560 :: GOSUB
     980 :: GOTO 720
 810 IF NV$="L" THEN GOSUB 1040 :: GOTO
     710
 820 IF NV$="Z" THEN 530
 830 IF NV$="N" THEN GOSUB 940 :: GOSUB
     300 :: GOSUB 300 :: GOTO 710
 840 FOR TL=1 TO LEN(NV$)
 850 IF ASC(SEG$(NV$,TL,1))<48 OR ASC(S
     EG$(NV$,TL,1))>57 THEN GOTO 720
 860 NEXT TL
 870 IF NV$="." THEN NV$="0."
 880 NV=VAL(NV$) :: IF NV<0 OR NV>255 TH
     EN 720
 890 Z(P,P1)=NV
 900 P=P+1
 910 IF P>12 THEN P1=P1+1 :: P=1 :: IF
     P1>20 AND L<100 THEN GOSUB 480 ::
     L=L+1 :: GOSUB 450 :: P1=1 :: P=1
     :: GOTO 720
 920 GOTO 720
 930 REM
 940 RESTORE 290 :: GOSUB 310 :: IF ANS
     $="" THEN NLINE=L :: GOTO 950 ELSE
     IF VAL(ANS$)<=189/D THEN NLINE=VA
     L(ANS$)ELSE GOTO 940
 950 GOSUB 310 :: IF ANS$="" THEN LPOS=
     (P1-1)*12+P ELSE IF VAL(ANS$)<=240
     THEN LPOS=VAL(ANS$)ELSE GOTO 950
 960 IF NLINE>189/D OR NLINE<0 OR LPOS>
     240 OR LPOS<1 THEN 940
 970 GOSUB 480 :: L=NLINE :: GOSUB 450
     :: P1=INT((LPOS+1)/12)::P=((LPOS+
     1)/12-(P1*20))*12 :: RETURN
 980 PRINT #3:CHR$(27);D$;CHR$(240);CHR
     $(0);
 990 FOR X=1 TO 20
1000 PRINT #3:CHR$(Z(1,X));CHR$(Z(2,X))
     ;CHR$(Z(3,X));CHR$(Z(4,X));CHR$(Z(
     5,X));CHR$(Z(6,X));CHR$(Z(7,X));
1010 PRINT #3:CHR$(Z(8,X));CHR$(Z(9,X))
     ;CHR$(Z(10,X));CHR$(Z(11,X));CHR$(
     Z(12,X));
1020 NEXT X
1030 CLOSE #3 :: OP3=0 :: RETURN
1040 TB=1 :: LN=1 :: LI=9 :: GOSUB 1060
     :: LN=10 :: LI=18 :: GOSUB 1060 :
     : LN=19 :: LI=20 :: GOSUB 1060
1050 GOSUB 410 :: RETURN
1060 FOR Y=LN TO LI
1070 FOR X=1 TO 12 :: PRINT TAB(TB+S-4)
     ;Z(X,Y); :: TB=TB+1 :: IF TB=6 THEN
     TB=1
1080 NEXT X :: NEXT Y :: PRINT :: PRINT
1090 RESTORE 280 :: GOSUB 310
1100 RETURN
1110 FOR @=1 TO 12 :: Z(@,X)=ASC(SEG$(@
     $,@,1)) :: NEXT @ :: RETURN
1120 @$="" :: FOR @=1 TO 12 :: @$=@$&CH
     R$(Z(@,X)):: NEXT @ :: RETURN
1130 GOSUB 410 :: RESTORE 240 :: GOSUB
     300 :: GOSUB 310 :: GOSUB 590
1140 GOSUB 640
1150 DISPLAY AT(7,1):"WIDTH OF GRAPHICS
     ":"1. 240":"2. 480":"3. 720 (960
     RES. ONLY)":"4. 960 (960 RES. ONLY
     )"
1160 ACCEPT AT(12,1)VALIDATE("1234"):WI
     DTH$ :: W=VAL(WIDTH$)
1170 DISPLAY AT(14,1):"NUMBER OF LINES?
     (1-";INT(189/W);")"
1180 ACCEPT AT(15,1)VALIDATE(DIGIT):NOL
     :: IF NOL>INT(189/W)THEN 1180
1190 DISPLAY AT(18,10):"PRINTING"
1200 GOSUB 560 :: PRINT #3:CHR$(27);"A"
     ;CHR$(8);
1210 FOR PRNT=0 TO NOL-1
1220 FOR PRIT=1 TO W
1230 PRINT #3:CHR$(27);D$;CHR$(240);CHR
     $(0);
1240 L=PRNT*W+PRIT-1 :: FOR X=1 TO 20 :
     : INPUT #2,REC L*20+X:@$ :: PRINT
     #3:@$;:: NEXT X
1250 NEXT PRIT :: PRINT #3:""
1260 NEXT PRNT
1270 CLOSE #3 :: OP3=0 :: GOTO 580
1280 GOSUB 410 :: DISPLAY AT(1,1):"INIT
     IALIZATION WILL DESTROY":"ANY RECO
     RDS BEING STORED":"ON THE FILE"
1290 DISPLAY AT(4,1):"DO YOU WISH TO CO
     NTINUE?":"(Y/N)"
1300 ACCEPT AT(5,7)VALIDATE("YN"):ANS$ :
     IF ANS$="N" THEN 570
1310 PG=2 :: GOSUB 410 :: DISPLAY AT(2,
     1):"PLACE BLANK DISK IN DRIVE #1"
     :: RESTORE 280 :: GOSUB 310
1320 GOSUB 430
1330 FOR X=1 TO 12 :: @$=@$&CHR$(0)::N
     EXT X
1340 FOR X=1 TO 3570 :: PRINT #2:@$ ::
     NEXT X :: CLOSE #2 :: GOTO 570
1350 IF OP2=1 THEN CLOSE #2
1360 IF OP3=1 THEN CLOSE #3
1370 END
```

Personal Record Keeping

Managing a Mobile Home Park

First of all, this true story has a moral to it, so we might as well get it out of the way now:

"Before going to all the work of writing a program to do a job, find out if a TI Command Cartridge can do the job for you."

The TI Command Cartridges are well written, almost totally error-free, and have been engineered for ease of use by non-programmers. Let's talk about one of these little jewels:

The *Personal Record Keeping (PRK)* Command Cartridge, when combined with your imagination, is a very flexible and powerful tool. In order to fully utilize this power, however, your TI-99/4A system should include a printer. The TI Thermal Printer works well and is probably the easiest and least expensive to use, but I chose a more expensive route: an Epson MX-80 printer operating through the RS232 Interface. This gives me a bit more power (e.g., longer print lines) for the *PRK*'s report formating. For most applications, you will also need either a cassette recorder or disk system to store your data files.

Before trying to set up and work with a data file using the *PRK*, you should carefully read the manual and all the examples that come with the cartridge. When you have done that, take a break, come back a little later, and do it again. That mild-mannered little *PRK* manual contains the answers to questions that will surely pop up when you start designing the solution to *your* problem. So keep it handy!

OK, now comes the real challenge. How do we decide that a problem can be solved using the *PRK* cartridge? First, we must completely describe the problem. Second, we must break the problem down into subproblems or tasks. Third, we identify the tasks that can be performed with the *PRK* cartridge. Fourth, we see if enough of the

And Now For Our Story . . .

Recently I had a customer ask me how much I would charge to write a program for him. I told him that it is difficult to determine until the program is completed, but he could figure on $15.00 per hour for a minimum of 10 hours. At that point, he decided to be brave and tackle the program himself. Of course, I was curious, so I asked him what he wanted the TI BASIC program to do.

He told me that he was the owner of a mobile home park. Each month he had to figure out the bill for each individual renter in the park. He wanted the TI-99/4A to save him time and decrease the chance for errors. After thinking over this problem for a minute, I asked him for details: What did he do to accomplish the job himself?

First, he walked around to each trailer space and copied the electric meter and gas meter readings into his notebook. A computer system could be designed to do this task but it would take extremely expensive peripheral hardware. . .

When I asked him what else was in the notebook that he used for this job, he said that it contained all the previous electric and gas meter readings. It also contained miscellaneous charges for each renter, the electric and gas rates, and the actual space rental fees. At this point it was obvious to me that the computer could easily act as a notebook and store all that data on cassette tape or floppy disk.

Next, he sat down at his desk with the notebook, pencil, paper and a calculator. For each trailer space, he performed the following calculations:

