No. 1920
$22.95

AT&T 6300
A COMPREHENSIVE USERS MANUAL

FREDERICK HOLTZ

TAB TAB BOOKS Inc.
Blue Ridge Summit, PA 17214

To my good friend Topsy Breeden.

FIRST EDITION

FIRST PRINTING

Library of Congress Cataloging in Publication Data

Holtz, Frederick.
AT&T 6300—A Comprehensive
User's Manual

Includes index.
1. AT&T PC 6300 (Computer)—Programming.
2. Computer programs. I. Title.
QA76.8.A76H65 1985 001.64'2 84-26754
ISBN 0-8306-0920-2
ISBN 0-8306-1920-8 (pbk.)

Front cover photograph courtesy of AT&T.

Contents

Acknowledgment

I would like to acknowledge and thank AT&T for supplying the PC-6300 and for providing a wealth of technical information about their product.

List of Programs

Introduction

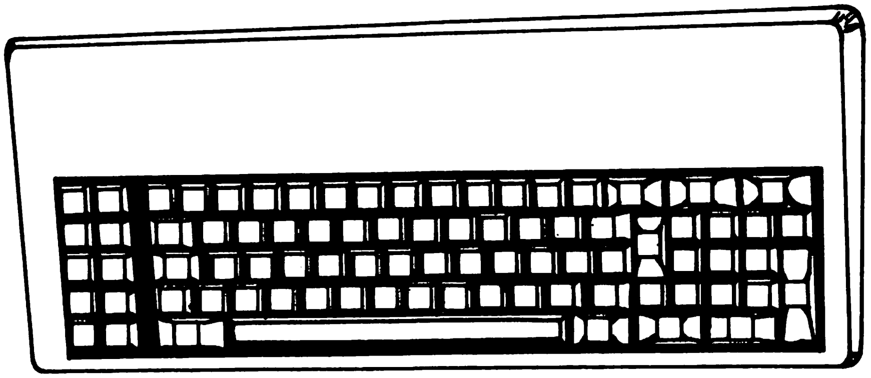
In June of 1984, another giant corporation entered the microcomputer field. AT&T announced their new PC-6300 microcomputer, which had been rumored for about a year. The AT&T PC-6300 is a fine machine that is already making its mark through enhanced operating speed. It is the culmination of many of the finest microcomputers that have gone before it.

This book is entended for the person who already has or is contemplating purchasing the AT&T PC-6300. It is written on a fairly nontechnical level, although there is a description of the machine and its general attributes. Beginners will especially appreciate the several chapters devoted to programming in GWBASIC, the standard language of the AT&T PC-6300. All readers should enjoy the wealth of programs that are presented in other chapters. In these pages, you will find programs that address filekeeping, color graphics, games, text mode, and even sound effects and music programs. All of these programs are ready to run on the PC-6300, and many can even be combined to arrive at highly specialized programs.

All programs are educational in that they explain more and more about the capabilities of the PC-6300. If you're just getting started in the personal computing field, or if you've been at it a while and want to move up to the PC-6300, this book will explain the computer in fully understandable language.

The PC-6300 is here to stay. I hope this book will provide you with the opportunity to get started on this excellent machine and progress as far as your imagination will allow.

Chapter 1



The AT&T Personal Computer

On June 26, 1984, AT&T announced several new data processing products for large and small businesses. Stuck between several "super-microcomputers" was the AT&T PC-6300, whose existence had been rumored for more than a year. Until shortly before this announcement, very few people knew exactly what the new personal computer from AT&T would be like. Probably taking a lead from IBM, its competitor, AT&T leaked rumors about the AT&T PC to the microcomputer buying public a full year before its announcement. Such leaks generally lead to several millions dollars worth of word-of-mouth advertising. This was certainly the case with the PC-6300.

Nearly everyone assumed that a company with AT&T's background would certainly come out with a very high-level product and give the front runner, IBM, a run for its money. Many rumors circulated. Some addressed a new type of operating system that would run on the AT&T PC. Others averred that the AT&T PC would be the first microcomputer to be offered with UNIX as its standard operating system. Some even ventured to guess that the AT&T PC might not even be programmed in BASIC, but would come with a C-language interpreter.

These speculations addressing UNIX and C are only natural, because AT&T (through Bell Labs) is the owner of the UNIX operating system and the developer of the C-language. As is usually the case, however, most of the rumors surrounding the introduction of the AT&T PC were wrong. The AT&T PC-6300 is here, and it is an excellent machine, one that is sure to make its mark in the microcomputer industry.

Charles Marshall, chairman of AT&T Information Systems, announced the new line of computers from AT&T. He said, "We're here today to take another step forward in fulfilling our commitment to provide integrated communications-based office automation systems for American business."

"The products we are announcing today are part of a logical evolution from our own communications-based Information Systems Architecture," explained Frank Vigilante, division president. He also explained that the products establish the foundation of the automated office. "Each works with the other. Because they can grow and function with newer, larger systems, tomorrow these systems will still be performing and protecting our customers' investment."

Turning to the AT&T Personal Computer, he explained that it runs the same MS-DOS operating system and business software as other leading personal computers and accepts the same plug-in accessory circuit boards. It offers greater speed, more features, and a higher level of standard equipment than its competition, at a competitive price. It has been certified "operationally compatible" by Future Computing, of Richardson, Texas. Mead Data Central, of Dayton, Ohio, has also certified that its LEXIS and NEXIS database software is compatible with the AT&T PC.

"We've brought the 16-bit computer up to date and added communications capabilities to enhance its performance and synergy with our 3B line," Vigilante said. "And unlike its leading competition, it provides high-resolution graphics and supports multiple printer outputs in a basic configuration."

The AT&T PC is available in two models, one with two half-height 360K floppy disk drives and 128K of RAM, or a second with one floppy drive, a 10-megabyte hard disk and 256K of RAM. Both models can be upgraded to 640K of RAM.

The AT&T PC can operate in an integrated computing environment with AT&T UNIX system-based 3B computers through the AT&T PC Interface, which was announced in March. This allows the AT&T PC to act as one of up to 18 workstations in a network with the more powerful 32-bit 3B super-minicomputer.

Marshall explained that AT&T will utilize all distribution channels that are appropriate to the marketplace, including direct sales to large customers, complementary channels such as OEM/VAR markets, and selected computer specialty retailers for the AT&T PC.

As it turns out, the AT&T PC-6300 is not the exotic offering everyone thought it would be. By *exotic* I mean completely different from all that have gone before. For instance, when Apple came out with their Lisa (and later, the Macintosh), these could be classified as exotic offerings because they used brand-new operating systems and operated in a manner that was significantly different from most computers that had gone before.

The AT&T PC-6300 has taken the best of what the microcomputer world has to offer today and used this as the base from which to build. Instead of the UNIX operating system, the PC-6300 uses MS-DOS, an operating system that has been with us since the introduction of the IBM PC in the early 1980's. The standard language of the PC-6300 is Microsoft BASIC, which is identical to Advanced BASIC on the IBM PC. AT&T's version is named GWBASIC, which stands for "Gee Whiz BASIC." BASICA by any other name is still BASICA, and this dialect of BASIC can be classified as today's microcomputer industry standard.

AT&T chose to go the route of standard 5 1/4-inch diskette drives with a 10-megabyte hard disk drive option. The point here is that everything about the AT&T PC fits well with industry support of previous computers. For the owners, this means compatibility with existing products on the market.

While IBM is the major competitor of AT&T in the microcomputer industry, AT&T makes no bones about the fact that compatibility of their AT&T PC-6300 with the IBM PC was a primary goal. In general, you will find that any software that runs on the IBM PC will also run on the AT&T PC. Additionally, most option boards designed for the IBM PC will work with the AT&T PC. The exception to this are boards that fall into the memory expansion category.

It was stated earlier that AT&T has taken the best the market had to offer and used that as the base from which to build. The AT&T PC is an improvement over that which was offered prior to its announcement. The microprocessor, which is the heart of any microcomputer, is different from the microprocessor used in the IBM PC.

The AT&T PC uses the Intel 8086 microprocessor, which is a full 16-bit chip—as opposed to an 8/16-bit unit like the Intel 8088 used in the IBM PC. To avoid an in depth technical discussion, I'll just say that the 8086 microprocessor in the AT&T PC operates at nearly twice the speed of the one used in the IBM PC.

For this reason, the AT&T PC should process information approximately two times faster than the IBM PC. It is this speed increase which renders most memory expansion boards designed for the IBM PC incompatible with the AT&T PC. Other than this exception, most expansion and software options available for the IBM machine are compatible with the AT&T PC. The buyer of the AT&T PC-6300, then, may select from literally thousands of products that have been on the market for years. This includes both software and hardware.

Generally, an exotic new computer is limited in the initial stages by the lack of compatible products from other manufacturers and suppliers. This is not true of the PC-6300 because of the compatibility that has been built in by AT&T. In fact, the AT&T PC-6300 is no more than an Olivetti Personal Computer with a few cosmetic changes. The PC-6300 is made by Olivetti for AT&T. Anyone can tell this by examining the circuit boards, each of which is stamped with the Olivetti name.

COMPONENTS OF THE AT&T PC-6300

The AT&T PC-6300 consists of three discrete components that are linked together to form the microcomputer system. These are the keyboard, (Fig. 1-1), the system unit (Fig. 1-2), and the adjustable monitor (Fig. 1-3). Standard memory configuration is 128K for the floppy disk unit and 256K for the unit with the hard disk option. The latter also contains its own hard disk controller card. The floppy disk controller card is resident on the motherboard, which is mounted at the bottom of the system unit. The 128K unit may be expanded to 256K by adding memory chips to the motherboard.

The AT&T PC-6300 contains seven expansion slots, two of which may be used with 16-bit plug-in boards. The display controller is located at the far left of the system unit and does not occupy one of the seven slots. This is technically a color display adapter, although the standard monitor supplied

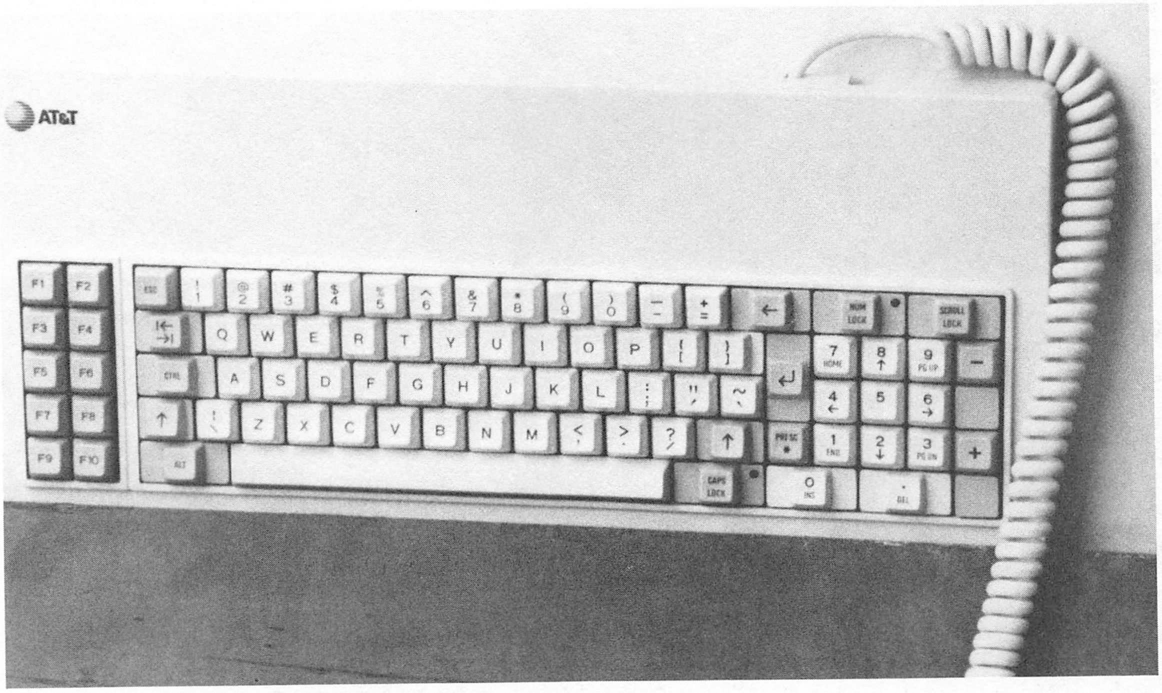


Fig. 1-1. The PC-6300 keyboard.