```
GAS BILL      = (CUR. GAS METER - PREV. GAS METER) x GAS RATE
              + GAS METER USE FEE
ELECTRIC BILL = (CUR. KWH METER - PREV. KWH METER) x KWH RATE
              + KWH METER USE FEE
TOTAL BILL    = GAS BILL + ELECTRIC BILL + MISC. CHARGES
              + SPACE RENTAL FEE
```

He recorded each of the items in the notebook for bookkeeping purposes, and then made out a statement for each tenant. Finally, he figured the total gas bill, the total electric bill, and the total income for the trailer park.

1. Maintains files of data in a structured fashion.
2. Allows data additions and updates within the files.
3. Permits mathematical operations on any numerical data structure or between numerical data structures.

4. Permits all data structures to be sorted in various ways.
5. Permits printing of data structures as reports, lists or what have you.

After further consideration, I decided that most of the tasks related to the "trailer park monthly billing problem" could be solved using the TI-99/4A with the *PRK* cartridge and a printer.

With my TI-99/4A fired up, *PRK* cartridge installed, and manual in hand, I started toying around, setting up the data structure of the file to use on this problem. I finally settled on the structure shown in Table 1.

FILE STRUCTURE

ITEM	TYPE	WIDTH	DEC	DESCRIPTION OF ITEM
1 SPACE #	CHAR	4	0	The trailer space number
2 RENTER	CHAR	15	0	Name of the trailer space renter
3 LAST GAS	DEC	10	3	The previous gas meter reading
4 CUR. GAS	DEC	10	3	The current gas meter reading
5 GAS RATE	DEC	6	4	The cost of the gas per unit volume
6 LAST KWH	DEC	10	3	The previous electric meter reading
7 CUR. KWH	DEC	10	3	The current electric meter reading
8 RATE/KWH	DEC	6	4	The cost of the electricity per KWH
9 G.M. CHRG	DEC	5	2	The monthly gas meter charge
10 GAS TOTAL	DEC	6	2	The cost of gas used plus the meter charge
11 E.M. CHRG	DEC	6	2	The monthly electric meter charge
12 ELEC.TOTL	DEC	6	2	The cost of KWH used plus the meter charge
13 RENT/MO.	DEC	6	2	The trailer space monthly rental cost
14 MISC.CHRG	DEC	7	2	Any other charge (maybe damage to lot...)
15 MO. TOTAL	DEC	8	2	Grand total of gas, KWH, rent, and misc.

TABLE 1

Once a structure has been defined, you can't go back and change it without redefining the entire file structure. In order to minimize this problem, the best policy to follow is to try out the file structure with a small amount of test data. It is a real pain to spend 4 hours entering real data into a file and then discover that one oddball piece of data is too big! By the way, the smaller you define the width of a data item, the more data items you can keep in memory. As you can see, some care must be given to the design of the file structure.

Look at Table 2. It shows my three sample file "pages" of test data. This is the way the data would look after putting in the initial values. Now look at Table 3. The current utility meter readings and any miscellaneous charges have now been entered as the trailer park operator would do once a month.

At this point, I realized I had to figure out how to use the *PRK* cartridge's *math transformations*. That sounds pretty ominous, doesn't it? But study of the manual revealed that it is nothing more than a set of simple equation *templates*. These are shown on page 25 of the *PRK* manual and included here in Table 4. By substituting an item name for the appropriate A, B, or C in the equations, I built up a set of math transformations to figure out the electric, gas, and total bills. The *PRK* cartridge guides you through this process nicely. The tailored set of math transformations is shown in Table 5 (in the order of execution).

Notice that the tailored math transformations set up the next month's LAST GAS, CUR. GAS, LAST KWH, CUR. KWH item fields after the current data was used. This means that next month the user won't have to worry about moving the old "current" values to the "last" fields for the next month too. (That ought to get your imagination working!)

Now for the big test: Run the tailored math transformations on the file of test data and see if it works. The results are shown in Table 6. It is interesting to compare Table 6 to Table 3. The comparison better illustrates the work of these tailored equation templates.

With all the real data in the file, it takes about half an hour to a full hour to process all the math. Sure, that is slow, but it is *accurate*—and the manager can be eating dinner while the *PRK* cartridge processes the data. After dinner, he can start the *PRK* cartridge printing out a report for each *file page*, as shown in Table 6. Finally, after a nice relaxed dessert or brandy, he can cut apart the pages of the report and tape them in the appropriate spot of the form shown in Figure 1. There is a separate form page for each space in the trailer park. By using tape only at the *top* of the little *PRK* page, he can flip through previous month's data (since the little pages are overlapping).

An Automatic *Manual* Feature

By using the ANALYZE PAGES mode of the *PRK* cartridge, you can read the total gas, electric, and monthly income. After selecting the mode, select 5 SEE ITEM

FILE: RENTALS
DATE: 6/20/81
TITLE: TABLE 2

PAGE #	1	PAGE #	2	PAGE #	3
1.SPACE #	A-23	1.SPACE #	B-44	1.SPACE #	B-45
2.RENTER	SMITH,C.W.	2.RENTER	JONES,SAM	2.RENTER	HEIM,WILLIAM
3.LAST GAS	799992.465	3.LAST GAS	830.592	3.LAST GAS	990498.328
4.CUR. GAS	0.000	4.CUR. GAS	0.000	4.CUR. GAS	0.000
5.GAS RATE	.1130	5.GAS RATE	.1130	5.GAS RATE	.1130
6.LAST KWH	128176.263	6.LAST KWH	18841.212	6.LAST KWH	130392.249
7.CUR. KWH	0.000	7.CUR. KWH	0.000	7.CUR. KWH	0.000
8.RATE/KWH	.0231	8.RATE/KWH	.0231	8.RATE/KWH	.0231
9.G.M. CHRG	2.50	9.G.M. CHRG	2.50	9.G.M. CHRG	2.50
10.GAS TOTAL	0.00	10.GAS TOTAL	0.00	10.GAS TOTAL	0.00
11.E.M. CHRG	5.00	11.E.M. CHRG	5.00	11.E.M. CHRG	5.00
12.ELEC.TOTL	0.00	12.ELEC.TOTL	0.00	12.ELEC.TOTL	0.00
13.RENT/MO.	98.00	13.RENT/MO.	105.00	13.RENT/MO.	105.00
14.MISC.CHRG	0.00	14.MISC.CHRG	0.00	14.MISC.CHRG	0.00
15.MO. TOTAL	0.00	15.MO. TOTAL	0.00	15.MO. TOTAL	0.00

TABLE 2

FILE: RENTALS
DATE: 6/20/81
TITLE: TABLE 3

PAGE #	1	PAGE #	2	PAGE #	3
1.SPACE #	A-23	1.SPACE #	B-44	1.SPACE #	B-45
2.RENTER	SMITH,C.W.	2.RENTER	JONES,SAM	2.RENTER	HEIM,WILLIAM
3.LAST GAS	799992.465	3.LAST GAS	830.592	3.LAST GAS	990498.328
4.CUR. GAS	800124.732	4.CUR. GAS	891.947	4.CUR. GAS	990674.998
5.GAS RATE	.1130	5.GAS RATE	.1130	5.GAS RATE	.1130
6.LAST KWH	128176.263	6.LAST KWH	19841.212	6.LAST KWH	130392.249
7.CUR. KWH	131002.097	7.CUR. KWH	23622.609	7.CUR. KWH	134305.045
8.RATE/KWH	.0231	8.RATE/KWH	.0231	8.RATE/KWH	.0231
9.G.M. CHRG	2.50	9.G.M. CHRG	2.50	9.G.M. CHRG	2.50
10.GAS TOTAL	0.00	10.GAS TOTAL	0.00	10.GAS TOTAL	0.00
11.E.M. CHRG	5.00	11.E.M. CHRG	5.00	11.E.M. CHRG	5.00
12.ELEC.TOTL	0.00	12.ELEC.TOTL	0.00	12.ELEC.TOTL	0.00
13.RENT/MO.	98.00	13.RENT/MO.	105.00	13.RENT/MO.	105.00
14.MISC.CHRG	0.00	14.MISC.CHRG	17.50	14.MISC.CHRG	0.00
15.MO. TOTAL	0.00	15.MO. TOTAL	0.00	15.MO. TOTAL	0.00

TABLE 3

ITEM TRANSFORMATIONS

1. A = B
2. A = B + C
3. A = B - C
4. A = B × C
5. A = B/C
6. A = B^C
7. A = ABS(B)
8. A = LOG10(B)
9. A = LOGE(B)
10. A = EXP(B)
11. A = ATAN(B)
12. A = TAN(B)
13. A = SIN(B)
14. A = COS(B)
15. A = INT(B)
16. A = SGN(B)
17. A = PI
18. A = RND

(See the *User's Reference Guide* for a discussion of these functions.)

TABLE 4

TAILORED MATH TRANSFORMATIONS FOR TRAILER PARK BILLING

LAST GAS = CUR. GAS - LAST GAS

GAS TOTAL = LAST GAS X GAS RATE

GAS TOTAL = G.M. CHRG + GAS TOTAL

LAST GAS = CUR. GAS

CUR. GAS = 0.000

LAST KWH = CUR. KWH - LAST KWH

ELEC.TOTL = LAST KWH X RATE/KWH

ELEC.TOTL = E.M. CHRG + ELEC.TOTL

LAST KWH = CUR. KWH

CUR. KWH = 0.000

MO. TOTAL = GAS TOTAL + ELEC.TOTL

MO. TOTAL = MO. TOTAL + RENT/MO.

MO. TOTAL = MO. TOTAL + MISC.CHRG

TABLE 5

FILE: RENTALS
DATE: 6/20/81
TITLE: TABLE 6

PAGE #	1	PAGE #	2	PAGE #	3
1.SPACE #	A-23	1.SPACE #	B-44	1.SPACE #	B-45
2.RENTER	SMITH,C.W.	2.RENTER	JONES,SAM	2.RENTER	HEIM,WILLIAM
3.LAST GAS	800124.732	3.LAST GAS	891.947	3.LAST GAS	990674.998
4.CUR. GAS	0.000	4.CUR. GAS	0.000	4.CUR. GAS	0.000
5.GAS RATE	.1130	5.GAS RATE	.1130	5.GAS RATE	.1130
6.LAST KWH	131002.097	6.LAST KWH	23622.609	6.LAST KWH	134305.045
7.CUR. KWH	0.000	7.CUR. KWH	0.000	7.CUR. KWH	0.000
8.RATE/KWH	.0231	8.RATE/KWH	.0231	8.RATE/KWH	.0231
9.G.M. CHRG	2.50	9.G.M. CHRG	2.50	9.G.M. CHRG	2.50
10.GAS TOTAL	90.42	10.GAS TOTAL	100.50	10.GAS TOTAL	112.35
11.E.M. CHRG	5.00	11.E.M. CHRG	5.00	11.E.M. CHRG	5.00
12.ELEC.TOTL	3025.30	12.ELEC.TOTL	545.38	12.ELEC.TOTL	3101.50
13.RENT/MO.	98.00	13.RENT/MO.	105.00	13.RENT/MO.	105.00
14.MISC.CHRG	0.00	14.MISC.CHRG	17.50	14.MISC.CHRG	0.00
15.MO. TOTAL	185.72	15.MO. TOTAL	247.38	15.MO. TOTAL	222.85

TABLE 6

STATISTICS. Then choose an item—such as GAS TOTAL—and a display like Figure 2 will appear. The gas total for the entire trailer park is contained in the value of SUM. See what reading the manual reveals to you.

Before getting out of the *PRK* cartridge, you must save the data file on cassette tape or floppy disk for next time. Yes, the math transformations are also saved automatically at the same time.

Well, that's the story. I guess the only thing to add is that the *Personal Record Keeping* Command Cartridge isn't the solution to *all* problems. But if you study it and experiment enough, you will be ready to wield this valuable and flexible tool when the appropriate situation arises. So go ahead—give the cartridge a try. I'll bet that soon you too will be witnessing a "Command" performance.

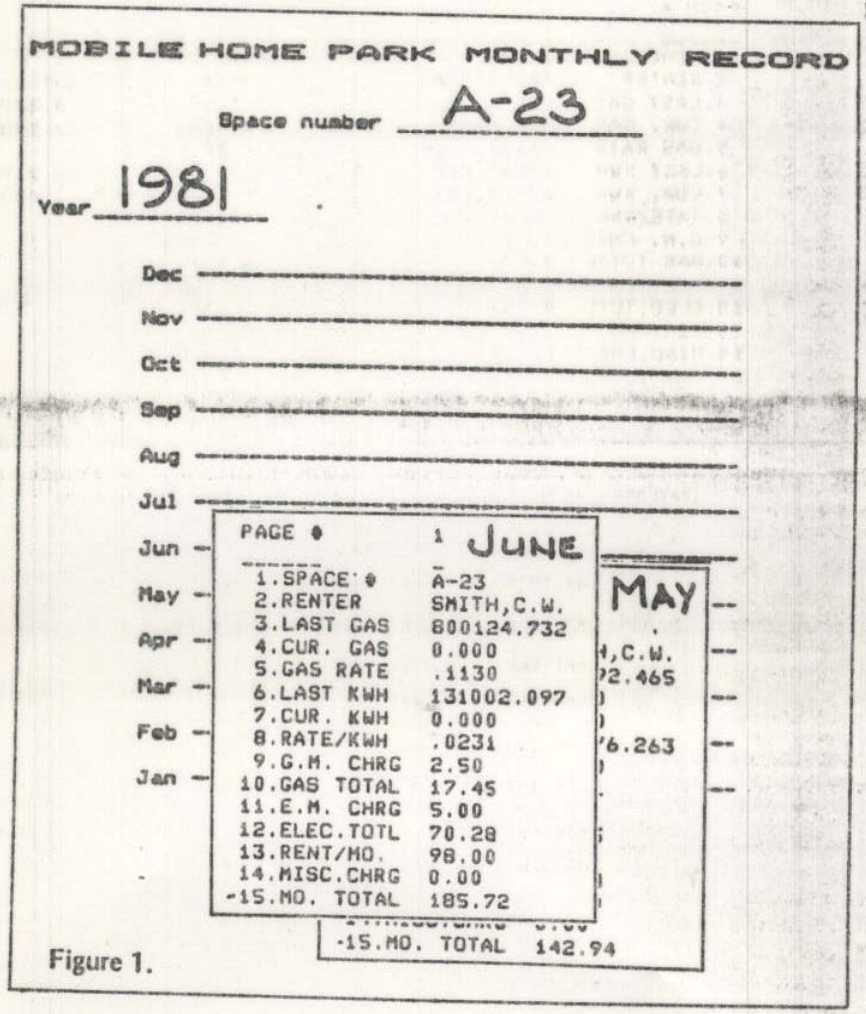


Figure 1.

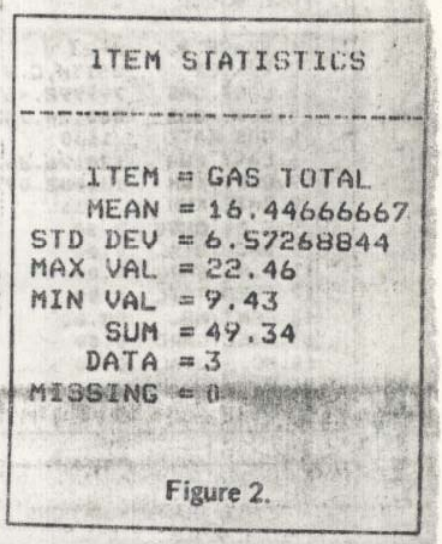


Figure 2.

The Small Investor & the TI-99/4A

A LOOK AT THE DOW JONES NEWS SERVICE

Information utilities such as The Source and MicroNet allow any individual with a microcomputer and modem to tap into a rich vein of information resources. These databases, however, are aimed almost exclusively toward the general consumer population and as such cannot adequately cover the needs of serious, small investors. That's where the Dow Jones News Service (DJNS) comes in: The combination of the DJNS and the TI-99/4A may be the most significant advance in investment analysis since the electronic calculator made its debut. . .

In addition to giving you historical stock quotes, DJNS gives you current-day quotes for all listed stocks, bonds, options and U.S. Treasury issues. The DJNS also has some specialized databases which you can access for information about particular companies, market sectors or market indicators.

For a comprehensive review of a stock or industry, the Media General database provides detailed technical and fundamental indicators on the item of your choice.

The conservative investor can access the Disclosure Online database for a profile on most major companies, plus a 10-K report that lists almost all the important (to the investor) information that can be found in a corporation's financial statement.

The Money Market Service database is a new service introduced by Dow Jones in February 1981. Commentary, tables and graphs on the economy are displayed for most of the important indicators used in determining the current business climate. Of course, the ever-popular Dow Jones averages are also available, as are Trading Activi-

minute. After 7:00, this rate is reduced drastically! Until the next morning, news can be accessed for 20 cents per minute, and historical market quotes for 15 cents. The start-up fee for the service is $50, but there are no monthly charges or minimum on-line times. For high-volume users there is pricing option A. Under this option, there is a $75 monthly fee in exchange for lower prime-time rates during the business day. Pricing option B should be satisfactory for most individual investors.

[To access the Dow Jones New Sevice and its databases you will need the TI *Terminal Emulator II* Command Cartridge to send and receive the appropriate signals, as well as the TI RS232 Interface and an RS232C-compatible telephone coupler (or modem).—Ed.]

After news has been obtained on the News Service, there are really only two things that can be done with it: (1) it can be kept temporarily, or (2) kept permanently. News that is to be kept temporarily is best stored on a disk or printed copy for ease of access and readability. When keeping news permanently, cassette tapes can be both cost effective and reasonably efficient, especially if bought in volume.

For aspects of the service other than news, there are many different ways to use both the historical and current quote databases. The historical quotes are available in either monthly or quarterly format for any given item. While a weekly format would be desirable, the monthly quotes can be used to determine most long- and intermediate-term trends. For the very short-term, one month of daily quotes is always available. These can be used to develop a 10-, 15- or 20-day moving average of prices for the item being researched, and if saved over a period of time, can be used in any format.

For the novice investor, the Media General database provides a sufficient amount of both technical and fun-

past behavior as revealed in graphs can be used to predict future price movements.

The serious investor may prefer to develop his or her own analytical tools. One current theory on Wall Street

today maintains that about half of a stock's performance is due to movement of the market in general, and about half of the movement is due to characteristics peculiar to that particular stock. Naturally, anyone who can predict the movement of the market, even for a short time, has a very powerful financial tool.

For this reason, my own predilection is for analyzing the leading market indices. This analysis can be facilitated by the TI *Personal Record Keeping* Command Cartridge (*PRK*). Each page you set up with the *PRK* can represent one day, and the first few lines can label the index to be tracked. The remaining lines can be the 10-, 15-, or 20-day averages of the aforementioned indices. The use of math transformations in the *PRK* cartridge allows you to compute the average for each of the indices, but you must enter the average manually with the Change Page option. The average has a useful by-product which the *PRK* computes automatically: the standard deviation. I have found this statistic to be a good indicator of market volatility. It too can be entered and tracked with the average. The ability of the *Statistics* Command Cartridge to analyze data produced with the *PRK* cartridge is a definite plus. Even though the *Statistics* cartridge is a more sophisticated analytical device, and offers more tools to work with than the *PRK* cartridge, I do not feel that it is essential to index analysis—only helpful.

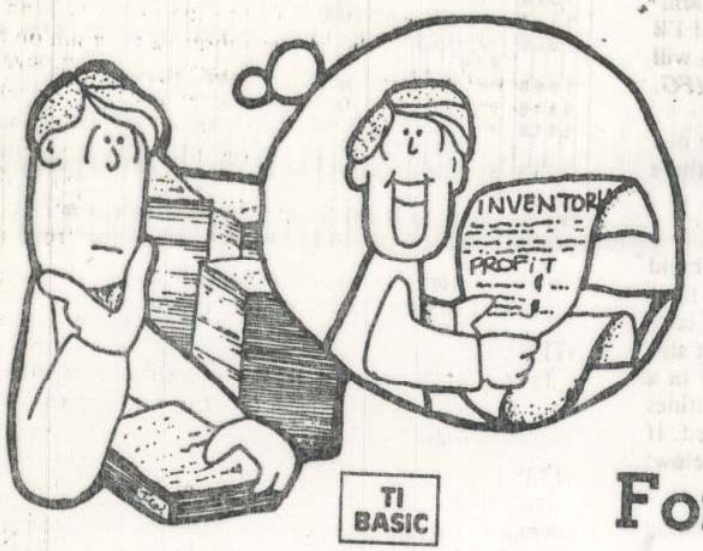
Investors with access to a TI-59 programmable calculator as well as a TI-99/4A can perform some rather astounding mathematical computations without a strong math background. Quotes obtained through the News Service can be processed in a *Least Squares Curve Fit* program detailed in a Texas Instruments publication, *Sourcebook for Programmable Calculators*. This will result in a series of simultaneous equations which can be solved with either the *Master Library-2* program on the TI-59 or the *Math Library-2* program on the TI-99/4A. In theory, the resulting equation should be a reasonably accurate description of the line from which the datapoints were taken, and it can be used to predict the future behavior of the line. Naturally, the number and quality of the datapoints chosen determine the accuracy of the predictive equation, and any conclusion drawn from such analysis is at best highly speculative.

Fundamental analysis using the TI-99/4A also has many applications. You can program balance sheet and income statement analyses, and then compare them to an "ideal" or average analysis in order to determine the variances which may reveal the strengths or weaknesses of a particular company or industry. The information for these analyses can be found in the 10-K section of the Disclosure On-Line database of the News Service.

Of course, these are only a few of the applications that are possible with the TI-99/4A and the Dow Jones News Service. In the past, this mathematical analysis of the market and its component stocks was inaccessible or simply incomprehensible to the small investor. But now, with the help of your TI-99/4A, it's both possible and easy to take a sophisticated approach to market analyses.

I would recommend that any investor with a TI-99/4A computer call Dow Jones on their toll-free number (800-257-5114 except N.J.) to request their free information packet detailing prices and services.

Good luck, 99'ers! If this works for you, your only problem may be writing a suitable income tax program!



TI BASIC

Interactive Forms Generator

When I started in business, I decided to utilize my TI-99/4A as much as possible. One of the things I wanted to do with the computer was to generate customized business forms: purchase orders, price lists, invoices, and sales orders.

Right away you may be thinking: "He could buy all those forms ready made. . ." Yes, but that's not challenging or really as much fun. Not only that, but printing up custom forms (ones that bear your company name and address) is not cheap. Around here a minimum order of triplicate invoices costs about $40 for 500. (And I probably wouldn't use all 500 before wanting to modify the form anyway. . .). Furthermore, multiplying that $40 figure by the 12 *different* forms (including price list pages) I presently use gives a starting cost of $480! That is almost enough money to buy an Epson MX-80 printer!

Well, you guessed it: I bought the printer—plus the serial interface, the RS232 cable, and the TI RS232 interface. The whole setup did cost more than the original estimate, but I can write off the added cost as "hobby money" for now. With the right software I could sit down at the TI-99/4A keyboard, activate a program that would prompt me to fill in the blanks of a form that was in memory, and finally print out as many copies as I wanted on the MX-80.

I wrote such a program and I called it the *Interactive Forms Generator*. It is written in a general fashion to work with any correctly formated data file. I then made up a Form Data File for each of my forms. A Form Data File is just a bunch of ASCII text lines stored in a string array. Each text line may be written as a DATA LINE to be printed on the MX-80 or as a COMMAND LINE to direct the *Interactive Forms Generator* program.

How Does It Work?

The *Interactive Forms Generator* (*IFG*) program asks questions of the operator via the TI-99/4A screen. *IFG* accepts inputs from the operator via the keyboard and interprets instructions from the Form Data File's COMMAND LINES. In other words, the *IFG* program works with you to load your Form Data File, fill out the form, and finally print it out on the MX-80.

Let's say I am generating a Sales Order Acknowledgment form to send to a customer. First, I load the *IFG* program for diskette (or casette). Second, I type RUN and hit ENTER. Third, the *IFG* program asks:

```
MAKE A CHOICE--
1. LOAD NEW FORM FILE
2. FILL OUT SAME FORM
3. PRINT COPIES
4. TERMINATE
```

I enter 1 and follow instructions from the *IFG* program to load the Sales Order Acknowledgment Form Data File. Fourth, the program asks:

```
MAKE A CHOICE--
1. LOAD NEW FORM FILE
2. FILL OUT SAME FORM
3. PRINT COPIES
4. TERMINATE
?
```

I enter 2. Fifth, *IFG* will look through the Form Data File for the COMMAND LINES. Interpreting the lines, *IFG* will prompt me via the screen for the information needed to fill out the form's blanks. Also, in interpreting the COMMAND LINES, *IFG* may perform simple math functions on fields of DATA LINES to calculate tax, totals, etc. After all the COMMAND LINES have been used, *IFG* again asks:

```
MAKE A CHOICE--
1. LOAD NEW FORM FILE
2. FILL OUT SAME FORM
3. PRINT COPIES
4. TERMINATE
?
```

This time I enter 3 and the *IFG* program asks:

```
ENTER NUMBER OF COPIES TO PRINT-
```

I enter some number and *IFG* sends only the DATA LINES of the Form Data File to the MX-80, which does the rest! See Figure 2 for a look at the completed form sample.

Boy, isn't that slick. . .just like the big guys—perhaps a little slower, but that's OK until the business grows to the point that speed is important. (By the way, for Christmas I can generate a very long form letter with a year's worth of family news, then use *IFG* to fill out a separate salutation for each relative. So the whole family gets the latest without my getting writer's cramp! I'll bet that with your imagination and creativity you will come up with some other neat applications for *IFG*, too. . .)

OK, OK. You want to know how you can make one of these Form Data Files, don't you? Well then, there are a couple ways:

Building a Form Data File: Method 1

If you have some kind of editor program that will build an ASCII text string array, you are all set. All you have to do is make sure it will output the special ASCII control codes used by the printer to do its tricks. It must also output the Form Data File to cassette or diskette in a compatible format. Listings 1 and 2 for subroutines CASSOUT and DISKOUT illustrate what is needed. If you don't have an editor program, see Method 2, below:

Building a Form Data File: Method 2

This is a real simple—but much more tedious—method of building the Form Data File.

STEP 1.

Sit down with pad of paper and a pencil. Now design each character-string line of the form. Use the CHR$ ( ) function to put in the string special codes that can't be directly entered by a key on the 99/4A keyboard. The codes can be looked up in the MX-80 (or other printer's) manual. The samples shown below are: CHR$(27), ESCape code; CHR$(13), Carriage Return code; CHR$(10), Line Feed code:

```
CHR$(27)&"E"&"THE DOG RAN HOME
 QUICKLY"&CHR$(10)&CHR$(13)
```

STEP 2

Now fire up your 99/4A. Enter the following program lines:

```
100 REM
110 REM     *FILEBUILD* PROGRAM
120 REM
130 OPTION BASE 1
140 DIM A$(70)
150 REM
```

Then enter your character-string lines from paper into the string array via the TI-99/4A keyboard as follows:

```
160 REM  NOW FOR THE CHARACTER STRINGS
     IN THE ARRAY
170 REM
180 A$(1)=CHR$(27)&"E"&"THE DOG RAN HO
    ME QUICKLY"&CHR$(10)&CHR$(13)
190 A$(2)="AFTER DINNER THE DOG BURPED
    AND SCRATCHED HIS EAR."&CHR$(10)&
    CHR$(13)
200 A$(3)="*******"
??? A$(??)="THAT'S ALL FOLKS!
    "&CHR$(10)&CHR$(10)&CHR$(13)
1000 REM
```

Now enter the following lines of program code:

```
1010 REM  MAKE X EQUAL TO THE NUMBER OF
      LINES IN A$
1030 X=??
1040 PRINT "WRITE FILE TO?"
1050 PRINT "        1 - CS1"
1060 PRINT "        2 - DSK"
1070 INPUT CHOICE
1080 IF (CHOICE<1)+(CHOICE>2)=-1 THEN 1
     040
1090 ON CHOICE GOSUB 2000,3000
1100 PRINT "  C O M P L E T E"
1110 PRINT "NOW SAVE ME ON TAPE OR DISK
     ....."
1120 END
1130 REM
1140 REM OUTPUT SUBROUTINES FOLLOW.....
1150 REM
```

Finally, enter the two subroutines CASSOUT and DISKOUT starting at lines 2000 and 3000.

STEP 3.

Type in RUN. You should end up with your own Form Data File on tape or diskette. This can now be used with the *IFG* program,

STEP 4.

Hold it! Don't turn off the TI-99/4A yet! SAVE your *Filebuild* program on tape or diskette too. Chances are you will want to modify that form because of errors or change of design in the future. OK, now you can turn off the computer and hit the sack. (Notice that this kind of work is always done at midnight. . .)

To help clarify the above process, I generated a simple *Filebuild* program (Listing 3). Note that text lines A$(1)–A$(13) are DATA LINES and text lines A$(14)–A$(19) are COMMAND LINES (more on these next). Data File 2 shows the resulting Form Data File (as printed by my editor program). Figure 1 shows the results of running *IFG* using this Form Data File.

```
KOMPUTAR KORNER
P.O. BOX 483
143 SUNSET DRIVE
ELECTRIC CITY, WASHINGTON
99123

SHIP TO:
        MR. CHIP BUSBY
        9900 SEAMORE AVE.      APT. # 102
        ELECTRIC CITY
        WASHINGTON
          99123
```

Figure 1.

Power to the *IFG*!

How do we get the Form Data File to tell the *IFG* what to do? By making up COMMAND LINES. What makes a COMMAND LINE special? It must start with these two characters: !!. What can a COMMAND LINE tell *IFG* to do? It can tell it to output a message to the TI-99/4A display. How? Here's a sample:

```
!!"THIS MESSAGE WILL BE WRITTEN ON THE
 99/4A DISPLAY"
```

Note that anything between quotes will be displayed.

What about telling it to get something from the 99/4A keyboard? OK—whenever *IFG* does this, it stuffs the information obtained into a line of the Form Data File either right-justified or left-justified. To get input from the keyboard and stuff it left-justified, use this FIELD DEFINITION syntax:

!! :28:1:32: (!!: *:line#:first character:last character:*)

To get information right-justified (which is needed for lining up decimals) do the same except add a ">" sign after the first ":". Example:

!! :>28:1:32:

By the way, *IFG* will show you on the display how much space you have to write in and will let you know if you overflow.

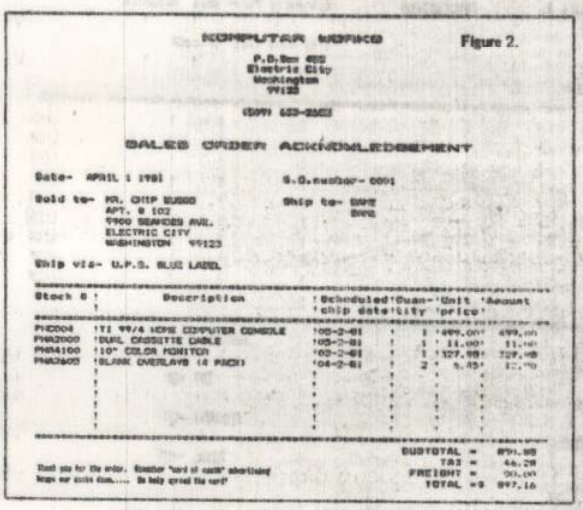
Didn't I say something about *IFG* doing math calculations, you ask? Right. You can write COMMAND LINES to add, subtract, multiply, and divide. Each term and operator simply must be enclosed in parentheses. A term may be a FIELD DEFINITION or a constant. Here are some samples:

!! (:28:1:32:) (*) (.05) (=) (:34:17:24:)
!! (:2:1:4:) (+) (:3:1:4:) (+) (:4:1:4:) (=) (:6:1:4:)
!! (:34:57:68:) (*) (:34:22:32:) (=) (:>34:70:82:)
!! (12.1) (/) (:2:22:32:) (−) (33.3) (=) (:2:22:32:)

I know what you're probably saying right now: "Wow, that is really a lot of power! Is that all *IFG* can do?" Well, there is one more small thing. You can write a COMMAND LINE sequence that will repeat a given number of times. Each time the sequence is repeated, all included FIELD DEFINITION line numbers are incremented. IFG always asks after each repeat cycle if you want to do another. This last feature makes it simple to fill out a form with a multi-line list. Here is a sample repeat sequence:

!!@10; "stock #?" :28:2:10: "description?" :28:12:44:
!!"scheduled ship date?" :28:46:56: "quantity?"
:>28:58:62:
!!"unit price?" :>28:65:70:
!! (:28:58:62:) (*) (:28:65:70:) (=) (:>28:72:79:) @

This sequence will start at Form Data File line 28 and go to line 38. Notice the repeat sequence is bracketed by "@" symbols and the number between the first "@" and the ";" tells how many repeat cycles. Study this sample for a bit and figure out what it does. Then you can look over Data File 1. It is the whole Form Data File produced by my editor program for my Sales Order Acknowledgment form. Figure 2 shows the results of running *IFG* with the Sales Order Acknowledgment Form Data File. The above repeat sequence is from lines 60-62 of that Sales Order Acknowledgement Form Data File.



COMPUTER WORKS

P.O.Box 000
Electric City
Washington
99123

(509) 000-0000

SALES ORDER ACKNOWLEDGEMENT

Date- APRIL 1 1981 S.O.number- 0001

Sold to- MR. CHIP BLOOD
APT. # 102
9900 SEARCES AVE.
ELECTRIC CITY
WASHINGTON 99123

Ship to- SAME
SAME

Ship via- U.P.S. BLUE LABEL

Stock #	Description	Scheduled ship date	Quan-tity	Unit price	Amount
PHC004	TI 99/4 HOME COMPUTER CONSOLE	05-2-81	1	499.00	499.00
PHA2000	DUAL CASSETTE CABLE	05-2-81	1	11.00	11.00
PHM4100	12" COLOR MONITOR	05-2-81	1	327.99	327.99
PHA3605	BLANK OVERLAYS (4 PACK)	04-2-81	2	5.85	11.70

SUBTOTAL = 849.69
TAX = 46.29
FREIGHT = 00.00
TOTAL => 897.16

Thank you for the order. Remember "word of mouth" advertising keeps our costs down..... So help spread the word!

Figure 2.

Finally, Listing 4 is the *Interactive Forms Generator* program. I recommend loading it without all the comment lines to save memory. If you use the disk drive system, you should use CALL FILES(1) and NEW prior to loading *IFG*. That will give space for about a 70-line Form Data File.

Listing 1

```
2000 REM
2010 REM     SUBROUTINE  "CASSOUT"
2020 REM
2030 OPEN #1:"CS1",INTERNAL,OUTPUT,FIXE
     D 192
2040 REM
2050 REM     "X" MUST EQUAL THE NUMBER OF
     TEXT LINES
2060 REM
2070 PRINT #1:X
2080 FOR I=1 TO X+1 STEP 2
2090 PRINT #1:A$(I),A$(I+1)
2100 NEXT I
2110 CLOSE #1
2120 RETURN
```

Listing 2

```
3000 REM
3010 REM     SUBROUTINE  "DISKOUT"
3020 REM
3030 PRINT "ENTER WHICH DISK, 1-3?"
3040 INPUT DISK
3050 IF (DISK<1)+(DISK>3)=-1 THEN 3030
3060 PRINT "ENTER FILENAME:"
3070 INPUT NAME$
3080 IF (LEN(NAME$)<1)+(LEN(NAME$)>10)=
     -1 THEN 3060
3090 OPEN #1:"DSK"&STR$(DISK)&"."&NAME$
     ,OUTPUT,INTERNAL,VARIABLE 132
3100 REM
3110 REM     "X" MUST EQUAL THE NUMBER
     OF TEXT LINES
3120 REM
3130 PRINT #1:X
3140 FOR I=1 TO X
3150 PRINT #1:A$(I)
3160
3170 CLOSE #1
3180 RETURN
```

Data File 1

```
1 ESCFESCHOC4DC2LFCR
2 $1$0ESCEESC$                KOMPUTAR WORKSLFLFCR
3                                   P.O.Box 483LFCR
4                                 Electric CityLFCR
5                                 WashingtonLFCR
6                                   99123LFLFCR
7                               (509) 633-2633LFLFLFLFCR
8 DC2DC4ESCFESCHLFCR
9 SOESCE      SALES ORDER ACKNOWLEDGEMENTLFLFLFCR
10 ESCFSISODate-                              S.O.number-CR
11 DC4DC2                                                        LFCR
12 LFCR
13 SISOSold to-                        Ship to-CR
14 DC4DC2                                                        LFCR
15 HT                                                       LFCR
16 HT                                                       LFCR
17 HT                                                       LFCR
18 HT                                                       LFCR
19 LFSISOShip via-CR
20 DC4DC2                                                  LFLFCR
21 ==========================================================LFCR
22 SISOStock #        Description          Scheduled Quan- Unit  AmountCR
23 DC4DC2       !                             !      !     !        LFCR
24 SISO                                    ship date tity  priceCR
25 DC4DC2       !                             !      !     !        LFCR
26 DC4DC2CR
27 ==========================================================LFCR
28 HT           !                             !      !     !     LFCR
29 HT           !                             !      !     !     LFCR
30 HT           !                             !      !     !     LFCR
31 HT           !                             !      !     !     LFCR
32 HT           !                             !      !     !     LFCR
33 HT           !                             !      !     !     LFCR
34 HT           !                             !      !     !     LFCR
35 HT           !                             !      !     !     LFCR
36 HT           !                             !      !     !     LFCR
37 HT           !                             !      !     !     LFCR
38 ==========================================================LFCR
39 SISO                                          SUBTOTAL =CR
40 DC4DC2                                                        LFCR
41 SISO                                             TAX =CR
42 DC4DC2                                                        LFCR
43 SISO                                          FREIGHT =CR
44 DC4DC2                                                        LFCR
45 SISO                                            TOTAL =CR
46 DC4DC2                                                        LFCR
47 ESC3:HXLCR
48 VTSIThank you for the order.  Remember "word of mouth" advertising LFCR
49 keeps our costs down.....  So help spread the word!LFCR
50 FFCR
51 !!"SALES ORDER ACKNOWLEDGEMENTLFLF" LFCR
52 !!"date?" :11:11:30: "S.O. number?" :11:33:64:CR
53 !!"sold to?" :14:14:37: "sold to address lines=4LF"CR
54 !!"address 01" :15:13:34: "address02" :16:13:36:CR
55 !!"address 03" :17:13:36: "address 04" :18:13:36:CR
56 !!"ship to?" :14:52:76# "ship to address lines=4LF"CR
57 !!"address 01" :15:51:75: "address 02" :16:51:75:CR
58 !!"address 03" :17:51:75: "address 04" :18:51:75:CR
59 !!"ship via?" :20:15:54:CR
60 !!810; "stock #?" :28:2:10: "description?" :28:12:44:CR
61 !!"scheduled ship date?" :28:46:56: "quantity?" :28:58:62:CR
62 !!"unit price?" :28:65:70: (:28:58:62:)(8)(:28:65:70:)(+)(:)28:72:79:) @CR
63 !! (:28:72:79:)(+)(:29:72:79:)(+)(:30:72:79:)(+)(:31:72:79:)(+)CR
64 !!(:32:72:79:)(+)(:33:72:79:)(+)(:34:72:79:)(+)(:35:72:79:)(+)CR
65 !! (:36:72:79:)(+)(:37:72:79:)(+)(:40:73:80:)CR
66 !!"tax rate (ie: .0544)?" :42:73:80: (:40:73:80:)(8)(:42:73:80:)(+)CR
67 !! (:)42:73:80:) "freight charge?" :44:73:80: (:40:73:80:)(+)CR
68 !! (:42:73:80:)(+)(:44:73:80:)(+)(:)46:73:80:)CR
```

Data File 2

```
1 ESCESCBISGKOMPUTAR WORKSLFCR
2 P.O.BOX 483LFCR
3 143 SUNSET DRIVELFCR
4 ELECTRIC CITY, WASHINGTONLFCR
5     99123LFCR
6 LFLFLFLFLFCR
7 SB         SHIP TO:LFCR
8                                                          LFCR
9                                                          LFCR
10                                                         LFCR
11                                                         LFCR
12                                                         LFCR
13 LFLFLFLFLFCR
14 !!"SHIPPING LABELLFLF"CR
15 !!"CUSTOMER NAME":8:35:70:CR
16 !!"STREET ADDRESS":9:35:70:CR
17 !!"CITY":10:35:70:CR
18 !!"STATE":11:35:70:CR
19 !!"ZIP CODE":12:40:60:CR
```

Listing 3

```
110  REM  "FILEBUILD" PROGRAM--VERSION3
     :    SHIPPING LABELS
120  REM
130  OPTION BASE 1
140  DIM A$(70)
150  REM
160  REM  NOW FOR THE CHARACTER STRINGS
       IN THE ARRAY
170  REM
180  A$(1)=CHR$(27)&"E"&CHR$(15)&CHR$(1
     4)&"KOMPUTAR WORKS"&CHR$(10)&CHR$(
     13)
190  A$(2)="P.O. BOX 483"&CHR$(10)&CHR$(
     13)
200  A$(3)="143 SUNSET DRIVE"&CHR$(10)&
     CHR$(13)
210  A$(4)="ELECTRIC CITY, WASHINGTON"&
     CHR$(10)&CHR$(13)
220  A$(5)="      99123"&CHR$(10)&CHR$(1
     3)
230  A$(6)=CHR$(10)&CHR$(10)&CHR$(10)&C
     HR$(10)&CHR$(10)&CHR$(13)
240  A$(7)=CHR$(14)&"          SHIP TO:
     "&CHR$(10)&CHR$(13)
250  A$(8)="
                "&CHR$(10)&CHR$(13)
260  A$(9)="
                "&CHR$(10)&CHR$(13)
270  A$(10)="
                "&CHR$(10)&CHR$(13)
280  A$(11)="
                "&CHR$(10)&CHR$(13)
290  A$(12)="
                "&CHR$(10)&CHR$(13)
300  A$(13)=CHR$(10)&CHR$(10)&CHR$(10)&
     CHR$(10)&CHR$(10)&CHR$(13)
310  A$(14)="11"&CHR$(34)&"SHIPPING LAB
     EL"&CHR$(10)&CHR$(10)&CHR$(34)&CHR
     $(13)
320  A$(15)="11"&CHR$(34)&"CUSTOMER NAM
     E"&CHR$(34)&":8:35:70:"&CHR$(13)
330  A$(16)="11"&CHR$(34)&"STREET ADDRE
     SS"&CHR$(34)&":9:35:70:"&CHR$(13)
340  A$(17)="11"&CHR$(34)&"CITY"&CHR$(3
     4)&":10:35:70:"&CHR$(13)
350  A$(18)="11"&CHR$(34)&"STATE"&CHR$(
     34)&":11:35:70:"&CHR$(13)
360  A$(19)="11"&CHR$(34)&"ZIP CODE"&CH
     R$(34)&":12:40:60:"&CHR$(13)
370  REM
380  REM  MAKE X EQUAL THE NUMBER OF LI
     NES IN A$
390  REM
400  X=19
410  PRINT "WRITE FILE TO?"
420  PRINT "          1 - CS1"
430  PRINT "          2 - DSK"
440  INPUT CHOICE
450  IF (CHOICE<1)+(CHOICE>2)=-1 THEN 4
     10
460  ON CHOICE GOSUB 530,660
470  PRINT "C O M P L E T E"
480  PRINT "NOW SAVE ME ON TAPE OR DISK
     . . . . "
490  END
500  REM
510  REM  OUTPUT SUBROUTINES FOLLOW...
     . .
520  REM
530  REM
540  REM  SUBROUTINE "CASSOUT"
550  REM
560  OPEN #1:"CS1",INTERNAL,OUTPUT,FIXE
     D 192
570  REM
580  REM  "X" MUST EQUAL THE NUMBER OF
     TEXT LINES
```

```
590 REM
600 PRINT #1:X
610 FOR I=1 TO X+1 STEP 2
620 PRINT #1:A$(I),A$(I+1)
630 NEXT I
640 CLOSE #1
650 RETURN
660 REM
670 REM     SUBROUTINE  "DISKOUT"
680 REM
690 PRINT "ENTER WHICH DISK, 1-3?"
700 INPUT DISK
710 IF (DISK<1)+(DISK>3)=-1 THEN 690
720 PRINT "ENTER FILENAME:"
730 INPUT NAMES
740 IF (LEN(NAMES)<1)+(LEN(NAMES)>10)=
    -1 THEN 720
750 OPEN #1:"DSK"&STR$(DISK)&"."&NAMES
    ,OUTPUT,INTERNAL,VARIABLE 132
760 REM
770 REM     "X"  MUST EQUAL THE NUMBER
    OF TEXT LINES
780 REM
790 PRINT #1:X
800 FOR I=1 TO X
810 PRINT #1:A$(I)
820 NEXT I
830 CLOSE #1
840 RETURN
```

This program (Listing 4) scans the file for lines that start with !!. Then these Interactive COMMAND LINES are parsed for four types of commands:

1. *Comments* or messages to prompt the interactive user. This type of command is in the form of text preceded by a quote and followed by a quote.

2. *Field-definition* type commands define the physical field into which the user's keyboard input will be stored. The field has the form—
   :<*line number*>:<*start position*>:
   <*end position*>: Example— :23:5:22
   A Sample COMMAND LINE is:
   !!"Enter the serial number—" :19:7:22:

3. Repeat Command Sequence starts with—
   !!@<Numeric Value>:- and must end with a @.
   Everything in between will be repeated the number of times specified by the numeric value. A sample might be:
   (Line 20) Serial Number— Model—
   (Line 21) Serial Number— Model—
   (Line 22) Serial Number— Model—
   (Line ??) !!"Fill in the table values that follow:"
   (Line?+1) !!@3:"Enter serial number;" :20:15:24
   (Line?+2) !!"Enter the model number:" '20'31'40' @

4. Math Transformations are made up of terms and operators. Terms may be Field-definition or constant types. Operators are "*", "/", "+", "-", and "=". All terms and operators must each be enclosed in parentheses.
   !!(:23:5:22) (*) (.0544) (=) (:>23:17:35:)
   Note the ":>" in the last term which causes the answer to be right justified in the field.

The *IFG* program is set up for use with an EPSON MX-80 printer connected as device:

"RS232.CR.EC.DA=8.BA=9600"

If you are using a different baud rate, the OPEN statements for the printer on line number 1380 must be changed.

You can use a different RS232 printer with the *IFG* program but first check lines 190-360 to make sure these character sequences are compatible with your printer. Especially check RESETEPSON, which initializes the printer.

Listing 4

```
100 REM  •  INTERACTIVE FORMS  •
110 REM  •        GENERATOR     •
130 REM
140 REM
170 OPTION BASE 1
180 DIM A$(70)
190 RESETEPSON$=CHR$(18)&CHR$(20)&CHR$
    (27)&CHR$(70)&CHR$(27)&CHR$(72)&CH
    R$(13)
200 BLANK$="
    "
210 QUOTE$=CHR$(34)
220 BANG$=CHR$(33)&CHR$(33)
230 COLON$=CHR$(58)
240 SEMICOLON$=CHR$(59)
250 RIGHTARROW$=CHR$(62)
260 AMPERSAND$=CHR$(64)
270 OPENPAREN$=CHR$(40)
280 CLOSEPAREN$=CHR$(41)
290 DECIMAL$=CHR$(46)
300 ZERO$=CHR$(48)
310 PLUS$=CHR$(43)
320 MINUS$=CHR$(45)
330 MULTIPLY$=CHR$(42)
340 DIVIDE$=CHR$(47)
350 EQUAL$=CHR$(61)
360 SPACE$=CHR$(32)
370 YES=1
380 NO=0
390 REM  CENTRAL CONTROL MENU
400 CALL CLEAR
410 PRINT "MAKE A CHOICE--"
420 PRINT
430 PRINT "  1 - LOAD NEW FORM FILE"
440 PRINT "  2 - FILL OUT SAME FORM"
450 PRINT "  3 - PRINT COPIES"
460 PRINT "  4 - TERMINATE"
470 PRINT
480 INPUT CHOICE
490 IF (CHOICE>0)+(CHOICE<5)=-2 THEN 5
    20
500 PRINT "ERROR, TRY AGAIN..."
510 GOTO 410
520 ON CHOICE GOSUB 550,840,1340,3020
530 GOTO 400
540 REM  1 - LOAD NEW FORM FILE SUBR
    OUTINE
550 CALL CLEAR
560 PRINT "ENTER DEVICE:"
570 PRINT "                1 FOR CS1"
580 PRINT "                2 FOR DISK"
590 INPUT DEVICE
600 IF (DEVICE<1)+(DEVICE>2)=-1 THEN 5
    60
610 IF DEVICE=2 THEN 690
620 OPEN #4:"CS1",INTERNAL,INPUT ,FIXE
    D 192
630 INPUT #4:X
640 FOR I=1 TO X STEP 2
650 INPUT #4:A$(I),A$(I+1)
660 NEXT I
670 CLOSE #4
680 RETURN
690 REM  -- READ FROM DISK
700 PRINT "ENTER WHICH DISK, 1-3?"
710 INPUT DISKNO
720 IF (DISKNO<1)+(DISKNO>3)=-1 THEN 7
    00
730 PRINT "ENTER FILENAME:"
740 INPUT FILENAME$
750 IF (LEN(FILENAME$)<1)+(LEN(FILENAM
    E$)>10)=-1 THEN 730
760 OPEN #5:"DSK"&STR$(DISKNO)&"."&FIL
    ENAME$,INPUT ,SEQUENTIAL,INTERNAL,
    VARIABLE 132
770 INPUT #5:X
780 FOR I=1 TO X
```

```
790 INPUT #5:A$(I)
800 NEXT I
810 CLOSE #5
820 RETURN
830 REM  2 - FILL OUT SAME FORM
840 REM  MAIN SCANNER- LOOKS FOR COM
    MANDS
850 CALL CLEAR
860 TERM=1
870 FOR I=1 TO X
880 REM  >>>> COMMAND LINE?>>>
890 IF SEG$(A$(I),1,2)=BANG$ THEN 910
    ELSE 1310
900 REM  >>>> IF SO, START OF REPEAT S
    EQUENCE?>>>
910 IF SEG$(A$(I),3,1)=AMPERSAND$ THEN
     920 ELSE 990
920 IF REPEAT=YES THEN 990
930 REM  >>>> INITIALIZE THE REPEAT S
    EQUENCE.>>>
940 REPEAT=YES
950 MAXREPS=VAL(SEG$(A$(I),4,POS(A$(I)
    ,SEMICOLON$,4)-4))
960 REPSTART=I
970 REPS=0
980 REM  LINE PARSER
990 FOR J=3 TO LEN(A$(I))
1000 P$=SEG$(A$(I),J,1)
1010 IF P$=QUOTE$ THEN 1030 ELSE 1060
1020 REM  >>>> GO SUBROUTINE 'COMMENT'
    >>>
1030 GOSUB 1480
1040 J=K
1050 GOTO 1300
1060 IF P$=COLON$ THEN 1080 ELSE 1140
1070 REM  >>>> GO SUBROUTINE 'FIELDINPU
    T' >>>
1080 IF SEG$(A$(I),J+1,1)=RIGHTARROW$ T
    HEN 1090 ELSE 1110
1090 RIGHTJUSTIFY=YES
1100 J=J+1
1110 GOSUB 1650
1120 J=K
1130 GOTO 1300
1140 IF P$=OPENPAREN$ THEN 1160 ELSE 11
    90
1150 REM  >>>> GO SUBROUTINE 'MATH TERM
    ' >>>
1160 GOSUB 1860
1170 J=K
1180 GOTO 1300
1190 IF P$=AMPERSAND$ THEN 1210 ELSE 13
    00
1200 REM  >>>> IS IT THE 1ST AMPERSAND?
     >>>
1210 IF J=3 THEN 1300
1220 REM  >>>> NO, IT IS THE END OF REP
    EAT SEQUENCE MARK.>>>
1230 REPS=REPS+1
1240 IF REPS=MAXREPS THEN 1290
1250 INPUT 'REPEAT ENTRY? (1=YES 0=NO)'
    :MORE
1260 IF MORE=NO THEN 1290
1270 I=REPSTART
1280 GOTO 890
1290 REPEAT=NO
1300 NEXT J
1310 NEXT I
1320 RETURN
1330 REM  3 - PRINT COPIES
1340 REM  FORM PRINT SECTION
1350 CALL CLEAR
1360 PRINT 'ENTER NUMBER OF COPIES'
1370 INPUT 'TO PRINT-':Z
1380 OPEN #3:"RS232.CR.EC.DA=8.BA=9600"
    ,VARIABLE 132
1390 PRINT #3:RESETEPSONS
1400 FOR I=1 TO Z
1410 FOR J=1 TO X
1420 IF SEG$(A$(J),1,2)=BANG$ THEN 1440
1430 PRINT #3:A$(J)
1440 NEXT J
1450 NEXT I
1460 CLOSE #3
1470 RETURN
1480 REM  'COMMENT' SUBROUTINE
1490 COMMENT$=""
1500 FOR K=J+1 TO LEN(A$(I))
1510 P$=SEG$(A$(I),K,1)
1520 IF P$=QUOTE$ THEN 1620
1530 P=ASC(P$)
1540 IF P>96 THEN 1550 ELSE 1570
1550 P=P-32
1560 P$=CHR$(P)
1570 COMMENT$=COMMENT$&P$
1580 NEXT K
1590 PRINT "*** ERROR IN LINE #";I
1600 PRINT "*** MISSING QUOTE..."
1610 GOTO 1640
1620 PRINT COMMENT$
1630 PRINT ""
1640 RETURN
1650 REM  'FIELDINPUT' SUBROUTINE
1660 FRONT$=""
1670 BACK$=""
1680 REM  >>>> DECODE THE FIELD PARAMET
    ERS >>>
1690 GOSUB 2680
1700 PRINT
1710 MIDDLE$=SEG$(A$(LINE),START,LENGTH
    )
1720 PRINT "***"&MIDDLE$&"***"
1730 PRINT
1740 INPUT TEXT$
1750 IF SEG$(TEXT$,1,1)=RIGHTARROW$ THE
    N 1760 ELSE 1780
1760 RIGHTJUSTIFY=YES
1770 TEXT$=SEG$(TEXT$,2,LEN(TEXT$))
1780 IF LEN(TEXT$)>LENGTH THEN 1790 ELS
    E 1830
1790 PRINT "--- TEXT STRING TOO LONG..."
1800 PRINT "PLEASE ENTER SHORTER LINE"
1810 GOTO 1700
1820 REM  >>>> GO STUFF THE FIELD >>>
1830 GOSUB 2860
1840 RETURN
1850 REM  'MATH TERM' SUBROUTINE
1860 J=J+1
1870 ON TERM GOTO 1890,1990,2050
1880 REM  >>>> PROCESS FIRST TERM >>>
1890 IF SEG$(A$(I),J,1)=COLON$ THEN 190
    0 ELSE 1930
1900 GOSUB 2680
1910 FIRSTTERM$=SEG$(A$(LINE),START,LEN
    GTH)
1920 GOTO 1960
1930 ENDFIELD=POS(A$(I),CLOSEPAREN$,J)
1940 FIRSTTERM$=SEG$(A$(I),J,ENDFIELD-J
    )
1950 K=ENDFIELD
1960 TERM=2
1970 GOTO 2660
1980 REM  >>>> PROCESS SECOND TERM >>>
1990 ENDFIELD=POS(A$(I),CLOSEPAREN$,J)
2000 SECONDTERM$=SEG$(A$(I),J,ENDFIELD-
    J)
2010 TERM=3
2020 K=ENDFIELD
2030 GOTO 2660
2040 REM  >>>> PROCESS THIRD TERM AND
    CALCULATE >>>
2050 IF SECONDTERM$=EQUAL$ THEN 2070 EL
    SE 2200
2060 REM  >>>> PROCESS ANSWER AND STORE
     AT 3RD TERM LOCATION >>>
2070 IF SEG$(A$(I),J,1)=COLON$ THEN 208
    0 ELSE 2170
```