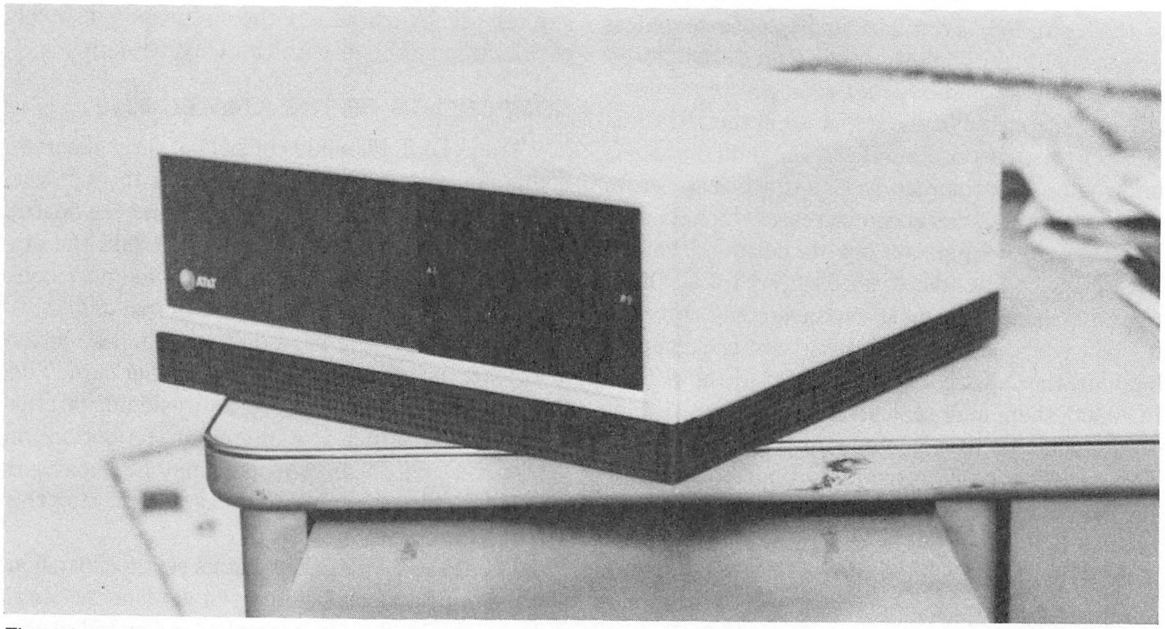


Fig. 1-2. The PC-6300 system unit.

Fig. 1-3. The AT&T Personal Computer monochrome monitor is adjustable for ease of viewing.

with the AT&T PC is monochrome. To obtain color capability, it is necessary to purchase the optional AT&T RGB color monitor, which connects to the same receptacle as the monochrome display. This is a real advantage over the IBM PC, which requires one card for its monochrome monitor and another for a color monitor. Those units equipped with a monochrome monitor still have full graphics capability, although the color commands will simply yield a display output in different shades of gray. There are several display modes designed for text, medium resolution, high resolution, and super-high resolution operations. The latter mode offers twice the display resolution of most standard microcomputers today. With the standard display board, the AT&T PC-6300 has 16-color capability. All 16 colors may be displayed on the text screen, while only 4 colors may be displayed at any one time on the medium-resolution graphics screen. The remaining two graphics modes are monochrome only.

The floppy disk drives format disks that offer 360K byte storage. These drives use double-sided, double-density, 5 1/4-inch floppies. The hard disk drive unit offers 10M byte storage capability, and is bootable.

There are many nice features offered by the AT&T PC. These are not monumental offerings, but little niceties that add a lot to operator convenience. Among these is a reset switch on the front panel, which allows you to activate a system reboot without switching the computer off and then back on again. The monitor is another nicety. Instead of being fixed in place, you can swivel it left and right and up and down for personal viewing comfort. Also, the system unit is cooled by a very large fan mounted at the rear of the chassis. The large blades allow a larger volume of air to be moved at a lower fan speed. Therefore, the unit is very quiet and runs at only a lukewarm temperature.

Probably the nicest feature of the AT&T from the stand point of convenience is overall system size. No, the AT&T PC-6300 is not a portable when you define a portable as a self-contained, single-unit computer that can be carted around like a briefcase or small piece of luggage. By the same token, it's not a tremendous hulk either. The system unit is compact, and the screen and keyboard are lightweight. This means that it's a fairly simple task to move the AT&T PC from one location to another. The system unit is about three-quarters the size of most IBM PC compatibles, although its weight is about the same. In testing this machine, it was necessary for me to move it several times to locations separated by over 100 miles. In every instance, takedown and setup required less than 5 minutes, and the system travels well on the back seat of a subcompact automobile. Many other component, desk-top computers do not offer this same ease of portability.

For those potential buyers who already own MS-DOS compatible machines, I think you will find the transition to the PC-6300 automatic. As a matter of fact, it takes less than an hour in most instances to feel right at home with the PC-6300. Keyboard layout is identical to that of the IBM PC, and the key action is about the same. The keyboard may be situated in one of three positions, as there are adjustable feet at the rear of the keyboard that can assume three different heights.

If you're buying a microcomputer for the first time, the AT&T PC-6300 is an ideal choice. This is true because myriad expansion products and software are already available for it through its MS-DOS compatibility. Also, documentation such as this book and many others can be applied directly to the PC-6300. Most books that explain programming in Microsoft BASIC can be used just as if they were written for the owner of an AT&T PC.

For the person who already owns an MS-DOS compatible computer, the AT&T PC is the natural way to move up, especially if you're searching for faster execution speed. Chances are, all of your previous programming applications will run perfectly on the PC-6300, except at a speed that may be two or more times faster than that of machines that use the Intel 8088 microprocessor.

Overall workmanship seems to be better than average, and AT&T has already announced options that are or will soon be available to enhance communications with other microcomputers and even mainframes. Also, there is a new display board in the works that will allow for 16 simultaneous colors in the various graphics modes. In checks I have made, it would seem that the outside suppliers of products for microcomputers—such as modems, display boards, memory boards, etc.—are gearing up to include the AT&T PC-6300. It would seem that the "fast machines" and the fast memory which they require are becoming standard in the microcomputer industry, and this will benefit all microcomputer users.

COMPATIBILITY

Of course, when anyone buys a microcomputer, support has to be the main consideration. Following the IBM PC boom that occurred in 1981, many smaller companies came out with "compatibles" that were offered at a much-reduced cost compared to the IBM PC. Many of these compatibles were bought up, but apparently not enough to prevent financial disaster for many of these companies. Some of them went under, leaving the owners of these companies holding the bag. Also, some of the compatibles were not quite as compatible as advertising led their buyers to believe.

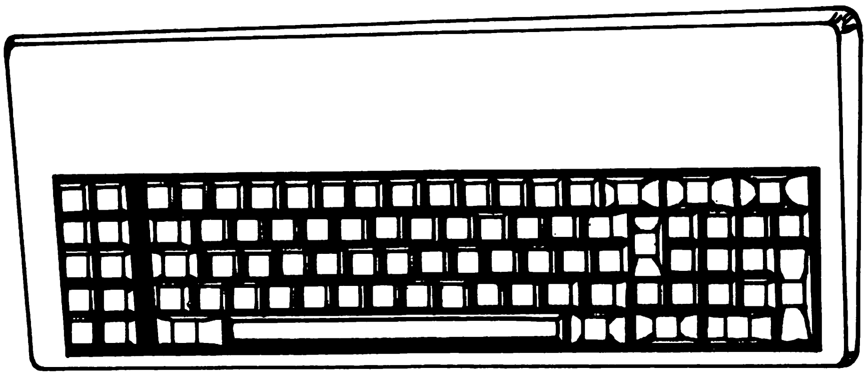
None of these situations apply to the AT&T PC-6300. One can be certain that the corporate giant, AT&T, will be with us until icicles form in Hades. Their customer support in the field of telephone communications is already established, and this will be carried over to the microcomputer division. The compatibility of the AT&T PC-6300 is well documented. As a matter of fact, AT&T has prepared an *approved* list of nearly 1000 MS-DOS programs that have been tested and found to be fully compatible with the PC-6300. Therefore, the AT&T PC-6300 can be considered to be a very safe buy. It can also be considered a very high level machine that will take the microcomputer user about as far as he/she desires. To go to a higher level machine would probably mean a supermicro or minicomputer costing several times more than the $3000 to $5000 price tag of the AT&T PC.

In a free enterprise system, competition is the name of the game. I think we will see one-upmanship tactics used to a very high degree among the major manufacturers of high-level personal computers, of which group AT&T is now part. This means that AT&T cannot rest on its laurels, but will have to continuously improve their existing products. Shortly after the announcement of the AT&T PC-6300, IBM announced their new IBM PC-AT. According to IBM, the "AT" stands for Advanced Technology, and is in no way a play on the AT&T name . . . again according to IBM. As these and other corporate giants battle it out for a big chunk of the microcomputer marketplace, microcomputer users will be offered higher-level options and products that will be of tremendous benefit.

In summary, the AT&T PC-6300 has taken everything that has gone before and made it better, while maintaining a very high degree of compatibility. The main advantage of the AT&T PC over the IBM-like machines is its operating speed, which is generally superior in all applications. The PC-6300 has become an established product, even though it has been out for a relatively short period of time. Much of this is due to the support that already existed for this type of machine. With its capability of accepting full 16-bit option boards, one can expect the AT&T PC to be the target of many outside option suppliers who have been limited in the past by the 8-bit-only buses of previous MS-DOS machines. This may open the door for a whole new line of extremely fast and versatile options that will address established operations in a new way and offer a large range of brand-new capabilities never before thought of for microcomputers.