```
2080 IF SEG$(A$(I),J+1,1)=RIGHTARROW$ T
     HEN 2090 ELSE 2110
2090 RIGHTJUSTIFY=YES
2100 J=J+1
2110 GOSUB 2680
2120 TEXT$=FIRSTTERM$
2130 GOSUB 2860
2140 K=K+1
2150 TERM=1
2160 GOTO 2660
2170 PRINT "ERROR IN MATH- FILE LINE#";
     I
2180 GOTO 2660
2190 REM >>>> GET THE THIRD TERM .....
     >>>
2200 IF SEG$(A$(I),J,1)=COLON$ THEN 221
     0 ELSE 2240
2210 GOSUB 2680
2220 THIRDTERM$=SEG$(A$(LINE),START,LEN
     GTH)
2230 GOTO 2270
2240 ENDFIELD=POS(A$(I),CLOSEPAREN$,J)
2250 THIRDTERM$=SEG$(A$(I),J,ENDFIELD-J
     )
2260 K=ENDFIELD
2270 TERM=2
2280 IF POS(FIRSTTERM$,DECIMAL$,1)+POS(
     THIRDTERM$,DECIMAL$,1)=0 THEN 2310
2290 ALIGN=YES
2300 GOTO 2330
2310 ALIGN=NO
2320 REM >>>> OK, NOW DO MATH >>>
2330 IF POS(BLANK$,FIRSTTERM$,1)=NO THE
     N 2350
2340 FIRSTTERM$=ZERO$
2350 IF POS(BLANK$,THIRDTERM$,1)=NO THE
     N 2370
2360 THIRDTERM$=ZERO$
2370 FIRSTTERM=VAL(FIRSTTERM$)
2380 THIRDTERM=VAL(THIRDTERM$)
2390 IF SECONDTERM$=PLUS$ THEN 2400 ELS
     E 2420
2400 TEMPTERM=FIRSTTERM+THIRDTERM
2410 GOTO 2530
2420 IF SECONDTERM$=MINUS$ THEN 2430 EL
     SE 2450
2430 TEMPTERM=FIRSTTERM-THIRDTERM
2440 GOTO 2530
2450 IF SECONDTERM$=MULTIPLY$ THEN 2460
       ELSE 2480
2460 TEMPTERM=FIRSTTERM*THIRDTERM
2470 GOTO 2530
2480 IF SECONDTERM$=DIVIDE$ THEN 2490 E
     LSE 2510
2490 TEMPTERM=FIRSTTERM/THIRDTERM
2500 GOTO 2530
2510 PRINT "MATH OPERATOR BAD - FILE LI
     NE #";I
2520 GOTO 2660
```

```
2530 TEMPTERM=INT((TEMPTERM*100))/100
2540 TEMPTERM$=STR$(TEMPTERM)
2550 IF ALIGN=NO THEN 2650
2560 IF POS(TEMPTERM$,DECIMAL$,1)=0 THE
     N 2590
2570 ADJUST=LEN(TEMPTERM$)-POS(TEMPTERM
     $,DECIMAL$,1)+1
2580 ON ADJUST GOTO 2610,2630,2650
2590 FIRSTTERM$=TEMPTERM$&DECIMAL$&ZERO
     $&ZERO$
2600 RETURN
2610 FIRSTTERM$=TEMPTERM$&ZERO$&ZERO$
2620 RETURN
2630 FIRSTTERM$=TEMPTERM$&ZERO$
2640 RETURN
2650 FIRSTTERM$=TEMPTERM$
2660 RETURN
2670 REM      GET FIELD DEF. SUBROUTINE
2680 K=J+1
2690 NEXTCOLON=POS(A$(I),COLON$,K)
2700 LINE=VAL(SEG$(A$(I),K,NEXTCOLON-K)
     )
2710 K=NEXTCOLON+1
2720 NEXTCOLON=POS(A$(I),COLON$,K)
2730 START=VAL(SEG$(A$(I),K,NEXTCOLON-K
     ))
2740 K=NEXTCOLON+1
2750 NEXTCOLON=POS(A$(I),COLON$,K)
2760 LEND=VAL(SEG$(A$(I),K,NEXTCOLON-K)
     )
2770 K=NEXTCOLON+1
2780 IF REPEAT=NO THEN 2800
2790 LINE=LINE+REP$
2800 LENGTH=LEND-START+1
2810 IF LENGTH<1 THEN 2820 ELSE 2840
2820 PRINT "### ERROR IN LINE #";I
2830 PRINT "### FIELD LENGTH NEGATIVE..
     .
2840 RETURN
2850 REM    FIELD STUFFER SUBROUTINE
2860 IF LEN(TEXT$)=0 THEN 3010
2870 IF RIGHTJUSTIFY=NO THEN 2930
2880 FOR M=LEN(TEXT$)TO LENGTH-1
2890 TEXT$=SPACE$&TEXT$
2900 NEXT M
2910 RIGHTJUSTIFY=NO
2920 GOTO 2960
2930 FOR M=LEN(TEXT$)TO LENGTH-1
2940 TEXT$=TEXT$&SPACE$
2950 NEXT M
2960 IF START=1 THEN 2980
2970 FRONT$=SEG$(A$(LINE),1,START-1)
2980 IF LEND=LEN(A$(LINE))THEN 3000
2990 BACK$=SEG$(A$(LINE),LEND+1,LEN(A$(
     LINE)))
3000 A$(LINE)=FRONT$&TEXT$&BACK$
3010 RETURN
3020 END
```

Getting DOWN to Business

Risks and Benefits

You don't need to be reminded that microcomputers are having more than a micro impact on business. If you are reading this, it is because you would like some of that impact to benefit you. In this series of articles, we will explore some of those benefits and show you how to incorporate them in your business or professional work. They will be at least partly cautionary—written to try to keep you out of trouble. And don't expect only *success* stories. After all, *failures* can be most instructive. . .

Planning Use vs. Integrated Use

It is important to distinguish between two major and very different categories of business and professional use of the computer. The first I will call *planning* use. This category includes a lot of activities that are helpful to business and professional people. Applications in this category tend to be analytical or evaluative. They need not be done on a regular basis, but can often be a dramatic help in charting future direction and improving the profitability of a business. Some applications require rather little in the way of input data and are essentially projections; others analyze whatever body of historical data that might be available. Some common examples are the following:

- Comparisons of ROI (Return On Investment) for the various options.
- Interest calculations (e.g., effective interest rates on installment loans).
- Profitability analyses for comparing charges and costs of providing various services.
- Lease vs. purchase analyses.

The second category of use is what I call *integrated* use. This category includes a lot of functions that support a business on a minute-to-minute or day-to-day basis. These are, for example:

- Maintenance of inventory records.
- Preparation of invoices, orders, service contracts, bills, etc.
- Accounts payable and accounts receivable.
- Maintenance of customer or mailing lists.
- Payroll records.
- General ledger and other accounting records.

The potential benefits to your business of applications like these are enormous. But then, so are the risks! Before you allow your business to become dependent on a microcomputer (or any other computer) and set of computer programs, there are a number of steps you must take to safeguard it against the small and large catastrophes that could be (at the least) a major setback for you. This is not to *discourage* you from integrated uses, but rather, to *encourage* you to be very careful about implementing them. [You should also look at "Murphy's Law," which has some steps we recommend you take to protect yourself against this ubiquitous and insidious law: "If anything can go wrong, it will!" It *may* apply (and indeed *has* applied—more frequently than most would care to admit) to integrated computer applications.—Ed.]

A good place for you to start using the power of your microcomputer is in planning applications. They don't require extensive systems of programs or comprehensive detailed business records. They don't need to be done at any given moment, at peril of disaster to your business. And you don't have to chase down some itinerant programmer or software house to update your program upon change of, say, some federal tax formula—again, at peril of disaster. Furthermore, you can implement some planning applications yourself, without extra software, disk drives, extensive data files, or a lot of time.

Projections: A Planning Use

Perhaps you've heard the story of the wealthy Indian maharajah who was challenged to a chess match by a shrewd foreign merchant. The merchant put up one hundred gold coins as his part of the wager, but only asked for rice if the maharajah lost the game: one grain of rice on the first square of the chessboard, two grains on the second square, four on the third, eight on the fourth, and so on. The maharajah was amused and somewhat skeptical that the merchant would ask for only a few grains of rice, but nevertheless accepted the challenge. Naturally—or there would be no point to the story—the maharajah lost. And as the prize was being paid, the full impact became shockingly clear: So much rice did not exist in the world! And even if it did, the immense wealth of the maharajah could have paid for only a tiny fraction of it. . .

You are not often confronted with this type of wager. But you do have opportunities to evaluate, just as the maharajah *should* have. A computer can help you to project events into the future, vary the assumptions and tabulate the projected results. Using a computer program, you can analyze a much more complex situation than you would be willing to do with just a pencil, calculator and paper. You can change your assumptions and let the computer recalculate and reprint the projections, and thus gain much more understanding of the consequences of various contingencies as you play what is essentially a game of "What if. . . ?" As an extra benefit, consider this effect: The necessity of making clear and explicit assumptions usable by the computer may force you to think more clearly and objectively than you might have done otherwise. (I wonder whether baseball clubs would pay as much for some of their benchwarmers and stars if they evaluated the consequences and contingencies objectively.)

A Program Outline

Perhaps the best thing about a projection program is the ease with which you can write it yourself in BASIC. The fundamental tool is a two-dimensional array. If you thought anything connected with arrays was necessarily complex and difficult, please read on. You'll soon discover that an array application can be a lot easier than you imagined.

An array is nothing more than a table in computer storage; a two-dimensional array has rows and columns. We must assign meaning to each, and write our program to honor those meanings. In a projection program, I let each column represent a year (or month?). If the problem requires, I let the numbers in the first column represent initial values, investments, or costs; the numbers in the last column represent residual values, or perhaps totals over all years in the projections. Each row represents a significant quantity that we want to project over the time span.

	1 1981	2 1982	3 1983	4 1984
1 Rents				
2 Vacancy loss				
3 Gross revenue				
4 Property tax				
5 Insurance				
6 Interest				
7 Maintenance				
8 Management				
9 Depreciation				
10 Gross expenses				
11 Net income				

Figure 1. Use of a Table for a Rental Projection

Figure 1 shows a simple projection of a rental operation. There are four columns in the array; they represent 1981, 1982, 1983, and 1984. Each row represents a quantity necessary to the projection of rental results. The array can be declared in BASIC by:

```
60 DIM T(11,4)
```

A BASIC program can refer to any number in the array: For example, to refer to the maintenance expense in 1982 we refer to T(7,2).

1. Set initial (1981) values
2. FOR each additional year compute the projected values
3. FOR each row of the table PRINT a row of the table

Figure 2. Outline of a Projection Program

The outline of the program is shown in Figure 2. Let us examine each of the steps of the outline and show how it would be programmed in BASIC. A bunch of LET statements takes care of the first step. If rental income for 1980 is projected to be $20,000, with a vacancy rate of 5 percent, property tax of $3200, insurance or $700, etc., the first several BASIC statements would be:

```
1010 LET T(1,1)=20000
1020 LET T(2,1)=.05*T(1,1)
1030 LET T(3,1)=T(1,1)-T(2,1)
1040 LET T(4,1)=3200
1050 LET T(5,1)=700
```

These illustrate several ways of assigning values:

- directly as a given number (as in statements 1010, 1040, 1050);
- as a multiple of another number (as in statement 1020)
- as sum or difference of other numbers (as in statement 1030);

It will be clear from your application how to assign each of your values.

```
   10 REM  SKELETON OF A PROJECTION PROG
      RAM
   20 REM  ROWS OF T REPRESENT INCOME OR
        EXPENSE ITEMS
   30 REM  COLUMNS OF T REPRESENT YEARS
   60 DIM T(11,4)
 1000 REM  STEP 1. INITIAL VALUES
 1010 LET T(1,1)=20000
 1020 LET T(2,1)=.05*T(1,1)
 1030 LET T(3,1)=T(1,1)-T(2,1)
 1040 LET T(4,1)=3200
 1050 LET T(5,1)=700
 1990 REM  STEP 2. PROJECT TO FUTURE YEA
      RS
 2000 FOR J=2 TO 4
 2010 LET T(1,J)=T(1,J-1)*1.08
 2020 LET T(2,J)=.05*T(1,J)
 2030 LET T(3,J)=T(1,J)-T(2,J)
 2040 LET T(4,J)=T(4,J-1)*1.06
 2050 LET T(5,J)=T(5,J-1)*1.10
 2200 NEXT J
 2990 REM  STEP 3. PRINT THE RESULTS
 3000 FOR K=1 TO 11
 3010 PRINT K,T(K,1),T(K,2),T(K,3),T(K,4
      )
 3020 NEXT K
 9990 END
```

The second step of the program is probably the most complex. The idea is to march across the table, usually deriving each number from the one to its left—that is, from the corresponding entry for the previous year. However, some of these entries, too, will be multiples, sums, or differences of other numbers in the same columm. We can use the BASIC statement FOR to good advantage here; it easily specifies a repetition for each year. In our rental example, these statements could be:

```
2000 FOR J=2 TO 4
2010 LET T(1,J)=T(1,J-1)*1.08
2020 LET T(2,J)=.05*T(1,J)
```

```
2030 LET T(3,J)=T(1,J)-T(2,J)
2040 LET T(4,J)=T(4,J-1)*1.06
2050 LET T(5,J)=T(5,J-1)*1.10
  :   :
  :   :
2200 NEXT J
```

These statements reflect assumptions that:

- Rental income increases at an 8% inflation rate.
- Vacancy continues at 5%.
- Property taxes increase at only (!) a 6% inflation rate.
- Insurance costs increase at a 10% inflation rate.

If you are not sure how all this works, take out your pencil and, for J with a value of 2, play computer by filling in numbers in the table yourself as the computer would.

Note how all the assumptions are built into the program; each one can be changed at will. You should, in fact, change several, and re-RUN the program several times in order to see the effect of each of your assumptions. This is sometimes call *sensitivity analysis*, but don't let big words scare you.

You may also use the full capabilities of BASIC for special situations. For example, we might project that in the third year, the property will be annexed to the city and taxes will go up 30 percent instead of 6 percent. We could replace statement 2040 by:

```
2040 IF J=3 THEN 2047
2043 LET T(4,J)=T(4,J-1)*1.06
2044 GOTO 2050
2047 LET T(4,J)=T(4,J-1)*1.30
```

Also, suppose that in the same year we expect to have to put on a new roof for $8000; this is a maintenance expense, but one in addition to the regular budgeted maintenance. And unlike the taxes, the extra maintenance does not continue into 1984. We may use another form of the multiplication here:

```
2070 IF J=3 THEN 2077
2073 LET T(7,J)=T(7,1)*1,08 (J-1)
2074 GOTO 2080
2077 LET T(7,J)=T(7,1)*10.8 2+8000
```

In the last step we display the table. The print-out can be prettied up with column headings, a description of each row, and other features. A bare-bones approach is sufficient, really, and could look like this:

```
3000 FOR K=1 TO 11
3010 PRINT K,T(K,1),T(K,2),T(K,3),T(K,4)
3020 NEXT K
```

This segment prints the four numbers of each row of the table on one line, so the table appears on paper just the way we have been thinking about it; each row of numbers is preceded by the row number (K), which at least helps you to identify and keep track of your output.

Listing 1 gathers these program segments into one skeleton. With this as as guideline, you should now be able to sit down and develop your own useful projection programs—applied to sales, production, commissions, or whatever else you need.

Getting DOWN to Business

Evaluating a Software Package

In the first section, I defined two categories of computer applications for business: (1) *planning*—concerned mostly with projections, and not having to be done at particular moments at peril to a business; and (2) *integrated use*—applications such as invoices, accounts payable and receivable, mailing list maintenance, general ledger, inventory, or others upon which a business crucially depends at particular times. In this article, we'll explore some of the implicatons of integrated use.

Programs for integrated use are likely to be rather extensive. After all, most such applications involve organization and management of significant quantities of data. This means that the programs must help you with the data entry, help you monitor the validity and correctness of the data, and help you update the data. The programs must also be able to retrieve data for processing, summarizing, and answering inquiries. Depending on the application, the programs may also have to generate controls for audit purposes, and provide tax reports.

The programs for an integrated use application must be well-designed and form what we would call an *information system*. To develop such a system takes a substantial amount of work probably several *months*, if not *years*, of programmer time. If your application is small enough for you to think about doing it on a TI-99/4A or other micro, it would be quite a mismatch of investment for you to pay for even six months of a programmer's time to develop a system. Therefore, you will want to buy a system that is already developed, packaged, and ready to install and use. You actually have a better chance of getting a good working product by buying a package than by having it done to your specifications by a programmer.

OK, you're in the market for a package. Besides cost, the most obvious criterion is whether a proposed package will meet your needs. Now is the time—even before seeing the details of a proposed package—to make yourself a checklist of the features you want your package to include. List each processing action that you think necessary in your system. Consider the data elements you think would have to be stored and related to each other in order to provide the information you will need at any given mo-

ment. If done in a detailed and comprehensive way, this would be close to what we would call a *systems analysis* of your application.

Great detail and comprehensiveness are not needed; the idea is to give you a starting point for judging the adequacy of a package you may be offered. You will probably find that a particular package is organized differently and operates differently from your outline. There's nothing wrong with that. Concentrate on the *results* produced and whether they are appropriate: Does the proposed package provide the information you consider essential? Then, of course, you can also judge whether the proposed package is convenient or awkward, and flexible or rigid.

A second suggestion is to talk to other users of the proposed package, and get their opinions of the package's strengths and weaknesses. You may be surprised how willing other users are to share their experiences. Even if you have to phone a couple of users long-distance, it will be well worth the trouble and cost.

You should not expect your needs in an information system to always remain the same. Your business changes; auditors make new demands; federal or state regulations change. This is where flexibility of a system comes in. Chances are that there will come a time when you will want your system to do something it was not designed to do. Then, you will need help in modifying the system. The supplier of the package is in the best position to know how to modify your system. But will he be around when you need him? Find out whether the *source* program is supplied and accessible to you. If it is, then you have a chance of getting someone near you to modify it when needed. Try to find out from the supplier and users how much trouble a minor modification would be. You may not be able to trust an answer you get absolutely, because judging how hard it will be to modify a program is difficult, but this is the best suggestion I can make.

In the next section I will review some business-related software. This will provide an opportunity for some more specific suggestions about the analysis of a package.

Now let us turn our attention to something more tangible—a program that should be of practical use to many of you.

Effective Interest Rate or Return On Investment

Suppose you have an opportunity to buy an investment for $1500. The investment is expected to pay $140 at the end of each of the next five years, and at the end of five years return a lump sum of $2000. What is the effective interest rate or total yield on this investment? Or, put another way, what is the return on this investment? This problem can be stated in terms of capital in your business: If you invest some amount in a certain piece of equipment or in a higher level of inventory or. . . ,you expect some estimated improvement in revenues. What is your expected return on this investment?

Since you have many opportunities and a limited amount of capital, you need to compare the expected rates of return on each of several opportunities in order to be able to make the best decision. Of course, there are usually intangible benefits, as well as variations in the risks of different investments. A return-on-investment calculation is, therefore, not the only—or necessarily the deciding—criterion in your decisions. Nevertheless, it will certainly provide valuable input in your decision-making process.

The program presented here is a relatively simple one. I define a component of the investment as one or more payments of equal amounts made at regular intevals. An investment will have two or more components; they are the main input to the program. Each component is described by:

(a) the amount of each payment (there may be only one).
(b) the time at which the first of these payments is made. Time is measured in months from the current moment, which is understood to be time zero.
(c) the number of months between payments. This is irrelevant if there is only one payment in a component, but we require a number anyway.
(d) the number of payments in this component.

For instance, the example above includes three components:

	(a)	(b)	(c)	(d)
1st Component	1500	0	1	1
2nd Component	-140	12	12	5
3rd Component	-2000	60	1	1

Note that the investment amount is given as a positive number, but the returns on the investment are given as negative numbers. The second component represents the five annual payments (12 months apart) starting 12 months after the current time. The first and third components represent single payments: the initial payment and the final payoff after five years (60 months).

The program makes provision for up to ten components; the number of components is the first input the program asks for.

The program strategy is to compute the residual present value at one interest rate higher and one lower than the effective interest rate. We use an interpolation formula to produce a better estimate of the effective interest rate, then narrow the range of possible effective interest rates, and repeat the process. The program stops when the residual value is less than some fraction of the total of the numbers used in computing the residual value, or when the range of possible effective interest rates is less than some tolerance. There are four parameters set in statements 200-230 of the program that you may want to change, depending on your requirements:

U9—starting upper bound for effective interest rate, set now at 30%.
L9—starting lower bound for effective interest rate, set now at 0%.
T9—tolerance for range of effective interest rate, set now at .05%. When the possible range is less than this, we conclude you have the rate closely enough.
P9—tolerance for residual present value, set now at .0001. Because of round-off error during the calculations, this tolerance should not be reduced much below this value.

Figure 1

```
ENTER NUMBER OF PAYMENT COMPONENTS? 3
ENTER AMOUNT OF PAYMENT? 1500
ENTER TIME OF FIRST OF THESE PAYMENTS? 0
ENTER PERIOD BETWEEN THESE PAYMENTS, IN MONTHS? 1
ENTER NUMBER OF THESE PAYMENTS? 1

ENTER AMOUNT OF PAYMENT? -140
ENTER TIME OF FIRST OF THESE PAYMENTS? 12
ENTER PERIOD BETWEEN THESE PAYMENTS, IN MONTHS? 12
ENTER NUMBER OF THESE PAYMENTS? 5

ENTER AMOUNT OF PAYMENT? -2000
ENTER TIME OF FIRST OF THESE PAYMENTS? 60
ENTER PERIOD BETWEEN THESE PAYMENTS, IN MONTHS? 1
ENTER NUMBER OF THESE PAYMENTS? 1

RESIDUAL PRESENT VALUE AT 0% IS -1200
RESIDUAL PRESENT VALUE AT 30% IS 731.7656652
RESIDUAL PRESENT VALUE AT 18.63580073% IS 290.8235145
RESIDUAL PRESENT VALUE AT 15.00040794% IS 93.29345296
RESIDUAL PRESENT VALUE AT 13.91833345% IS 27.69506322
RESIDUAL PRESENT VALUE AT 13.60435554% IS 8.02691232
RESIDUAL PRESENT VALUE AT 13.5139594% IS 2.310160891
RESIDUAL PRESENT VALUE AT 13.48799321% IS .6635205027
RESIDUAL PRESENT VALUE AT 13.48053936% IS .1904640003
EFFECTIVE INTEREST RATE COMPOUNDED MONTHLY,
    IS 13.48053936
```

Figure 1 shows a transcript of the execution of the program with the sample data given above.

Note that the program uses a subroutine starting at line 720; a parameter R is supplied to the subroutine, and parameters V and V3 are returned. If you have Extended BASIC, you can make these parameters explicit in the subroutine call. You can also rephrase some of the control structures using IF-THEN-ELSE and multi-line statements, and make the program much more readable. I leave this for you to explore.

Lease vs. Purchase Analysis

Quite complex programs are available to determine whether leasing or purchasing some piece of equipment is more advantageous. The effective interest rate program can be used for lease vs. purchase analysis, though it requires you to do some side calculation. One method of the analysis would be essentially to calculate the return on *purchasing* the equipment and *leasing* it back to someone else. You would include:

- cost of purchase(+)
- tax benefits from claimed depreciation (−)
- lease payments (−)
- maintenance cost, if maintenance is provided under the lease (+)
- any difference in insurance or other costs between purchasing and leasing (+ or −)
- expected cost of purchase at the end of lease period (−) *or* trade-in value at the end of lease period (−)

The rate of return indicated by this analysis can be compared with your borrowing cost, and the comparison would give you an indication of whether purchase or lease would be more advantageous to you.

As a small example, suppose you are going to get a widget-grinder. You can buy it for $12,000, or lease it for three years at $300 per month. No maintenance is involved, and the insurance cost is the same under lease or purchase. You expect that after three years you would need to trade this one in on a larger model. If you buy it, the trade-in allowance will be $6000. Assuming that either depreciation or lease payments would cost a net of only 60 percent of the actual amounts because of an assumed 40 percent tax rate, the input to the program would therefore be:

	(a)	(b)	(c)	(d)
1st Component	12000	0	1	1
2nd Component	−1200	12	12	3
3rd Component	−180	0	1	36
4th Component	−6000	36	1	1

If you want to check, this example gives an effective interest rate of about 14.1%. Presumably, it would be advantageous to purchase the widget grinder instead of leasing it.

Effective Interest Rate Program: Table of Variables

Arrays:

A1: amount of each payment in an investment component*

T1: time at which the first payment of that component is made (in months, from current time = 0)

F1: number of months between the payments in this component

N1: number of payments in this component

*An investment component is a series of one or more equal payments made at fixed intervals. Payments may be paid out (+) or received (−).

Parameters:

U9: upper limit for effective annual rate

L9: lower limit for effective annual rate

T9: tolerance: when the interval between upper and lower limits (L1, U1) is less than this, the program stops

U9: tolerance—when the residual present value at a trial interest rate, divided by the sum of the absolute values of all components, is less than this, the program stops

C: number of components

I: index of the current investment component under consideration (always goes from 1 to C)

L1: current lower bound on effective rate

U1: current upper bound on effective rate

R: trial interest rate, on which to calculate residual present value V

V: residual present value, based on trial interest rate R

L2: residual present value at lower limit L1

U2: residual present value at upper limit U1

V3: sum of absolute values of component present values

V4: present value of a component at rate R

V5: temporary variable used in computing V4

R1: monthly increase factor, using rate R

PROGRAM OUTLINE:
Effective Interest Rate

Line Nos.	
200-230	Set Parameters.
250-370	Obtain input data from user.
400-560	Set lower and upper limits, and the residual present value at each.
590-700	Iterate: interpolate to get a new trial interest rate R, replace either upper or lower bound by R.
720-920	Subroutine: computes residual present value at the trial rate R; also computes V3.
930-950	Report final result.