Networking, communications, mainframe connections, and much more are all in the future of the AT&T PC-6300. It is my opinion that this machine will be with us for quite some time and will not quickly fade into obsolesence. The buyer of the AT&T PC-6300 can be certain that his/her purchase is a timely one and will provide a microcomputer system that will receive widespread industry support and therefore provide many years of high-level service.

Chapter 2



GWBASIC Programming

The AT&T PC-6300 is programmed in GWBASIC. BASIC, developed at Dartmouth College, is an acronym for Beginner's All Purpose Symbolic Instruction Code. While all programs written in BASIC are very similar, there are some differences in GWBASIC and Apple BASIC, for instance. In other words, the PC-6300 uses one *dialect* while the Apple uses another. When you switch from computer to computer, it is necessary to know something about the exact nature of the BASIC dialect for which it is programmed.

If you've had some experience programming in BASIC and are simply making a transition from another machine to the PC-6300, you may wish to skip over this chapter. However, much of the material discussed here will serve as an excellent refresher course for those of you who already know how to program in BASIC. This chapter is really aimed at those persons who have never used BASIC before and are relatively new to the field of computers in general. The following discussion will take you on a step-by-step tour through portions of the GWBASIC language. If you study the materials in this chapter and input each program to the computer while you are learning, by chapter's end you will know a great deal about computer programming and will be better prepared to understand some of the programs presented in later chapters. While this chapter will not discuss every statement, function, and command in GWBASIC, it does discuss those used most often. A later chapter will serve as a reference to the language the PC-6300 understands, and will thoroughly explain each statement, command and function in detail.

BASIC language is quite powerful and uses easy-to-understand words that produce various computer operations. Each of these words is known as a command, statement, or function. Unlike foreign languages, such as French or German, the BASIC language is a small, finite language which is quite easy to master within a short period of time. Each command, statement, or function in BASIC usually has an English equivalent that describes the operation the computer will perform. Take the

PRINT statement, for instance. This is a statement (common to all dialects of BASIC) which causes the computer to print something on the display screen. An INPUT statement tells the computer that the operator needs to supply some information at a certain point in the program: the computer waits for this input from the keyboard. There are many other direct equivalents between BASIC and English. These will be discussed in this chapter.

A FIRST PROGRAM

A computer program written in BASIC must consist of a minimum of one program line. Some programs will have hundreds or even thousands of lines, but these programs are developed a line at a time. In BASIC, each line must have a number. Normally, lines are numbered in increments of 10, as in 10, 20, 30, 40, etc., or even 100, 110, 120, 130, etc. You can just as easily start numbering in increments of 1, as in 1, 2, 3, 4, 5. This is not normally done because few programs are written in finished form the first time. Usually, you will want to go back through the program and insert additional lines in various locations. If you number in increments of 10, there is plenty of space to insert additional lines. If your program contains two lines numbered 10 and 20, and you find it necessary to insert another line between them, you could number it 11, or 12, or 13. Usually you would number it 15 so that you could, if necessary, add other lines before and after it.

The first program you will write will consist of only one program line and will be used to display the word HELLO on the computer screen. Throughout this chapter, we will add to this one-line program to demonstrate other parts of BASIC programming.

Begin with the simple programs. Type in the number 10. Then hit the space bar and type PRINT "HELLO". That's all there is to it. You have written your first BASIC program. Your screen should look like this:

```
10 PRINT "HELLO"
```

There may be other printing on the screen left over from when you first activated the computer. Don't worry about this. Just make sure that the line discussed above is on the screen. Now, press the Enter key. Your program has been committed to computer memory. Examine the line on the screen carefully. Make sure it looks like the example given here. Now that the program has been input, it's time to execute it. In GWBASIC (and nearly every other dialect as well), this is done by typing RUN and then pressing Enter. You should see the word HELLO appear just below your program line.

If for some reason the computer cannot run your program, it will display what is known as an *error message* on the screen. This is a message that indicates that a program cannot be run and gives a clue as to why the computer found it unacceptable. One of the most common messages is Syntax Error. This means that you have input a statement that is unknown to the computer, or is in a form the computer cannot understand. If you misspelled the statement PRINT, for example, the syntax error message would appear.

I should point out that some of my instructions deal with custom more so than with what the computer requires. For example, if you had typed any of the following lines, the PC-6300 would still print HELLO on the screen.

```
10PRINT "HELLO"
10 PRINT"HELLO
10PRINT"HELLO
```

Some other dialects of BASIC might not accept the absence of a space between the line number and the first statement or function. Still others require quotation marks on both sides of the word you wish to display on the screen. The absence of the closing quotation mark in this simple program would have made no difference to the PC-6300, but in more complex programs you could get into serious trouble by not sticking to the proper format. However, you will always be safe if you include a space after the line number, and enclose in quotes all words to be printed on the screen.

As an exercise in machine operation, let's assume that you misspelled the word PRINT in our original program. When you tried to run the program, you would have gotten the syntax error message. It would now be necessary for you to correct this program line. Type in the following program line:

```
10  PRNT "HELLO"
```

Notice here that the word PRINT has been purposely misspelled. Now, type RUN and press Enter. You should see this on the display screen:

```
Syntax error in 10
ok
10  PRNT "HELLO"
```

You will also see the flashing cursor below the number 10. This whole scenario means that there is an error in line 10 which the computer has thoughtfully displayed for you to correct. On the PC-6300, a correction of this nature is quite simple. On the right side of your keyboard are keys marked with arrows pointing up, down, left, and right. Press the right arrow key once and the flashing cursor will move one position to the right. The other keys will move the cursor in the direction they point. In this particular case, we need to insert the letter I immediately before the N in the misspelled statement PRNT. To do this, keep hitting the right arrow cursor key on the keyboard until the blinking cursor is positioned beneath the letter N in PRNT. Now, locate the INS key at the bottom right of your keyboard. This stands for *insert*. With the flashing cursor beneath the letter N, press the INS key once. The computer now knows that you will be inserting something ahead of the N. Now, type in the letter I. Eureka! The misspelled statement now reads PRINT. As is always the case, it is now necessary to press the Enter key to commit this line to computer memory. Now type RUN. You should now see the word HELLO displayed on the screen followed by the BASIC prompt "Ok."

We can also easily set up another error condition by typing in

```
10  PRINTT "HELLO"
```

This will set up the same error message on the screen when you attempt to run the program. In this case, it is necessary to delete the extra T in the word PRINTT. Again press the right arrow cursor key until the flashing cursor lies beneath the second T. Now, locate the DEL key just to the right of the INS key. Press the DEL key once and the extra T will disappear. You can now press Enter and run the program again for a successful execution.

What have you learned to this point? First, any program in BASIC consists of BASIC keywords such as PRINT contained in program lines. Each line must have a number. You have also learned that whenever a program line is input, the Enter key must be pressed to commit it to current computer memory. You now know how to use the cursor positioning keys and the INSert and DELete functions to quickly edit mistakes in program lines without having to retype the entire line. In this particular program, retyping the line might have been almost as quick as going through the editing functions outlined, but imagine a program line that may be 50 or 60 characters in length. It might take you several minutes to input such a line, but only a few seconds to edit it.

CLEARING THE SCREEN

When you ran the previous program, you undoubtedly noticed that, while the word HELLO was printed in accordance with the program, all of the previous information that was displayed on the screen before the program was run remained there as well. In most situations, you will want to clear all information from the screen before your program is actually run. Most programs contain a BASIC statement at the beginning to clear the screen before any additional materials are printed. To accomplish this, we use the CLS statement in GWBASIC. CLS is an acronym for "clear the screen." When this statement is encountered, all information currently on the screen is erased completely. This gives us a clean slate for any other information that will be printed by succeeding program lines.

The previous program you typed in is still contained in computer memory. To see the program that the computer is currently ready to execute, simply type LIST. This means list the program currently in memory. Do this now and then hit Enter. You should immediately see your original program appear on the screen. We will now expand this original program to do the following:

1) Clear the screen
2) Print HELLO on the screen

Since we again wish to print the word HELLO, we will simply use the program line already in memory that accomplishes this. All we have to do is include another line to clear the screen. This line must go before the original program line, because we wish to clear the screen *and then* print HELLO. To expand the previous program, type in the following line:

```
5 CLS
```

Now, press Enter, then type LIST and press Enter again. You should see the following:

```
 5 CLS
10 PRINT "HELLO"
```

Your program is now twice its original size because it contains two lines instead of one. When this program is executed, the computer will clear the screen and then display the word HELLO. Again, type RUN and press Enter. All you should see on the screen following this program run is:

```
HELLO
Ok
```

The screen was cleared, HELLO was printed, and the computer prompted you that the program is finished by displaying Ok on the screen. For the moment, pay no attention to the blocks at the bottom of the screen. This is called your *key* and will be discussed later. The CLS statement has no effect on the key display.

You probably haven't realized it yet, but by following the previous instructions, you have been using two modes of inputting information to the computer. These modes are *direct mode* and *program mode*. Whenever a line number is typed in followed by a BASIC statement line, this is the standard program mode. In other words, we are inputting a set of instructions that are not to be executed until we run the program. However, when the word RUN has been typed in, it has not been preceded by a line number. This is the direct mode of input. When the direct mode is used, we are telling the computer to do something right now (as soon as Enter is pressed). RUN is known as a *command.* It is an immediate call to the computer to execute the program currently in memory. LIST is also a command which was entered in direct mode, to tell the computer to recite the program in memory. Either of these commands can also be used in program mode when preceded by a line number. This is not true in many other dialects of BASIC, but is a useful function which is good to have when writing complex programs. The use of commands in program mode will be discussed later.

CREATING A LOOP

A loop is a common occurrence in BASIC programs. It is an expression you will hear again and again. The dictionary defines a loop as a circle or a continuous repetition. This is what a loop is in BASIC as well. Let's take our previous program and make a loop out of it so that it prints HELLO again and again and again on the screen. Type the following:

```
 5 CLS
10 PRINT "HELLO"
20 GOTO 10
```

Hey! There's a new word in there that we haven't discussed before. This is one of the most useful statements in BASIC, common to all dialects. GOTO tells the computer to do just what the English equivalent would indicate. It tells the computer to go to another portion of the program and execute that line *and all successive lines*. Before running the program, let's discuss it further. Line 5 clears the screen. Line 10 prints the word HELLO on the cleared screen. The GOTO statement in line 20 tells the computer to go to line 10 and execute the program from that point on. Here's what will happen, at least from the computer's point of view:

1) Clear the screen (line 5)
2) Print HELLO on the screen (line 10)
3) Go back to line 10 (line 20)
4) Print HELLO on the screen (Line 10)
5) Go back to line 10 (line 20)
6) Print HELLO on the screen (line 10)
7) Go back to line 10 (line 20)

This process will go on forever, or until the program is manually halted. This loop is known as a *continuous* or *infinite* loop. The program can never end on its own because the last line in the program always tells the computer to go back and execute a previous line. Now run the program. You will see the word HELLO displayed at the left side of the screen from top to bottom. You can let this program run for a minute, an hour, or a year, and it will keep going. The only way you can stop program execution is to press the CTRL key at the left of the keyboard and then the BREAK key at the right of the keyboard to bring about a manual halt. This simply means that the programmer stopped the program run.