```
100 REM  *********************
110 REM  *  EFFECTIVE INTEREST RATE  *
120 REM  *********************
130 REM
140 REM
150 REM
160 REM
170 REM
180 REM
190 DIM A1(10),T1(10),F1(10),N1(10)
200 U9=30.0
210 L9=0
220 T9=0.05
230 P9=1.0E-4
240 REM  ACCEPT INPUT
250 PRINT "ENTER NUMBER OF PAYMENT COMPONENTS";
260 INPUT C
270 FOR I=1 TO C
280 PRINT
290 PRINT "ENTER AMOUNT OF PAYMENT";
300 INPUT A1(I)
310 PRINT "ENTER TIME OF FIRST OF THESE PAYMENTS";
320 INPUT T1(I)
330 PRINT "ENTER PERIOD BETWEEN THESE PAYMENTS, IN MONTHS";
340 INPUT F1(I)
350 PRINT "ENTER NUMBER OF THESE PAYMENTS";
360 INPUT N1(I)
370 NEXT I
380 PRINT
390 REM  SET LOWER & UPPER BOUNDS FOR EFFECTIVE RATE
400 L1=L9
410 U1=U9
420 REM  GET RESIDUAL VALUE AT LOWER BOUND
430 R=L1
440 GOSUB 720
450 L2=V
460 IF ABS(V/V3)<P9 THEN 930
470 REM  GET RESIDUAL VALUE AT UPPER BOUND
480 R=U1
490 GOSUB 720
500 U2=V
510 IF ABS(V/V3)<P9 THEN 930
520 REM  RESIDUAL VALUES MUST HAVE OPPOSITE SIGNS AT THE BOUNDS
530 IF U2*L2<0 THEN 590
540 PRINT "EFFECTIVE RATE NOT BETWEEN ";L9;" AND ";U9
550 PRINT "CHECK YOUR INPUT OR CHANGE BOUNDS L9 AND U9"
560 GOTO 950
570 REM  INTERPOLATE BETWEEN LOWER & UPPER BOUNDS
580 REM  FOR NEW TRIAL RATE R
590 R=(L1*U2-U1*L2)/(U2-L2)
600 GOSUB 720
610 IF ABS(V/V3)<P9 THEN 930
620 REM  TRIAL RATE REPLACES WHICHEVER BOUND HAS RESIDUAL VALUE WITH THE SAME SIGN
630 IF V*L2>0 THEN 670
640 U1=R
650 U2=V
660 GOTO 690
670 L1=R
680 L2=V
690 IF U1-L1<T9 THEN 930
700 GOTO 590
710 REM  SUBROUTINE TO COMPUTE RESIDUAL VALUE V AT RATE R
720 V=0
730 V3=0
740 FOR I=1 TO C
750 IF N1(I)>1 THEN 790
760 REM  COMPUTE RESIDUAL VALUE IF ONLY ONE PAYMENT
770 V4=(1+R/1200)^(-T1(I))*A1(I)
780 GOTO 880
790 IF R<>0 THEN 860
800 REM  SPECIAL CASE WHEN R=0
810 V4=N1(I)*A1(I)
820 GOTO 880
830 REM  COMPUTE RESIDUAL VALUE OF SERIES OF PAYMENTS
840 R1=1+R/1200
850 V5=(1-R1^(-N1(I)*F1(I)))/(1-R1^(-F1(I)))
860 V4=A1(I)*R1^(-T1(I))*V5
870 REM  IN ALL CASES, INCLUDE V4 IN V AND V3
880 V=V+V4
890 V3=V3+ABS(V4)
900 NEXT I
910 PRINT "RESIDUAL PRESENT VALUE AT ";R;"% IS ";V
920 RETURN
930 PRINT
940 PRINT "EFFECTIVE INTEREST RATE, COMPOUNDED MONTHLY, IS ";R
950 END
```

Getting DOWN to Business

Inventory

In this section, let's consider an inventory system for your computer. I don't have a particular system to review, but I want to discuss what should be involved in an inventory system, and why. This has implications for a number of other applications you might like for your business, such as order processing, accounts payable, and even a general ledger system.

We will address the kind of system you might use in a sales organization—either in a store or for mail or phone order. Some of the description also fits the situation of a raw materials inventory or even miscellaneous supplies. Since some of the activities may not fit you if your business is small, be prepared to discount some of the benefits.

First, it is important to know *why* you apply a computer to some task. You should have specific advantages in mind and know what you have to do to attain those advantages. And of course you should be prepared to change your operations as necessary. I have seen organizations that wanted to "put it on the computer" without any clear reason. Often such organizations waste time and resources changing the specifications, design, and operation of a system as they struggle to develop reasons for their system on the fly. Others merely wind up with a system which is a burden to run, with no advantages except an imagined prestige.

On the other hand, I did some work not long ago with a company that was going to get a computer. I expected that payroll would be one of their first applications, as it is in so many companies. But no, they had payroll near the bottom of their list. Because they had only 60 or so employees, they were able to do their payroll manually quite well and had other uses in mind that would give them definite advantages. For them, one of the first priorities was inventory.

What are some of the possible benefits of keeping your inventory records by computer?

1. If you are processing orders by computer, you can improve the efficiency of your warehouse operation in several ways:

   a. The computer can recognize which orders cannot be filled, and thus avoid sending the warehouse crew to look for the items.

   b. The computer can produce "pick slips" or a "picking list" arranged in a sequence to make the picking of the items from the warehouse efficient.

   c. The computer can help manage back orders; when new stock arrives, the computer automatically scans the file of back orders and fills any back orders for the items before allowing new orders a chance.

2. The computer can help you manage your inventory levels effectively and save you money. To do this, you must have good projections of future demand for each item. You can then time your reorders and calculate optimal reorder quantities. At least in theory, you should be able to reduce your working capital tied up in inventory, and at the same time be out of stock less often and therefore be able to fill more orders and keep customers happier.

Next, let's consider the information you must keep in your inventory file in order to have an effective inventory system. This file has a record for each product (and perhaps for each size, color, model and style). There is inventory status information: current quantity on hand, quantity on order from vendors, date expected, quantity on back order to customers, and quantity sold since last update. There is also historical demand information, such as quantity sold in each month in perhaps the last year. Finally, there is reorder and forecast information: i.e., preferred vendor, vendor's product number, vendor lead time, order quantity, order frequency or reorder point, and demand forecast.

If your computer is processing orders, you also maintain files of back orders (if permitted). The order processing programs obviously use and update the inventory file. If your computer is not used for processing orders, you must find some other means of updating your inventory data. One of the troubles with this is that your input data to the inventory system are likely to be much less reliable than the order-processing input would be.

An inventory system must include a number of other functions. There are simple updates to price, cost, and warehouse location, as well as addition and deletion of products. There are also inventory adjustments caused by events such as return of an item from a customer, or the removal of an item for product testing. The function of receiving into inventory is complex: Quantities on hand and on order must be updated. A payable transaction is generated—with its necessary comparison of actual arrival amount with invoiced quanitity—so there is an interface with your accounts payable system, if you are using one. Then your system must be sure to trigger the filling of back orders from the new stock *before* letting any new orders have access to it.

Periodically, you must count your physical inventory and adjust your computer inventory accordingly, since you need your inventory file to reflect reality, not wishful thinking. Many events can cause a discrepancy in inventory counts—things such as pilferage, mislabeling, or failure to make the minor adjustments necessitated by the odd-but-authorized removal or replacement of items. The computer should help the physical inventory process by

printing the stock list, and by making it easy to adjust the inventory for discrepancies found.

And then there are the functions involved in reordering: About once a week your system should sweep through all products and determine what to reorder. You of course have the opportunity to override the computer's suggestions, but any such decisions must be recorded (e.g., in quantity on order). Perhaps once a month your system should update some analysis programs that keep your demand history current and recompute forecasts, reorder quantities, etc.

There is even a connection to your general ledger system. After all, inventory is an asset, and any activity that affects the value of that asset should be reflected in your profit and loss, assets and liabilities.

All this is a great deal of work. Not only are there a lot of things to do, but they must be done accurately. I knew a company that went bankrupt, *primarily* because the order processing and inventory control they did by computer was not accurate. They tremendously overstocked some items because the computer said there were none on hand (and of course didn't fill orders because it thought there was no stock), and ran out of stock on other items because the computer thought there were plenty. Naturally the company couldn't fill those orders either! One of the causes of the snafu was the company's lack of understanding of how the system was supposed to work, how to ensure its accuracy, and how to diagnose inaccuracies.

On the other hand, another company, where I helped install order and inventory processing, listened very carefully to what we told them about operation for accuracy. They were not only willing to tighten some of their operations, but were also eager to be able to control their warehouse functions more closely. That company is still prospering.

There's another side to consider too. If you have a small list of products to keep track of, you probably do rather well keeping track of them already. And as for calculating optimal inventory levels, reorder quantities, etc., you can probably do that rather well with a pocket calculator and formulas you can find in many textbooks. So honestly, *would* the computer help you do a better job of managing inventory than you already do (or could do manually with the same effort you would have to put into a computer system)? If the answer is no, then save yourself money, time, and management energy by *not* doing computer inventory. If there are real benefits you would receive, I hope this section will make you a little more aware of what you must prepare for and strengthen your resolve to do it carefully, and do it accurately. The stakes are too high for you to wander casually into a computer inventory system!

Getting DOWN to Business

When Random Does Not Mean By Chance

Random-access files are extremely important in any conversational application that requires a data base of some kind. This includes any kind of *business* information system, but also includes a lot of others as well. Unfortunately, the concept of what *random access* actually is often gives rise to misunderstanding and even fear—that is, the fear that using random access is too complex to be attempted. In this article I will try to correct some of the misunderstandings and start you on your way to using random-access files.

The dictionary I took to college told me that random meant "going, made, occurring, etc., without definite aim, purpose, or reason." Synonyms given are *haphazard, chance, casual, aimless*. Thus, when I first heard of random access in reference to computer data, it didn't sound like anything I would want. The good people didn't *mean* haphazard or chance, or any of those other things; they meant access *directly* to a piece of data specifically wanted, *without* having to pass sequentially by a lot of other unwanted data to get there. To me, this is much better described by the phrase *direct access*, and I have been using direct access and talking against the term *random access* for years. But enough. The terminology *random-access* appears in my newer dictionary, and is generally understood in computer circles to mean "permitting access to stored data in any order the user desires." From the standpoint of the storage unit, access *is* random in the older sense, since the sequence of access requests is not at all predictable (compared with sequential access, which is entirely predictable). So this is the point—direct (I still like that word) access to whatever data we want, in any sequence.

Why is this important? Suppose you are using an inventory system. You have a transaction for product 539. Your last transaction was for product 762. What must you do to retrieve, update, and rewrite the record for product 539? If your inventory file is an ordinary sequential file, you must start at the beginning of the file, read all the records up to product 539, and rewrite each to a new file. Impossibly slow, yet it gets worse: After you do your thing with product 539, you either have to finish copying the rest of the records to the new files or postpone that, in hope that the next transaction will be for a product after 539 so we can save a trip through the whole file. What we clearly need is the ability to go directly to record 539, read it, and write the updated record back in the same place. Random-access files permit you to do just that, and the savings in time are what make a data-

based system feasible—not only for inventory, but for accounts payable or receivable, general ledger, etc.

Implementation in TI BASIC

In a random-access file, in TI BASIC and in every other system I know, all records must be the same length. The operating system knows the length of each record, knows where the file begins on disk, and therefore can calculate the exact location of the 367th record, or any other record. This calculation is used whenever we ask to read or write a particular record.

Let's look at the statements we use on random-access files. They are the same statements we use on ordinary (sequential) files, but some parameters are different. First, when we OPEN a random-access file, we must declare:

- file organization is RELATIVE
- file type is DISPLAY or INTERNAL
- open mode is INPUT, OUTPUT, or UPDATE
- record type is FIXED

Don't ask why the word RELATIVE is chosen to specify random access, but it may have something to do with the address calculation: The location of each record is computed relative to the beginning of the file. You may well want to construct your random-access files as INTERNAL, to save space and time required for converting DISPLAY (ASCII) files for internal use. An INTERNAL file cannot be listed directly, but you probably need a program to list a random-access file anyway. An *open mode* of UPDATE allows you to read and write records in your file, and this is what you want most of the time. UPDATE is also the default if you don't specify an open mode. As well as specifying FIXED record type, you may also specify the record length, and I recommend that you do. As an example,

```
OPEN #1:"DSK1.INVENTORY",RELATIVE,
 INTERNAL, UPDATE, FIXED 92.
```

opens the INVENTORY file on your DSK1 as your #1 file; the file has 92-byte records in internal format, for random-access reads and writes. When you first create a file, you can and should specify the number of records to be allocated initially; the number follows the word RELATIVE. For example, the program that first established this file could have used:

```
OPEN #2: "DSK1.INVENTORY",RELATIVE
 150,INTERNAL,OUTPUT,FIXED 92
```

To read a particular record, include the record number (the first record is numbered zero) in the INPUT statement; if N = 119, for example,

```
INPUT #2,REC N: PN,D$,Q,PR
```

reads the 119th record from the file into the variables PN, D$, Q, PR. The PRINT statement similarly includes the word REC and the record number of the record to be written:

```
PRINT #2,REC N: PN,D$,Q,PR
```

You can use the EOF function for a random-access file, but this is not the best way. Better is to use the record 0 to hold special information about the file, especially the length of the file. As soon as you open the file, read that record:

```
INPUT #2, REC 0: FL
```

Then, before accessing any record, compare its record number with FL:

```
230 IF N>FL THEN 260
240 INPUT #2,REC N: PN,D$,Q,PR
250 GOTO 280
260 PRINT "INVALID RECORD NUMBER.
    REENTER"
```

When we wish, we are allowed to read or write records sequentially in random-access file. And of course we should CLOSE a file at the end of the program.

Which Record Contains What?

Okay, so you can easily get the 119th record in your random-access file. But how do you know that the information you want is in the 119th record? That is the hard part. If you are willing to assign product numbers 1 to 200 to the 200 items in your inventory file, you have no problem. At least, not until you discontinue some products and add others. In many cases, you can't assign the key to your file (product number, social security number, account number, or whatever) like this at all. So we need some scheme that associates a record number with each of your keys.

There are a lot of ways to do this. I will show one here: an index, which I keep in a file of its own. Actually, it could be kept in the first several records of your random-access file if you wish. Let's suppose an inventory system with up to 200 products. The product numbers are already assigned, as integers like 17, 29, 83, 104, 105, etc. We can keep our index in a pair of arrays in main storage while we run our system: these arrays don't take a lot of room.

```
 60 DIM IPN(200),ILOC(200)
 70 OPEN #1: "DSK1.INVINDEX",SEQUENTIAL
    INTERNAL
    :
    :
180 FOR I=1 to 200
190 INPUT #1: IPN(I),ILOC(I)
200 NEXT I
```

The IPN array holds the product numbers, and the ILOC array the record numbers in the random-access file for the corresponding products. When we want to access a product, we search the IPN array, find the record number, then use it to access the product record directly.

Using this scheme, the sequence of records in the random-access file matters very little. The sequence in the index file (and therefore in the arrays) matters more. The easiest thing, but least efficient, is to search the IPN array sequentially, with the product numbers in either ascending sequence or no particular sequence. One better idea is to put the most frequently used records at the front of the index file, thus cutting down on the average number of index entries your program must search. Studies have shown that in situations like this, 80 percent of the desired accesses are to 20 percent of the items. A still more efficient (but longer to program) method is a binary search, requiring that the index be in ascending sequence by product number. But let's come back to that idea another time.

Putting It all Together

Let's see how some of this works. We will see, at least in outline, how to (1) update a particular record, using the index, and (2) how to add a new record to the file (and of course to the index). First, let's be a little more precise about how we keep information, again using an inventory system as the context.

1. The RELATIVE file is named INVENTORY; its first record (numbered 0) contains the allocated length of the file; the number of records actually used must not exceed that number. If the allocated length is 201 records, for example, we might at some time be using 160, and these would be numbered 1 to 160.

2. The index file is named INVINDEX; it contains an index entry for each of the allocated records in INVENTORY. The index entries are in sequence by product number. The unused records are identified in the index by a product number like 32767, which is larger than any actual product number. In addition at the very beginning of the INVINDEX file are:

(a) the number of allocated records
(b) the number of currently active records

As part of our program initialization, we must open the files and read the index into our arrays:

```
 60 DIM IPN(200),ILOC(200)
 70 OPEN #1: "DSK1.INVINDEX",SEQUENTIAL,
    INTERNAL
 80 OPEN #2: "DSK1.INVENTORY",RELATIVE,
    INTERNAL,UPDATE,FIXED 92
 90 INPUT #1: NALLOC,NACTV
100 FOR I=1 TO NALLOC
110 INPUT #1: IPN(I),ILOC(I)
120 NEXT I
```

Now, suppose the program has accepted a product number APN, and needs to retrieve the INVENTORY record for that product; we will show for simplicity a sequential, rather than a binary search through the index file:

```
310 FOR J=1 TO NACTV
320 IF APN=IPN(J) THEN 370
330 IF ANP>IPN(J) THEN 350
340 NEXT J
350 PRINT "PRODUCT NOT ON FILE"
360 GOTO . .
370 INPUT #2,REC ILOC(J): PN,D$,Q,PR,. . .
```

If the program goes on to update some fields of the record, the record can be rewritten with its updated contents very simply:

```
470 PRINT #2,REC ILOC(J): PN,D$,Q,PR,. . .
```

Inserting a new record for a new product number is a little tricky. Where to put it in the inventory file is no problem; it can go right after the last active record. The index will make it accessible at the right time, with no problem. But we have more to do with the index. We must insert a new index entry in its proper place in sequence. Let's look at that process. Suppose that the product number to be inserted is PN, and that we have ascertained that such a number is not in the file.

```
600 IN NACTV<NALLOC THEN 630
610 PRINT "NO MORE SPACE IN THE
    INVENTORY FILE"
620 GOTO . .
630 NACTV=NACTV+1
640 PRINT #2,REC NACTV: PN,D$,Q,PR
650 REM ADJUST INDEX
660 FOR J=1 TO NACTV
670 IF PN>IPN(J) THEN 690
680 NEXT J
690 FOR K=NACTV TO J+1 STEP -1
700 IPN(K)=IPN(K-1)
710 ILOC(K)=ILOC(K-1)
720 NEXT K
730 IPN(J)=PN
740 ILOC(J)=NACTV
750 REM REWRITE THE UPDATED INDEX FILE
760 RESTORE #1
770 PRINT #1: NALLOC,NACTV
780 FOR I=1 TO NALLOC
790 PRINT #1: IPN(I),ILOC(I)
800 NEXT I
```

None of these operations takes very long. We always have the index file, the index arrays, and the random-access file itself in sync.

Do you have a better scheme? You may very well have, especially for your particular application. There is a lot of room for different ways of using and managing random-access files. After all, what we have is really the capability of managing large arrays—kept on disk instead of main storage. I hope you can see the importance of, and get some idea of how to use, random-access files from this introduction.

Getting DOWN to Business

DIVIDE AND CONQUER

In an earlier section we discussed random access files, and explored some details of using them. This section will review a few of the main points, develop a further idea or two, and then show a full example program using random access files.

A random access file is essentially a big array stored on disk. We can treat it much as we would treat an array; we access an element of an array by using a subscript, whose job is to select one particular element of the array:

```
350 K = A(SUBS)
360 B(SUBS) = L
```

These are ordinary BASIC statements that retrieve a value from the A array and store one in the B array. Similarly, we can specify exactly which record we want to read from or write to a random access file:

```
440 SUBS = 37
450 INPUT #1,REC SUBS: PN,D$,Q,R
460 PRINT #2,REC SUBS: L$,AVE,RET
```

These statements read the 37th record from the #1 file and write a record into the 37th record position of the #2 file. So SUBS is used just like a subscript to select which record to read or write.

The Index to the Random-Access File

In random-access files, the problem is knowing which record should be stored (or found) in which location. In my last section, I described a small inventory system in which the key to each record was the product number, an integer. There are at most 200 product numbers, but they are *not* just the numbers 1 to 200. My storage scheme stores product records arbitrarily, in the random-access file, but includes an index file also. The index file contains a pair of numbers for each record: the product number and the record number (the subscript in the random-access file where the record for the product number is stored). For example, Figure 3-a shows the index and the file after four entries have been made to the file. The records were inserted for products 67, 105, 29, and 84, and the records are stored in the random-access file in the sequence in which the records were created. The function of the index is to keep a list of the product numbers and the position of each record in the random-access file.

Early in any program that uses the random-access file, I read the index file into a pair of arrays, IPN and ILOC. When I then want to access a product record, I can search the IPN array at electronic speeds for the desired product record, and go directly to the product record.

Binary Search

In my last column, I showed a sequential search of this table, and hinted that a binary search would be faster. Let's take a look at a binary search: It is not very complex, is really a time saver, and can be applied in many table-search situations.

First, let's be clear about the context. We have an array, whose elements may be numbers or strings. Let's use numbers in our example, but strings can work exactly the same way. The elements in the table must be arranged in ascending sequence. We also have a number, called the *search key*, that we want to find in the table. So the objective in the search is to find the subscript for which the table element matches the search key. If no match for the search key exists in the table, we say the search fails.

The idea is a divide-and-conquer strategy. At all times, we keep track of the lower bound and the upper bound of the possible subscript in the table. At the start of the search, these bounds are the beginning and end of the table, of course. The central idea is that each time through the search loop, we compare the search key with the table element *half-way between the upper and lower bounds*. If that element matches the search key, the search is finished successfully. Otherwise, if the search key is less than this middle element, the desired table element must be in the lower half of the currently remaining portion of the table. So, we bring the upper bound down to the entry just below the middle one. The final case is the one in which the search key is greater than then middle element. So, the desired element must be in the upper half, and we bring the *lower* bound up to the one just above the middle. We repeat this process; each time through, we reduce the remaining possible portion of the table by half. The search ends either when we have found our table element or when we find the lower bound is greater than the upper bound; this last condition shows that the search has failed.

Let's look at an example. Figure 1 shows a table of twelve numbers (product numbers?) in ascending sequence. Suppose we are searching for 135 in the table. Our search starts with a lower bound of 1 and an upper bound of 12. Our first iteration computes a middle of (1 + 12)/2 = 6 (rounded down). The 6th table entry is 119; the search key is larger so the lower bound is set to 7, and we have eliminated the entire lower half of the table from further consideration. In our second iteration (7 + 12)/2 = 9, and we compare the search key with the 9th entry, 158. This time, the search key is less, so the *upper* bound becomes 8. The third iteration tests the 7th element itself and sets the lower limit to 8. The fourth

Figure 1 A table of numbers in ascending sequence.

```
17
29
63
104
105
119
130
135
158
183
197
262
```

iteration finds that the 8th element matches the search key. Suppose our search key were instead 139; the iterations would be exactly the same, except that in the fourth iteration, lower and upper bounds are both 8, and we find that the search key is larger than the 8th table element. Thus the lower bound exceeds the upper bound, and the search fails.

Figure 2 A subroutine for a binary search.

```
1000 LOWB = 1
1010 REM 1000. SUBROUTINE TO BINARY SEARCH THE TABLE IPN.
1020 REM OF LENGTH NACTV, FOR SEARCH KEY APN.
1030 REM RETURNS ISUB = SUBSCRIPT OF MATCHING ENTRY,
1040 REM OR 0 IF THE SEARCH FAILS
1050 UPB = NACTV
1060 IF UPB < LOWB THEN 1140
1070 ISUB = INT ((LOWB + UPB)/2)
1080 IF APN = IPN (ISUB) THEN 1150
1090 IF APN < IPN (ISUB) THEN 1120
1100 LOWB = ISUB + 1
1110 GO TO 1060
1120 UPB = ISUB - 1
1130 GO TO 1060
1140 ISUB = 0
1150 RETURN
```

Figure 2 shows a subroutine that searches a table named IPN for a search key named APN. The length of the table in NACTV. The subroutine returns the value of the subscript ISUB of the matching element in the table, or returns ISUB = 0 if the search fails. Trace the subroutine, using the table shown in Fig. 1, to verify that the subroutine follows the logic described above.

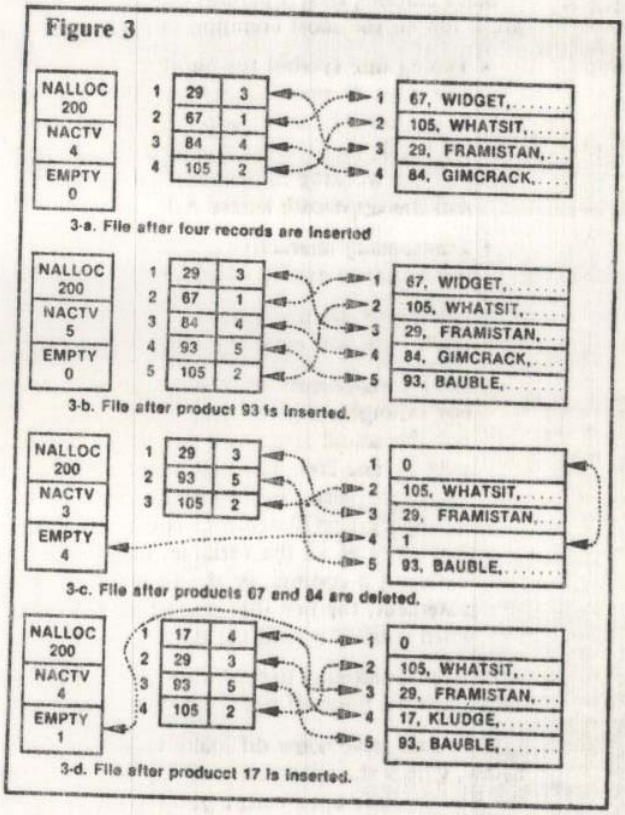
Inserting and Deleting Records

If we are only inserting records into a random-access file, there isn't much problem. We use the next record position in the random-access file, and adjust the index by moving the entries up to make room, in order to put the new product number where it should go in sequence. We showed this process in the earlier section. The problem is more complex if we also delete entries from the file from time to time. And, of course, we want to be able to reuse vacated file positions.

To keep track of these positions, we use what the computer scientists call a *linked list*. We keep a variable, called EMPTY, which gives the first available file position; if there are no deleted record positions currently available (they may all have been used up again, or maybe no record have been deleted at all), EMPTY equals zero. Suppose the record stored in the 11th position of the random-access file has been deleted. Then EMPTY = 11. And the 11th record *now* contains the *next available* file position, which might be 37. Then the 37th record contains the next available file position, etc. The last available file record contains a zero to mark the end of the list.

When deleting a record, we store in that record position the current value of EMPTY, and record the file position of this newly deleted record as the new value of EMPTY. When we need to insert a new record in the file, we look at EMPTY first, to see whether there are any previously deleted records whose positions we can recycle. If so, we use the first one in the list, but first read from it a new value to place in EMPTY, so the second available file position is now first. If EMPTY = 0, we are using all file position up to NACTV. In that case we just use the next file position after NACTV, unless NACTV = NALLOC, in which case we've run out of space.

Figure 3



You can see the process in Fig 3. First, Fig. 3-c shows the result of deleting first product 67 and then product 84 from the file. The list of empty file positions starts at position 4 (as EMPTY tells us), and then continues to position 1. The zero, stored at position 1, indicates that there are no more empty records. The index table records only the currently active products, and NACTV tells how many there are.

When product 17 is inserted (Fig. 3-d), it is put into the first available empty position, which is position 4. The list of empty positions is reduced, and the index, of course, is expanded. If two more products are inserted, the first will go into file position 1, but then the EMPTY list will itself be empty, so the next product will be put in position 6.

Appendix

Your Guide to Keying In Programs from *The Best of 99'er*

To save yourself both time and effort, *always* make sure you have all the proper hardware and software to RUN any program. The *No Disk* symbol is used to designate programs which (as listed) will completely fill available memory and cannot be RUN with the Disk Controller (and possibly the RS232 Interface) attached.

Some of the programs in *The Best of 99'er, Volume 1*, are short (about twenty lines or less). These brief sections of code are not typeset in our grid format. Lengthy programs with long program lines, however, have been mechanically typeset on a grid background for readability and proper spacing. Even with such clear listings, it is possible for you to make typing mistakes. You should, therefore, carefully read the section on editing program lines in your Texas Instruments *User's Reference Guide* (pp.II-38 and II-39) before beginning to key in programs.

Since entering long programs can lead to errors even for the most careful keypuncher, here are a few of the most common ones to watch for:

- **Typing one symbol for another.**
  The most common transformations are: substituting the letter O for the number 0, the letter I for the letter J, the lowercase letter L for the number 1, the letter S for the $, and the uppercase B for the number 8. This error is especially likely if you are working in hexadecimal numbers which are composed of the numbers 0-9 and the uppercase letters A-F.

- **Transposing characters.**
  For example, typing in 000154000 instead of 000145000.

- **Adding or deleting spaces.**
  Make sure you enter the same number of spaces as you find in the listing.

- **DATA statements are often sources of hidden errors.**
  For example, if your computer gives you an error message such as "BAD VALUE IN 260" the actual error may not be on line 260, but buried in a DATA statement *used by* line 260. The best way to handle this kind of error is to type in DATA statements very carefully and to verify them, by checking each character before typing the next statement. This is much simpler in the long run than printing out the values of all the variables in a given DATA statement. If you inadvertently leave out a comma, or if you leave out a value or a set of values in a DATA statement, the line that reads that DATA statement will give you an error message when it tries to read the data.

Before you attempt to run your typed-in program, first check it for the kinds of errors listed above to save yourself time and frustration.

If you do have some difficulty typing in the listings, refer to the Key-In Reference strip below. Check the appearance of any character you are in doubt about with the character on the Key-In Reference before you type it in.

To prevent the loss of a program while you are typing it in, it is a good idea to periodically save portions of it. If you are using a cassette storage system, you may want to refer to *A Beginner's Guide to Cassette Operation with A Home Computer*(on page 20 of this book) which includes many helpful hints as well as a detailed description of an effective method of saving programs on cassette tape. Basically, for each 25-50 lines you type in, use SAVE CS1 to save that program segment onto one of two tapes. Alternate between the two tapes each time you save the program. Be sure to rewind to the beginning of each tape before saving, so that you always record over and replace the shorter segment of program lines with the progressively longer segment. By following this procedure, you'll always retain most of your work even if the lights go out, or someone turns off the computer before you are finished.

Index

to
The Best of 99'er
Volume 1

When using this Index, note that boldface page numbers indicate charts, tables, listings or main sections (e.g.,**129ff**). Italics are used to designate article titles. In accordance with standard indexing procedures, "**ff**" has been used to represent subject discussions of more than 3 pages.

"After the initial excitement has wo[illegible] off . . .
8 out of 10 machines are used less than once a month.
. . . and 1 year later, they are virtually forgotten!"
—From a recent home computer industry survey.

AN URGENT MESSAGE FOR TI USERS

The reason behind the surprising statistic is quite simple. Once the excitement of purchase has worn off, home computer owners often find themselves overwhelmed—left with a complex machine and little practical knowledge or ongoing support.

TI-99/4A owners, especially, may feel left out of the continuing excitement represented by innovations in home computer systems and software. Fortunately, there is a remedy: a monthly magazine devoted to the interests of the TI user.

Home Computer Magazine (HCM) provides you with the latest news, the latest product information, and presents the most current programs and tips for getting the m[illegible] out of your TI Home Computer system.

While books and manuals provide basic, necessary information, only a timely, ongoing publication can bring continuing excitement and discovery into your home month after month.

In every jam-packed issue of *Home Computer Magazine* you'll find lively, informative articles with dynamic, full-color illustrations—articles dealing with education, business, and entertainment, as well as professional and home applications. Each issue contains extensive tutorials and fun-filled learning exercises for beginners, tips and "how-to" articles for intermediate-level users, and advanced programming techniques that keep the pros coming back for more—as well as page after page of ready-to-run software for everyone.

To complement its traditional, in-depth coverage of the Texas Instruments Home Computer, *HCM* has now added coverage of the Apple, Commodore, and IBM home systems—opening up an even wider world for TI owners in 3 important ways:

(1) More software—You'll get first-time-ever versions of Apple IIe, Commodore 64 and IBM PCjr programs translated to run on your TI-99/4A.

(2) More products—Third-party producers for the Apple, IBM, and Commodore machines will now present their TI-compatible products exclusively in the pages of *HCM*.

(3) More information—Our coverage of these larger systems will increase your knowledge and prepare you to make intelligent choices if you should purchase a second system at a later date—making all of your back issues that much more valuable for reference use.

From the People Who Know The Home Computer Best™

HOME COMPUTER magazine™

Formerly 99'er Home Computer Magazine™

You've certainly made the right choices so far. First, you decided it was time to bring a computer into your home. Next, you purchased an excellent machine. Then you selected this book—the most comprehensive information resource for that machine. Now, you have the chance to put it all together by subscribing to a magazine that will ensure you get the most value from your purchases—keeping you an *active* computer user. Don't be one of the 8 out of 10 "drop-outs" cited above. Act NOW and subscribe TODAY so you don't miss out on a single issue. There will never be a better time.

SATISFACTION GUARANTEED

Or the Unfilled Portion of Your Subscription Will Be Refunded

☐ **YES—Please sign me up as a subscriber. Enclosed is payment or credit card billing information.**

Term	U.S.A.	Canada & Mexico	Foreign Surface	Foreign Air
1-yr (12 issues)	☐ $25	☐ $32	☐ $43	Inquire
2-yr (24 issues)	☐ $45	☐ $52		
3-yr (36 issues)	☐ $63	☐ $70	Sample Issue	☐ $3.95

☐ Check or Money Order enclosed
(MUST BE IN *U.S. FUNDS* DRAWN ON *U.S. BANK*)

Name _____ **PLEASE**

Address _____ **PRINT**

City _____ **State** _____ **Zip** _____

Bill my: ☐ **VISA** ☐ **MasterCard**

Expiration Date _____

Signature _____

Charge Card Number:

Please Mail Your Order To: ▶ **Subscription Department (B99)**
Home Computer Magazine
P. O. Box 5537
Eugene, OR 97405

MasterCard or VISA phone orders only call:

TOLL-FREE 1-800-828-2212 WEST COAST *TIME*

Subscription Ordering Service M – F 7:00 AM – 5:00 PM, Sat 8:00 AM – 4:00 PM

In Oregon, Alaska, Hawaii Tel. (503) 485-8796