This program only clears the screen once because line 5 is only executed once. The program loops, then, is between lines 10 and 20. These lines will be executed over and over again. We could also include line 5 in the loop simply by changing line 20 to the following:

```
20  GOTO 5
```

This change can be made using the editing functions, or you can simply type in the line all over again. When this program is run, the word HELLO will be displayed in the upper left-hand corner, but it will be constantly flickering because the screen is being cleared each time before the word is printed. When the computer writes on a clear screen, it always begins printing at the upper left-hand corner unless it is told through programming to begin elsewhere. When the screen is not cleared, each succeeding printed character will be displayed at the left side of the screen one row below the previously printed characters.

Admittedly, none of the programs so far have been useful for anything other than instruction. The computer has not been used to display useful information on the screen. Please be patient. It is absolutely mandatory that you understand the basics behind computer programming before you can begin to make your programs "intelligent." Be assured that if you can understand the concepts behind the programs already discussed, you're getting closer to being able to write some good programs on your own. So far, you've learned a little about the PRINT statement, CLS, and GOTO. Hopefully, you're also becoming a bit more comfortable in writing programs. Do not proceed further unless you understand the previous program examples completely. To do so would only bring about confusion. You must understand what has gone before to be able to understand what is to follow.

MORE ABOUT LOOPS

The previous program caused the word HELLO to be printed over and over again on the screen. The GOTO statement is also referred to as a *branch* or a GOTO branch. There are other types of branch statements that we will be learning about later, but GOTO is the one seen most often.

Endless loops such as the one encountered in the previous program are useful in some programs, but most of the time it is necessary to exit the loop after a certain number of passes have occurred. Think of a pass as one cycle of the loop. By modifying the previous program, we can cause the loop to terminate after a specified number of passes. The following program is simply the previous program with two extra lines.

```
 5  CLS
10  PRINT "HELLO"
```

```
15  C=C+1
16  IF C=10 THEN END
20  GOTO 10
```

Line 15 would not seem to contain a statement, but it does—or at least the computer assumes that it does. Line 16 contains a new statement, called the IF-THEN statement. This is a test statement. It checks to see if a certain condition is true and, if it is, then it may bring about a branch or execute another statement. Let's start with line 15. The letter C represents what is known as a *variable*, a symbol which can be equal to anything we assign to it. The method of assignment shown in line 15 is sometimes referred to as a counting assignment. Each time line 15 is executed, variable C will be incremented, i.e., increased by 1.

Let's go through a partial sample run. Line 5 clears the screen. Line 10 prints HELLO on the blank screen. Line 15 is then encountered. At this point in execution, C is automatically equal to 0 because no previous assignment to C has been made. Line 15 is really the first assignment to variable C. The new assigned value will occur at the left of the equal sign, while the old value of C is seen at the right. The old value of C is 0. Therefore, the new value of C will be equal to 0 + 1, or 1.

Now, line 16 says if C is equal to 10 then END the program. When line 16 is executed for the first time, C will be equal to 1, so line 16 technically does nothing. It will only execute when C is equal to 10. Now, the GOTO statement in line 16 technically does nothing. It will only execute when C is equal to 10. Now, the GOTO statement in line 20 is encountered, and the loop begins. Line 10 prints HELLO again, and line 15 is encountered for the second time. Remember, the new value of C (the one to be assigned) is located at the left of the equal sign, while the old value of C is to the right. Since line 15 has already been executed once at this point, the old value of C is now equal to 1. Since line 15 states that C becomes equal to C + 1, the new value of C is equal to 1 + 1, or 2. Again, line 16 detects that the new value of C is not equal to 10, so the END statement is not executed. On the next pass of the loop, C is incremented to 3 (2 + 1). The loop will continue to cycle until C is finally incremented to a value of 10. At this point, line 16 detects the value of 10 and ends the program. The END statement is common to nearly every dialect of BASIC and is a signal to the computer to end the program.

Here is another way this program could be written.

```
 5  CLS
10  PRINT "HELLO"
15  C=C+1
16  IF C=10 THEN 30
20  GOTO 10
30  END
```

This program does exactly the same thing as the previous one. You can now run either program and you will see that HELLO is printed 10 times on the screen. The program then terminates. What was previously an endless loop has now been transformed into a loop with 10 passes. In the second program, line 16 has been altered to bring about a branch to line 30. Line 16 could also have been written.

```
IF C=10 THEN GOTO 30
```

In GWBASIC, the GOTO does not have to be used in an IF-THEN statement branch. The first example was the most efficient, but this second example will do exactly the same thing, although it requires an additional program line.

You will notice in this latter example that when C is incremented to 10 in line 15, line 16 detects this condition and branches directly to line 30. In this case, line 20 was not executed at all on the last pass of the loop. In BASIC, program lines are normally executed in ascending order. In other words, line 5 is executed, followed by the execution of line 10, and then 15, etc. If we inserted a line 7 between 5 and 10, then line 7 would be executed after line 5. The only way to change the order of execution is by using branch statements, which were introduced in this last example. These statements will be discussed in more detail later in this chapter.

There is one other element of these examples that needs further discussion. There is a statement that is common to every dialect of BASIC. This is the LET statement. Most computers today will handle this statement, but do not require its use. Line 15 in the preceding examples could have been written.

```
15  LET C=C+1
```

However, the way it was handled in the examples is the most efficient method of programming, because it requires less typing. In GWBASIC, the LET statement is optional, and the line C=C+1 is exactly equivalent to LET C=C+1. This explains my previous remark that line 15 did not appear to contain a statement. The PC-6300 assumes the LET statement automatically.

FOR-NEXT LOOPS

Probably the most useful loop found in BASIC is the FOR-NEXT loop. Just like IF-THEN, FOR-NEXT may be thought of as one complex statement. In order to demonstrate the FOR-NEXT loop, let's start from scratch again and erase the program currently in memory. To do this, a command called NEW is used. (You will remember that a command is entered in direct mode without a line number, and is executed immediately.) To erase a program from current computer memory, simply type in the NEW command and then press Enter. Your program is now erased. You can confirm this by using the LIST command. When LIST is entered, you will see that nothing is displayed. The program is gone.

The following program demonstrates the FOR-NEXT loop. It is exactly the same as the previous program which was used to print HELLO on the screen 10 times.

```
10  CLS
20  FOR X=1 TO 10
30  PRINT "HELLO"
40  NEXT X
```

As before, line 10 clears the screen. Line 20 begins the FOR-NEXT loop. Only the FOR portion of the statement is contained on this line. You should think of FOR as starting the loop and NEXT as ending it. Any lines between FOR and NEXT are a part of the loop; in this case, there is only one. Line 30 will cause HELLO to be displayed on the screen.

In this example, X is a variable, and is assigned a value of from 1 to 10. Here's how it works. When line 20 is executed for the first time, X is assigned a value of 1. Line 30 is then executed and HELLO is displayed on the screen. When the NEXT X is encountered in line 40, this brings about an automatic branch back to the line containing FOR. The loop now begins its second pass, and variable X is assigned a value of 2. Unless told otherwise, the FOR-NEXT statement will automatically increment its value by 1 during each pass. Line 30 is executed again. Line 40 causes the automatic branch back to line 20, and the loop continues cycling. When 9 passes have been completed, the last branch to line 20 occurs. The variable X is now equal to 10. Line 30 is executed, but when to is encountered, the loop has "timed out," i.e., X has reached its maximum assigned value of 10 and the loop is automatically exited. At this point, the program will END because there are no other lines following line 40, the end of the loop. The program could include an additional line following the loop sequence, such as

```
50  END.
```

Since END is automatically assumed when there are no more program lines to execute, this line is technically superfluous. If it helps you understand the program better, however, it should certainly be included.

This example has shown how a loop may be used to execute a program line or lines contained within the loop a specific number of times. We can demonstrate another useful purpose of loops by changing line 30 to:

```
30  PRINT X
```

This is the first usage of the PRINT statement to print out the value assigned to a variable. (Note that quotation marks are not used.) When the PRINT statement is used with a variable, the current value of that variable will be displayed on the screen. Run the program in its present form and you will see a counting sequence composed of the numbers 1 through 10. These are the sequential values of X. Remember, do not use the quotation marks when you wish to display the value assigned a variable on the screen. To see what happens when you do, change line 30 to 30 PRINT "X". When this program is run, the letter X will be displayed 10 times on the screen. The quotation marks tell the computer not to print the value of variable X, but instead to print whatever is inside the quotes.

It was mentioned earlier that a FOR-NEXT loop will automatically increment its variable by 1 unless told to do otherwise. The FOR-NEXT loop is also known as the FOR-NEXT-STEP loop, with the STEP portion indicating the value by which the variable is to be incremented. In the previous program, while you typed line 20 as

```
20  FOR X=1 TO 10
```

the computer saw it as:

```
20  FOR X=1 TO 10 STEP 1
```

It automatically incremented X by 1 on each pass of the loop.

Let's change the program at this point to bring about a different STEP value. To do this, we can use another command designed to make editing of programs extremely easy. In direct mode, type EDIT 20 and then press Enter. When this command is entered, line 20 will appear on the screen and the flashing cursor will be seen beneath the 2 in 20. Use the right arrow cursor control as described previously to move to the point one character past the 10 in line 20. Now, press the space bar and then type STEP 2. Your program line should look like this:

```
20  FOR X=1 TO 10 STEP 2
```

This line now tells the computer to increment X by two during each pass of the loop. Here's how it will work. When line 20 is first executed the value of X will be 1, and when the loop cycles for the second time, X will be incremented by two. Using a previous program example, the line will be equivalent to X=X+2 on the second and all succeeding loops. Now, run the program. Since line 30 displays the value of X on the screen, you will now see the numbers 1, 3, 5, 7, and 9 displayed vertically. This is the result of creating a FOR-NEXT loop which is stepped by 2.

You may be surprised that the final value of the loop (10) was not displayed. This number does not come up when X is stepped by 2 because the next two-step jump after 9 would be 11. Since X can only be equal to a maximum value of 10, the loop terminates. This example provided an odd number count, which will always be the case when a step of two or any even number is used, provided that the loop starts on an odd number. However, if the loop had started with 2 (i.e., X=2 TO 10 STEP 2) the value of X would always be even.

FOR-NEXT loops may also start with a zero. If we change line 20 to read:

```
20  FOR X=0 TO 10 STEP 2
```

the count sequence will be 0, 2, 4, 6, 8, 10. Notice here that another number has appeared. Let's go further with this. Change line 20 to

```
20  FOR X=0 TO 10
```

and then run the program. You will see that the numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 are displayed on the screen. How many passes has the loop made? The answer is 11.

When programming computers, it is always necessary to remember that 0 is a number. It is treated by the computer just like any other number, so if you want a loop to start at 0 and end at 10, it is the same count (cycle-wise) as creating a loop that counts from 1 to 11. Not thinking of 0 as a viable number creates many headaches for new programmers. Remember this, and debugging programs may become a bit simpler.

At this point, it will probably be necessary for you to review the entire section on loops, and especially the section on FOR-NEXT loops. In any programming language you will rely heavily on loops, so if you do not understand how a FOR-NEXT loop works, reread the previous materials until you do.

GOING FURTHER WITH FOR-NEXT LOOPS

Now that you're sure you understand loops and FOR-NEXT loops, let's delve further into this subject. All of the previous examples used positive numbers and counted upward (incremented). FOR-NEXT loops are neither restricted to positive numbers nor to incrementing these numbers. Using the previous program, change line 20 to:

```
20  FOR X= -10 to 1
```

Now RUN this program. You will see the value of X displayed on the screen starting with −10 and moving up to +1. In this case, the loop makes 12 passes. When X is equal to −10 during one pass, it will be equal to −10 + 1, or −9 during the next; when the pass is completed where X is equal to −1, the next pass will assign X the value of −1 +1, or 0. This is still a positive step pass, in that X is always incremented by +1.

We can also make the FOR-NEXT loop count backward. If you've skipped ahead of me, you may have tried changing line 20 to:

```
20  FOR X=10 TO 1
```

This is often a first-time attempt to get a loop to decrement or decrease the value of X, in this case to count backward from 10 to 1. This won't work—at least not in this form. To decrement a loop it is necessary to insert a negative STEP value. Try this:

```
20  FOR X=10 TO 1 STEP -1
```

Now RUN the program. It should work perfectly and you will receive a count on the screen starting with 10 and ending at 1. This loop is identical to 20 FOR X=1 TO 10 in that a total of 10 passes will be made. However, if you're going to *use* the value of X within the loop, the situation is completely different. In the first example, X will be equal to 10 on the first pass, 9 on the second, etc. In the second example, X will be equal to 1 on the first pass, 2 on the second, etc. If you wish to experiment, change the negative step value to −2 or −3, and see what happens to the value of X.

In all of the FOR-NEXT loops examples so far, whole numbers have been used as step values for demonstration purposes. FOR-NEXT loops are certainly not limited to them. Let's change line 20 again to:

```
20  FOR X=1 TO 10 STEP .5
```

Now RUN the program. You will see that this step value is reflected in the loop. The variable X will now be equal to 1, 1.5, 2, 2.5, etc., and this particular loop will make 19 passes. You could also use −.5 and loop values from 10 to 1, to produce the same figures in descending order.

Variables may be inserted for any numeric value in the loop. Let's erase the program currently in memory (by typing NEW) and start over again. Input the following program:

```
10  CLS
20  A=1
30  B=10
40  C=.5
50  FOR X=A TO B STEP C
60  PRINT X
70  NEXT X
```

This program will do exactly what the previous one did—count from 1 to 10 in increments of .5. Here, however, variables have been used to assign values to the loop. In line 50, X will have a low value of A, a high value of B, and a step value of C. In this case, the letters A, B, and C have been used to represent numbers, just as X has been so used in all of these examples. The FOR-NEXT statement assigned values to X, while lines 20-40 assign values to A, B, and C. Remember, these can be classified as LET statement lines and, unless the values of A, B, and C are changed by other LET statement lines, they will always be equal to the values originally assigned.

At this point, you should review all of the materials presented thus far on loops, and begin experimenting on your own. Try any values in the FOR-NEXT loops you desire, and see if you can get your loops working properly. If you use a wide value range such as:

```
FOR X=1 TO 20000
```

it will take your computer some time to count through all these numbers. It will also take it far longer to print all of these values on the screen. The more program lines contained in a FOR-NEXT loop, the longer the execution time. For this reason, it is absolutely essential that any lines that do not absolutely have to be in the loop are placed outside it.

MORE ON THE PRINT STATEMENT

As you will recall, the PRINT statement was the first one introduced in this chapter. We know that any characters enclosed in quotation marks following the PRINT statement will be displayed on the screen exactly as typed. If the quotation marks are omitted and a variable is substituted for the character string, the *value* of the variable will be displayed on the screen.

Erase the program currently in memory and type in the following lines:

```
10  CLS
20  FOR X=1 TO 5
30  PRINT "HELLO"
40  NEXT X
```

This is almost identical to a previous program, but this one will delve more deeply into the PRINT statement. When you run this program you will see HELLO displayed 5 times on the screen in a vertical format, each word written directly beneath the previous one. Now, let's edit program line 30. (Simply type in EDIT 30 and press Enter.) Use the proper cursor control key to move to a point one character past the closing quotation marks, type in a semicolon, and press Enter. Program line 30 should now look like this:

```
30  PRINT "HELLO";
```

Run the program. Look what happened! HELLO was again printed 5 times on the screen, but the words were printed horizontally. The semicolon, then, instructs the computer to display the characters in a horizontal format, with the second HELLO written immediately to the right of the first. The result is:

```
HELLOHELLOHELLOHELLOHELLO
```

All of the words are run together, but this can be easily corrected. Edit line 30 again, moving your cursor to the point just beneath the closing quotation marks. Hit the INS key and the space bar once. Line 30 should now look like this:

```
30  PRINT "HELLO ";
```

Run the program again. This look much neater. The computer is doing the same thing it did before, except there is an extra character within the quotation marks. This character—a space—is not seen on the screen, but serves to separate the words. I stated earlier that a zero is treated like any other number by the computer. The same applies to the space character. While you can't actually see the character produced by the space bar, the computer can, and it treats a space just like it does any other letter, numeral, or symbol.

Now, let's combine a quoted word and a numeric variable in one PRINT statement line. Change line 30 to:

```
30  PRINT X "HELLO"
```

Now run the program. Here, you will see that the value of X is displayed followed by a space and then the word HELLO. (The space is automatically inserted by the computer and has nothing to do with the spacing between the variable in the program line and quoted phrase.) We can reverse this by first printing the quoted phrase and then the value of X. Change line 30 to read

```
30 PRINT "HELLO" X.
```

When this program is run, HELLO will be displayed followed by the value of X.

Many dialects of BASIC require the use of a semicolon to separate quoted phrases, or quoted phrases and variables, to be printed in horizontal format with one PRINT statement. This "horizontal format" applies to a single line only. Change line 30 to:

```
30 PRINT "HELLO" X;
```

This produces a "true" horizontal format, with *all* of the displays created by line 30 printed horizontally on the screen. The semicolon will disable the computer's automatic "line feed" function wherever it is used—in this case, at the end of a PRINT statement line. The use of a semicolon between HELLO and X is not necessary in GWBASIC, but to get a complete horizontal format, however, you must have an end-of-line semicolon or comma.

When a comma is used to separate two elements within a PRINT statement line, the results are quite different. Change line 30 to read:

```
30 PRINT "HELLO",X
```

Now run the program. Here, the word HELLO is printed and then followed by 10 spaces before the value of X is printed. This creates an effect similar to TAB on the typewriter. In this mode, the computer is displaying information in two print zones, each 165 characters wide. The word HELLO is in one zone, and the value of X is at the beginning of the other. You won't see this used very often, but it is handy for displaying menus, charts, and other screens common to computers.

VARIABLES

We have already dealt with one type of variable in the preceding discussions. However, in BASIC programming, there are several types. A variable is something that is used to represent a real value or quantity. The variables we have been dealing with are called *numeric* variables, because the letters are used to represent numbers. In discussing FOR-NEXT loops, the variable X was used to represent the loop value or count. Any other letter would have sufficed as well, or even a combination of letters, such as XYZ or a letter/number combination, such as X1. In BASIC, variables are comprised of letters or combinations of letters and numbers, but never numbers alone.

In addition to numeric variables, there is another type of variable used to handle what are known as *string* values. String values are often words, letters, or combinations of letters and numerals. Just as we used numeric variables to represent numbers, we can use string variables to represent letters or combinations of numerals and letters. The name of a string variable always terminates with a dollar sign ($). Where the letter A may be used as a numeric variable to represent a number, A$ can be used to represent a word, letter, or letter/numeral combination. Note also that A and A$ are *different* variables, and could both be present in a program. String variables may also be used to represent numbers alone, although this is rarely done except in special programming situations.

The following program demonstrates the use of a string variable to represent the word HELLO.

```
10 CLS
20 A$ = "HELLO"
30 FOR X = 1 TO 10
40 PRINT A$
50 NEXT X
```

Here, the word HELLO is assigned to the string variable A$. Whenever a string variable is assigned, its valued must be contained in quotation marks, regardless of whether it consists of letters, numerals, or combinations thereof. If the line were typed as A$ = HELLO, you would get a syntax error message. When this program is run, the word HELLO will be displayed as before 10 times. To the computer, A$ means HELLO because this is its assigned value. By inserting a semicolon following A$ in line 40, the word HELLO will be printed in a horizontal format, just as before.

That semicolon is important in more ways than one. To illustrate, change line 40 to

```
40  PRINT A$X
```

and run the program. Remember, in PC-6300, it's not *always* necessary to separate the elements in a PRINT statement by a semicolon to have them printed next to each other on the same line. We could also have written this line as 40 PRINT A$;X and achieved the same results.

However, let's reverse this sequence and print X first, followed by A$. Change line 40 to:

```
40  PRINT XA$
```

Again, we have not used the semicolon for separation. Run the program. Horrors!! Nothing happens! In this case, it is absolutely essential to use a semicolon between the numeric variable X and the string variable A$. You fooled the computer! Remember that the dollar sign always marks the end of the string variable. In the first example, where A$ was followed by X, the computer recognized the dollar sign as terminating one type of variable and then recognized X as the numeric variable. However, in the second example the computer interpreted XA$ not as the numeric variable X followed by the string variable A$, but simply as one string variable named XA$. Since there was no assignment to a string variable named XA$ in a previous program line, the computer assumes that XA$ is equal to nothing—not zero, but nothing. The actual assignment the computer made might have looked like XA$ = "". Notice that there is not even a space character here. (This is sometimes called the "null string," and occasionally it can be useful.) When the computer interpreted line 40, it read it as, "Don't print anything, 10 times." And that's exactly what it did.

To get the program to work properly, change line 40 to:

```
40  PRINT X;A$
```

Now run the program. You should see the value of X displayed on the screen and on the same line separated by a space, the word HELLO. This combination is printed 10 times in vertical format on the screen. Now change line 40 to

```
40  PRINT X;A$;
```

to display the horizontal combination of X and A$ in horizontal format across the screen. A portion is printed on the next lower row simply because it was impossible to squeeze 10 printings of the numeric variable and the word on one program line. When a computer display exceeds the length of one line, the remainder is automatically displayed on the next.

Remember, in BASIC there are two principal types of variables. The numeric variable represents a number or numeric quantity, while the string variable is terminated with a dollar sign and represents a string quantity. When assigning a value to a string variable, the value must be enclosed in quotation marks.

Numeric variables can be manipulated or processed in more ways than string variables. For instance, if numeric variable A is assigned a value of 10 and numeric variable B is assigned a value of 5, a computer line such as:

```
40  C = A/B
```

is permissible. The slash indicates that A is to be divided by B. The computer will automatically substitute the assigned values of A and B, complete the mathematical operation, and assign to variable C the value of 2 (10/5 = 2). The following shows other symbols which represent mathematical operations:

C=A + B	(add A and B)
C=A - B	(subtract B from A)
C=A * B	(multiply A by B)
C=A ^ B	(raise A to the power of B)
C=SQR(A)	(take the square root of A)

On the other hand, string variables, even though they may be used to represent numeric quantities, cannot be used to perform mathematical operations. Take the following program for instance:

```
10  A$="4"
20  B$="8"
30  C$=B$/A$
40  PRINT C$
```

It's not necessary to input this program, since it will result in an error message. Lines 10 and 20 are perfectly legal (as is line 40, for that matter), but line 30 creates the problem. These are string variables and standard mathematical operations cannot be performed on string variables. If you change line 30 to

```
30  C$=B$ * A$
```

or

```
30  C$=B$ - A$
```

you will still get the same error message, which is "type mismatch." This simply indicates that you tried to perform a mathematical operation on a string variable. It simply won't work. (There is a way that string variables can be converted to numeric variables, but this is saved for a later discussion.)

One "mathematical" operation can be performed with string variables. Input the following program:

```
10  A$ = "3"
20  B$ = "4"
30  C$ = A$ + B$
40  PRINT C$
```

When the program is run, the screen will display the number 34, which is not the sum of the numbers 3 and 4. The value of C$, 234, is the result of printing A$ and B$ side by side. We can demonstrate this further by changing lines 10 and 20 as follows:

```
10  A$ = "HELLO"
20  B$ = "GOODBYE"
```

After these changes are made, run the program again and the computer will display "HELLOGOODBYE". The "+" operator alone can do double duty: in mathematics it means "add," but with strings it simply connects them end-to-end—a nonmathematical operation called "concatenation."

INPUT STATEMENT

In many types of computer programs the user is expected to supply information from the keyboard while the program is running, which the computer uses in the context of the program. Sometimes it will output the answer to a mathematical problem based upon the numbers input at the keyboard. At other times the keyboard input is used to help the computer decide which program lines to execute. The INPUT statement is tied very closely to the variables just discussed, and also resembles the PRINT statement in many ways.

When the computer encounters the INPUT statement it temporarily halts program execution, and will not resume until something has been input at the keyboard and the Enter key depressed. The information typed in is then assigned to a variable which may be a string or numeric variable, depending on the information the computer is looking for. The following program illustrates the use of INPUT to assign the user input from the keyboard to a numeric variable. It will then process a mathematical function using the input as one of its variables.

```
10  CLS
20  INPUT A
30  X = A * 2
40  PRINT X
50  END
```

You should see some very familiar statements and operations in this program, the only thing different being the INPUT statement in line 20. This program allows the user to supply any number, which is then multiplied by 2 and assigned to variable X. The value of X is then printed on the screen. After line 10 clears the screen, the statement INPUT A causes the computer to halt and display a question mark. The computer is saying to you, "Give me a value for variable A." And it will not continue until you type in a number and press Enter. Line 30 assigns, to a new variable X, the value of A times 2. Line 40 tells the computer to display the value of variable X on the screen, and line 50 uses the END statement to terminate program execution. (In GWBASIC, the statement is unnecessary unless there are other lines in your program.) Run the program. When the question mark prompt appears, type in the number 10 and press Enter. Immediately, the computer will display the number 20 on the screen (10 * 2 = 20), and the program terminates. You can use this program over and over again to automatically double any value you input at the keyboard.

Since the variable used with the INPUT statement in line 20 is not terminated with a dollar sign, the computer expects a numeric input rather than a string input. Therefore, anything typed in via the keyboard must be a purely numeric quantity. The computer will accept nothing else. Run the program again and when the question mark appears, type in a few letters. As soon as you hit Enter, you will get the error message "Redo from start." The computer has said, "No way!" It knows that it must have a numeric input via the keyboard to continue execution. You can simply press the Enter key without typing in anything else and execution will continue. You have, in effect, input a value of 0 to the computer; its output answer will be 0, since 2 times 0 is still 0.

This program can be expanded upon endlessly to perform many different types of mathematical operations. It also needs to be cleaned up a bit because when you type in the number, it remains on the screen along with the computer's answer. The following program clears the screen, allows you to input a number, clears the screen again, and then prints only the answer. It is the same program as before, with one additional line.

```
10  CLS
20  INPUT A
25  CLS
30  X = A * 2
40  PRINT X
50  END
```

Line 25 includes the extra CLS statement, which removes your input from the screen but not from the computer's memory. As soon as you type your number, it will be displayed on the screen, but when you press the Enter key, execution begins again and line 25 clears the screen once more, leaving a clean surface for the computer to output the value of X.

One problem with this program lies in the fact that it can only output one answer before it terminates execution, so you have to run the program again to allow for a different input and another answer. This is where an endless loop might be handy. To set up an endless loop, let's get rid of line 50, which contains the END statement, and replace it with:

```
50  GOTO 20
```

The program now makes an endless loop. Run the program now. As before, you will see the question mark, so type in a number. When you press Enter, the screen will clear again and the answer will be displayed. However, just beneath this answer you will see another question mark. This means that the program has done its job and branched back to line 20. You can now input another number. You can keep on doing this as long as you want. When you wish to terminate the program, simply press the CTRL key and the BREAK key. This brings about a manual halt.

This simple program is easy to understand, but it still lacks some of the refinements that a high-level computer like the PC-6300 is capable of providing. Wouldn't it be nice if the computer could tell us what it was looking for when it reached an INPUT statement? Fortunately, the INPUT statement can be used much like a PRINT statement. Just before execution is halted, a prompt or message may be printed on the screen to tell the user what the computer is looking for. The following program demonstrates:

```
10  CLS
20  INPUT "TYPE IN ANY NUMBER";A
30  CLS
40  X = A * 2
50  PRINT X
60  GOTO 20
```

This is the same program as before, except that an instruction or prompt has been included in quotation marks following the INPUT statement line but preceding the variable designator. When line 20 is executed, the message in quotation marks is displayed before execution is halted during the wait for keyboard input. A semicolon is used here to separate the variable from the quoted phrase. The semicolon at this point has a similar effect as that when used with the PRINT statement. In this case, it means that the value you type in for A will be displayed to the right of the prompt message.

You may also use a comma in place of the semicolon. If this is done, your keyboard input will be displayed on the line below the one which the prompt message occupies. Any time a quoted phrase is used with an INPUT statement, it is always necessary to separate it from its variable with a semicolon or comma. Also, you must use a variable following the prompt or a syntax error message will occur.

Using the INPUT statement in this manner speeds up the time it takes to input a BASIC program. Line 20 could also be replaced with two lines, such as:

```
20  PRINT "TYPE IN ANY NUMBER"
25  INPUT A
```

This is the equivalent of line 20 in the above program, but here, two lines have been used. The PRINT statement is used to display the prompt, whereas the standard INPUT statement is used in line 25 to read your keyboard input. This is somewhat wasteful, since the more program lines you have the more memory is required for storage, and it often takes longer for the program to execute.

Let's expand this program one more time to take care of one little problem that crops up when an endless loop is programmed. You always have to halt execution manually. The following program includes what might be called an exitable endless loop.

```
10  CLS
20  INPUT "TYPE IN ANY NUMBER";A
25  IF A = 333 THEN END
30  CLS
40  X = A * 2
50  PRINT X
60  GOTO 20
```

This is the same program as before, except for line 25. Here, an IF-THEN statement has been used to end the program if variable A is equal to 333. You will remember that this variable represents the keyboard input. This program will continue to allow you to input numeric values until you type in the exit sequence. Run the program through as many cycles as you wish, and when you wish to exit, simply type in 333 and the program is terminated. A prearranged signal value such as this is often called a "flag." (It would not be convenient to use a FOR-NEXT loop in this situation, since the programmer has no way of knowing how many numbers the user might wish to pass through this program.) Using a flag is not a great deal simpler than using the manual halt, but it reflects good programming practices. In such a situation, you might wish to alter the prompt in line 20 to:

```
20  INPUT "TYPE IN ANY NUMBER
TYPE 333 TO EXIT";A
```

This lets the user know exactly what is expected and what to do if an exit is desired.

Since we already know that variables may be of two types, numeric and string, we might assume that the INPUT statement may be used to accept either type of variable from the keyboard, and this is certainly the case.

```
10 CLS
20 INPUT "TYPE IN ANY WORD";A$
30 CLS
40 FOR X = 1 TO 5
50 PRINT A$
60 NEXT X
```

This program is used to reprint any string input five times on the screen. Though the prompt tells you to type in any word, you may also type in numbers or combinations of letters and numbers. Remember, a string variable will accept almost anything. (The lone exception is the comma. A string variable can contain commas only in special circumstances, so it is usually best to avoid them.) Here's how the program works. Line 10 clears the screen. Line 20 halts execution (after the prompt is printed) until something is input from the keyboard. When you press Enter the screen is cleared again, and a FOR-NEXT loop is entered at line 40. This loop counts from 1 to 5, and will print the value of A$ five times. Run this program and type in anything you wish, as long as it doesn't contain a comma. If you simply press Enter, nothing is displayed, because A$ will be equal to "", or nothing at all.

The INPUT statement requesting a string variable is often used simply to halt program execution in order to allow on-screen information to be displayed for as long as the user desires. Take the following program, for example:

```
10 CLS
20 A$ = "HELLO"
30 B$ = "GOODBYE"
40 PRINT A$
50 CLS
60 PRINT B$
```

Type it in as shown and then RUN it. A$ and B$ are equal to HELLO and GOODBYE respectively. Line 40 prints HELLO on the screen. We then want to clear this message from the screen (line 50) andthen print GOODBYE. To a beginning programmer, it may look good in principle, and indeed, the program works just like I've described it. However, when you run it, you will see an immediate problem. The computer processes so rapidly that you don't really see HELLO. As soon as it's printed, the CLS statement in line 50 clears the screen, so all you really see is GOODBYE. In such a situation an INPUT statement becomes very handy. We're not looking for any specific information from the keyboard, but it's necessary to halt execution so the user can determine when to start again. The END statement won't work here, because that would stop execution altogether and force us to re-run the entire program—only to end up with the same problem.

This situation can be solved very easily by adding a line to the program. Add:

```
45 INPUT A$
```

Now run the program again. You will now see the word HELLO clearly displayed on the screen. There will also be a question mark below it indicating that the computer is looking for input. Instead of typing in any characters, simply hit the Enter key when you're ready to re-start execution. Now, line 50 clears the screen and GOODBYE is printed. You might also include a prompt with the INPUT statement, such as "PRESS ENTER TO CONTINUE," which is often the case in many types of programs that include user instructions.

USING VARIABLES

Since our discussion is dealing more and more with variables, we should discuss their structure in a little more detail. First, you cannot use all of the characters on the keyboard as part of a string variable name. You may use any number and any letter (upper- or lowercase), and you may also use the at sign (@) and the underline character (_). You could use a variable name such as:

```
@437KP$ = "HELLO"
```

You could not use:

```
,,73%W$ = "HELLO"
```

because commas are not allowed. CAR$ = "HELLO" is fine. CAR$T$ = "HELLO" is not. The dollar sign correctly appears at the end of the name, but another dollar sign is included in the name.

When naming numeric variables, you cannot use any of the statements, functions, and commands (called *keywords*) found in GWBASIC. For example, LISST = 1 is fine, because LISST is not a statement, function, or command. LIST = 1 is unacceptable and will result in an error message, because this word is used in GWBASIC. With string variables, however, LET LIST$ = "HELLO" is fine, because LIST$ is not a keyword. However, there *are* a few statements in GWBASIC that do end with a dollar sign (CHR$, TIME$, MID$, RIGHT$, LEFT$, and DATE$) and cannot be used as string variable names.

ASSIGNMENTS

Computers are mathematical beasts, and writing computer programs often involves coming up with a mathematical formula that will cause the computer to do the desired job. (This is what we've been doing throughout this chapter.) In one example, the keyboard input number was doubled. The formula for doubling any number is: X = A * 2, where A is the number to be doubled and X is assigned the value of 2 times A. This is a simple formula and one which most of us can do in our heads, provided the value of A is not terribly high or complex.

Many applications that involve calculating your grocery bill, complex trigonometry, or even figuring the size of a sail for a sloop have been developed to the point where printed formulas are already available. Electronics is another field where this is apparent. Committing such applications to the computer is a fairly simple task, and even highly complex formulas can be quickly input to the computer simply by typing them in as they appear in reference books.

As an example, let's take the formula $P = I^2R$, which is used in electronics to figure power in watts. Here, I stands for current in amperes and R stands for resistance in ohms. To write a program that would calculate power (P) based upon current (I) and resistance (R), all we have to do is make the proper assignments to variables and we're home free. Here is the previous formula written as a computer program:

```
10 CLS
20 INPUT "CURRENT";I
30 CLS
40 INPUT "RESISTANCE";R
50 P = (I^2)*R
60 PRINT P
```

The formula in line 50 is the same one presented in the discussion. Parentheses are used to indicate that I is to be raised to the second power before it is multiplied by R. This differentiates it from I raised to the power of the product of 2 times R.

This is a simple formula, but even those that are half a page long can be typed in almost exactly as they appear in the reference books. If you have to work these formulas on paper, it can take some time. But once the computer has the formula in a properly written program, the answer is available almost instantly. Once the computer has the formula, all it needs is values for the variables.

The program could also have been written without the user input capability, by omitting the INPUT statements and substituting actual values for I and R. Line 50 might then read:

```
50 P = (4^2) * 5
```

where current is equal to 4 amperes and resistance to 5 ohms.

Surely you must be thinking that anyone who would be working with such a formula would already know that 4 squared is 16 and that 5 times 16 is 80. This is true, but let I be equal to 4.22896 and R equal 5.14398. Do that one in your head! This is the beautiful thing about computers. They don't think in terms of "hard" numbers and "easy" numbers. All numbers are separate entities to them and a long one is really no harder than a short one.

READ/DATA STATEMENTS

A very useful pair of statements in BASIC are READ and DATA. DATA statements contain what are often called *data elements*. This is the DATA statement's sole purpose, simply to hold elements. The READ statement pulls items from the DATA statement lines on a sequential basis. This means that the first item is read first, the second is read second, and so on. The following program demonstrates the use of these statements.

```
10  CLS
20  INPUT "PRESS ENTER TO READ A
DATA ELEMENTS";A$
30  CLS
40  READ X
50  PRINT X
60  GOTO 20
70  DATA 14,28,32,64,100
```

Run the program. Each time you press Enter in response to the prompt, a DATA element will be displayed on the screen. The data element is always displayed in the upper left-hand corner because of the CLS statement in line 30. Notice that each time you press Enter, the next data element is read in line 40 and displayed in line 50. The numbers themselves are contained in the DATA statement lines. After one item is read, it is not read again.

If you've gotten a bit ahead of me, you've probably already discovered an error message. This occurred on the pass immediately after the last data element, the number 100, was read. The "out of DATA in 40" error message indicates that all of the data items have been read, but the READ statement in line 40 is still looking for more. Often, READ statements are placed in FOR-NEXT loops. This means you must make certain the number of data elements equal the passes in the loop.

We can correct this error message situation by using a new statement. Add the following line:

```
55  IF X = 100 THEN RESTORE
```

The RESTORE statement tells the READ statement to go back to the first element in the DATA statement line. If you'll think of the READ statement as counting through the DATA elements, you can think of RESTORE as resetting this count to 1. Run the program again. You can press Enter as often as you wish, and you will get no error message. When the last number in the DATA statement line has been read, on the next pass the first number will come up again. If you want the program to end as soon as the last DATA element is read, change line 55 to

```
55  IF X = 100 THEN END
```

All we're doing is setting up an IF-THEN statement to see if the last element in the DATA statement line is being displayed. Since there are no more DATA elements, the program terminates.

You will notice here that the variable following the READ statement is a numeric type—an indication that the DATA statement line contains only numeric values or numbers. However, DATA-READ statements can also be used to store and access string information, as in the following program:

```
10  CLS
20  INPUT "PRESS ENTER TO READ A
DATA ELEMENT";A$
30  CLS
40  READ X$
50  PRINT X$
60  IF X$="GOODBYE" THEN END
70  GOTO 20
80  DATA HELLO,HOW,ARE,YOU,
GOODBYE
```

Run this program and you will see the words displayed on the screen in exactly the same way the numbers were.

FUNCTIONS

In BASIC, a function may be thought of as a statement that is really a mathematical formula preprogrammed into the computer, and used to effect mathematical operations on numeric and/or string variables. This definition will not hold true for every function found in GWBASIC, but it will for most. Functions are very powerful tools. Some will allow you to manipulate string variables, while others will be used with numeric variables to return geometric, trigonometric, and other types of mathematical numbers based upon the value of the variable.

The LEN Function

The LEN function in GWBASIC stands for length, and will return the length of a string variable. When I say return the length, I mean that it will assign to a numeric variable the number of characters in the string. The following program will demonstrate the LEN function and may be input to your computer:

```
10  A$ = "HELLO"
20  A = LEN(A$)
30  PRINT X
```

The string variable whose length is being measured in line 20 is enclosed in parentheses. In this example, A$ is equal to the word HELLO, which we can easily see contains five characters. In line 20, the numeric variable X will assume the value 5, the length of A$, and the number 5 will appear on the screen. One must always remember that to the computer, a space is just as much a character as any other letter, number, or symbol on the keyboard. Therefore, if A$ were equal to "HELLO BOB", the space between the O and the B would also be counted as a character, and X would be equal to 9.

The program following shows another way to write a program using the LEN function. This one uses the INPUT statement to read from the keyboard, and assign this input to the string variable A$. CLS statements have been added to make for a clean screen display. This program will tell you the length of any words, numbers, or characters you input via the keyboard.

```
10  CLS
20  INPUT A$
30  CLS
40  X = LEN(A$)
50  PRINT X
```

When this program is run, the screen will be cleared and you will see a question mark (?) prompt telling you that the computer is expecting some user input. You can type in anything at this point. When you press Enter, the screen will be cleared again and the value of X, which is the number of characters supplied, will be printed on the screen.

The program below is set up on an exitable endless loop. It does the same thing as the program above, but allows you to continue to input data from the keyboard. When you wish to exit the program, all you have to do is press the Enter key.

```
10  CLS
20  INPUT"TYPE IN ANY PHRASE TO BE
MEASURED";A$
30  CLS
40  X = LEN(A$)
50  IF X = 0 THEN END
60  PRINT X
70  GOTO 20
```

Line 20 allows you to input the characters to be represented by A$. A prompt is included with the INPUT statement to instruct the user as to what is expected. Line 40 uses the LEN function to assign X the value of the number of characters in A$. Line 50 contains our exit routine. It simply says if the value of X is zero (no characters in the string) then END the program. When you wish to exit a program, simply supply a null string by pressing Enter, in response to the INPUT statement prompt.

It may be difficult now for you to imagine any useful purpose for the LEN function, but as your programming experience increases, you will find yourself using it more and more. One possible use is a typing test program. Here, the LEN function would indicate the number of characters typed. By knowing the number of characters input and the time it took to input them, typing speed can be quickly ascertained. The LEN function is also useful in some types of programs as to check for the proper user input. For example, a game program might require the user to input a word that is no longer than 5 characters. The LEN function can be used to test the length of the user's input and generate an error message if the input exceeds the specified length.

The INT Function

The INT function stands for integer. It is used only with numeric variables and returns the integer equivalent of this variable. An integer is simply a whole number (positive or negative). The numbers 3, 5, 6, 9, and 10 are integers, whereas 3.5, 9.3, 10.1, etc., are not. The INT function simply truncates or lops off the decimal portion of a number. (The number 0 is also an integer, so the integer equivalent of the number 0.003 is 0.) The following program demonstrates the use of the INT function.

```
10  A = 10.5
20  X = INT(A)
30  PRINT X
```

When this program is run, the computer screen will display the number 10, which is the integer equivalent of the real number 10.5. That was simple enough!

One must always remember that the INT function technically does not *round* numbers, though you could say that the INT function always rounds *down*. It will always return the largest integer that is less than or equal to A in the above program. This seems fairly simple until one starts dealing with negative numbers. If A were equal to –3.6, the INT function would return –4. One might expect that –3.6 as an integer would end up as –3, until one stops to think that –3 is a larger number than –3.6. In this example, the INT function has still rounded down.

The RND Function

The RND function stands for *random*. You can think of RND as a variable that can be equal to any value between 0 and 1, not including these two numbers themselves. The RND function works in close conjunction with the RANDOMIZE statement. RANDOMIZE simply "shuffles" the random number generator, and RND is the random number output. The RND function is extremely useful in programming games of chance on the PC-6300 because one never really knows what number it's going to represent. Whenever you use RND in a program, you should also use the RANDOMIZE statement (at the beginning of the program). If you don't, the output from RND will be the same during a given program run.

Between RANDOMIZE and RND, you can generate what can be classified as truly random numbers, at least to us. From a technical standpoint, these numbers are not really chosen at random (like drawing a number out of a hat). The computer actually has a mathematical formula to guide it in making its selection, but it is one that is too complex for most of us to anticipate. (This is technically known as pseudo-random number generation.) The program below will demonstrate the RANDOMIZE statement and RND function.

```
10  RANDOMIZE
20  FOR X = 1 TO 10
30  PRINT RND
40  NEXT X
```

When this program is run, the computer will generate an automatic prompt, due to the RANDOMIZE statement in line 10, which will read "Random number seed (–32768 to 32767)?" The computer is telling you that it wants you to type in any number between –32768 and +32767 for it to use in the randomizing formula. After you've typed in this number, simply press Enter and the screen will then display 10 random numbers. RUN the program again and type in a different number, and the random numbers will be completely different because of the new seed number.

The RND function is rarely used by itself, as shown in line 30. Most often, we use RND as a multiplier. This can be demonstrated by writing a simple game that might be thought of as a computerized version of flipping a coin.

First, let's think of the possibilities which occur when a coin is flipped. From a practical standpoint, only one of two can occur: the coin can either land with heads up or tails up. Since we have two possibilities, we need the computer to output one of two possible numbers at random. One number would represent heads, while the other would represent tails. In other words, we want to make sure that only two numbers are possible. Let's start with the program below.

```
10  CLS
20  RANDOMIZE
30  CLS
40  X = RND * 2
50  INPUT "PRESS ENTER TO FLIP
COIN";A$
60  PRINT X
70  GOTO 40
```

This program is a good start for a coin flip game, but it will not do the whole job yet. The RANDOMIZE statement is used in line 20 to allow you to input a random seed number. In line 40, the numeric variable X is equal to RND times 2. The number 2 was chosen because we're looking for two possibilities, heads or tails. We'll let the number 1 represent heads and 2 tails. The INPUT statement in line 50 is used to give us some control over when the coin is flipped. Each time Enter is pressed, the value of X, which represents the flip, will be printed on the screen. The GOTO statement in line 70 branches back to line 40, where X is assigned another random number. Each time the RND function is executed, a different random number will occur.

Run this program and press Enter five or six times. You will note immediately that at no time do the numbers 1 or 2 appear on the screen. What you have is a series of numbers with long decimals. However, if you look closely you will see that all of these numbers fall into two categories; one group is equal to more than 1, while the other group is less than 1. The computer is on the right track.

Our goal is to write a program that will output either a 1 or a 2. Think back to the previous function that we discussed. The INT function will always return a number that is a whole number. Since the numbers 1 and 2 are indeed integers, we might well be able to use INT in our coin flip program. Change line 40 to:

```
40  X = INT(RND*2)
```

Now, the program will always output an integer. RUN it again and press Enter several times to flip the computerized coin. We're getting close now, because only two numbers are being output, although they are 0 and 1 rather than 1 and 2. This would probably suffice if we let 0 equal heads and 1 equal tails, but this was not the original intent of the program. We want the numbers 1 and 2. The purpose of this will become apparent shortly. What do we do?

The answer is simple. By adding 1 to each of the numbers already output, we will always arrive at one of two possible numbers, 1 and 2. That's our goal! Change line 40 to:

```
40  X = INT(RND*2) + 1
```

Make sure the +1 appears outside of the parentheses. Now run the program again. Eureka! The computer is now outputting 1s and 2s as originally planned. Let's go one step further and make the game more lifelike, using IF-THEN statements. Let's change line 60 completely and add another line to be numbered 65:

```
60  IF X = 1 THEN PRINT "HEADS"
65  IF X = 2 THEN PRINT "TAILS"
```

Run the program again and you will see that our coin flip game has been completed. Congratulations! You have just written your first game program! It wasn't all that difficult, was it?

Now, to explain emphasis on tailoring this program to output the numbers 1 and 2 to represent heads and tails let's see why this may be important for future programs. Some games of chance use devices that normally output numbers. The first thing that comes to mind is any dice game. Each die contains numbers from 1 to 6 on its sides. In the coin flip game, we could have stayed with the starter program, which simply output decimal numbers that were either less than or more than 1. We could have used IF-THEN statements to tell the computer to print heads if X was less than 1 or tails of X was more than 1. We can't do this with dice. We must have whole numbers that range from 1 to 6.

We will now write a simple dice program using a computer version of one die. You will remember that we used the number 2 in the coin flip program to be multiplied by RND. This number was chosen because we were looking for 2 possible conditions. In a dice program, we are looking for 6 possible conditions. Therefore, we will replace 2 with a 6. The finished program is shown below.

```
10  CLS
20  RANDOMIZE
30  X = INT(RND*6) +1
40  INPUT "PRESS ENTER TO ROLL THE
DIE";A$
50  PRINT X
60  GOTO 30
```

Hey! This program is even less complex than the finished coin flip program, because we don't have to convert the computer's numbers into on-screen words. Run the program. After you've input the seed number you will be greeted with a prompt, and each time you press Enter a number ranging from 1 to 6 will appear on the screen. You never know which number is going to come up, because the computer is outputting it based upon the random number generator. This is a true computer representation of a dice game.

Certainly, you might complain that only one die is incorporated. Most of the work has already been done, and to add another die it's only necessary to modify one line in the original program and add one more. Change line 60 to:

```
60  PRINT X;Y
```

and add line 45:

```
45  Y = INT(RND*6) + 1
```

By adding another line (45), we have inserted another RND function and assigned to Y the output of this line. It is possible for both X and Y to be equal, representing a double. It is even more possible for the two to be unequal. Run the program and you will see that sometimes you get a double, and sometimes you don't. Let's add another line to make the program even more interesting.

```
65  IF X = Y THEN PRINT "DOUBLE"
```

The IF-THEN statement checks for a condition of X being equal to Y. If this is true this same line will print the word "DOUBLE" on the screen. Add another line.

```
66  IF X = 1 AND Y = 1 THEN
    PRINT"SNAKE EYES"
```

This IF-THEN statement checks for a condition of both X and Y being equal to 1. This is the familiar condition known as "snake eyes" in some dice games. This message is displayed along with the previous message of "DOUBLES," since both conditions are true. Variables X and Y are equal to each other *and* both are equal to 1.

Line 66 is quite a bit different than the previous examples of IF-THEN statements. In this example, the word AND is called a *logical operator.* (Sometimes you will also see an OR in its stead.) This is very simple to understand and the line is almost self-explanatory. A condition is true only if the values on both sides of the AND are true. In this case, if X is equal to 1 *and* if Y is equal to 1, then do whatever follows the THEN statement. With this in mind, you should be able to insert your own program line (67) which will test for the condition of "boxcars" (both dies being equal to 6).

To review, remember that the RANDOMIZE statement simply shuffles the random number generator based on the number you input when the program is first run. It is necessary to do this only once during the program run and not each time the RND function is used. RND may be thought of as a number lying somewhere between 0 and 1. It will never be equal to 0 nor to 1. It may be used like any other variable. In most cases, it is used as a multiplier, but it can also be added to a real number, subtracted from a real number, or as a divisor or dividend. The INT function may be successfully used to cause all random numbers output to the screen to be whole numbers.

The CINT Function

The CINT function is very much like the INT function in that it will always return an integer. However, CINT performs this conversion process by rounding. CINT will round up or down depending on the fractional portion. Any fractional value of 0.5 or higher will cause CINT to round up, and any value lower than 0.5 will be rounded down. The following program demonstrates the CINT function.

```
10 X = 24.3
20 Y = 24.6
30 A = CINT(X)
40 B = CINT(Y)
50 PRINT A
60 PRINT B
```

When this program is run, the numbers 24 and 25 will appear on the screen. The variable A represents the CINT value of 24.3. Since .3 is less than .5, CINT rounds down. Variable B represents the CINT value of 24.6. Since .6 is more than .5, CINT rounds up. CINT is often used in place of INT when performing calculations with monetary values.

Other String Functions

In GWBASIC, there are several powerful functions that are dedicated to the handling of string variables. These are used heavily in word processing programs, and also crop up often in many other types of programs. We have already met LEN$, and these new ones are not terribly difficult to understand if you take them a step at a time.

LEFT$. The LEFT$ function is used to pull a sequence of characters from a string, starting with the left-hand side of that string. The following program explains it best.

```
10 A$ = "HELLO"
20 X$ = LEFT$(A$,2)
30 PRINT X$
```

This program assigns the word HELLO to the string variable A$. Line 20 assigns a portion of A$ dictated by the LEFT$ function to the string variable X$. Line 30 prints the value of X$ on the screen. RUN this program, and you will see that the screen displays HE. Look at line 20. The number 2 inside the parentheses tells the computer to assign to the string variable X$ the two LEFT-most characters in A$. If you change the 2 to 1, then only an H will be printed because this is the first character from the left in A$. Change this number to 4 and see what happens. The LEFT$ function is very useful if you wish to retrieve only a portion of a string.

RIGHT$. The RIGHT$ function works just like the LEFT$, except it starts searching at the right of the string. The following program demonstrates.

```
10 A$ = "HELLO"
20 X$ = RIGHT$(A$,3)
30 PRINT X$
```

When this program is run, the computer will display LLO, since these are the rightmost three characters in A$. It's hard to generate an error message with either of these two functions. If you specify more letters than are contained in the string, the computer simply returns the entire string.

MID$. While LEFT$ and RIGHT$ are very useful functions, they always force us to accept all the characters in a sequence from the left, or to the right. We cannot go into a string and pull out a middle section, for instance. However, GWBASIC also gives us MID$, which will allow us to do just that. It works very much like LEFT$ or RIGHT$, except we have to give it two numbers, one to indicate the position within the string to start its search and one to indicate the end of search. All numbers are given in relation to the left of the string. The following program demonstrates the use of MID$.

```
10  A$= "HELLO"
20  X$= MID$(A$,2,3)
30  PRINT X$
```

When this program is run, the computer will display the value of X$, which is ELL. We told the computer to search the string starting with the second character to end three characters later. The first character, then, is the E, the second is L, as is the third. Do not be confused. MID$ does not count three characters after the second character, but three characters including the second character in this example. With the MID$ function, we can easily pull out any portion of a string desired.

TIME$. The TIME$ function returns the current time, which is maintained by the computer's internal clock. It is necessary to assign a value to TIME$ at the beginning of the program. Once this time has been inserted, the PC-6300 will automatically update it on a second-by-second basis. Let's start with the following program.

```
10  X$ = "09:36:22"
20  TIME$ = X$
30  PRINT TIME$
```

Line 20 could also have been written as TIME$ = "09:36:22", which would have allowed us to delete line 10. Either method will work. When you run this program, nothing spectacular happens. The value of X$ will be printed on the screen. Now, wait a few seconds and in direct mode (remember—no line number), type PRINT TIME$. When you press Enter, the value of TIME$ will be displayed on the screen, but you will notice that TIME$ is no longer equal to the value you assigned it in the previous program. It is now equal to a value that reflects the seconds that have passed since the original program was run. The clock is now keeping time based upon the value that was inserted by the program above.

The following program will allow you to input the current time and then see it constantly displayed.

```
10  CLS
20  INPUT "CURRENT TIME IN HOURS,
MINUTES, ANDSECONDS";X$
30  TIME$ = X$
40  CLS
50  PRINT TIME$
60  GOTO 40
```

The INPUT statement in line 20 allows you to supply your current clock time. Your input is assigned to the string variable X$, and in line 30 the TIME$ function is initially assigned the value of X$. The screen is then cleared, and the value of TIME$ is displayed on the screen. The GOTO statement in line 60 branches back to line 40, where the screen is cleared and TIME$ is printed again. This new TIME$ value will be the updated version, which has been controlled by the PC-6300's clock. I hope you have not run the program yet, since in a few instances I have set up programs which are incorrect, or do not operate in the proper manner to illustrate a point. This is one such example. RUN the program at this time. You will notice a flickering and flashing and the time is displayed in the upper lefthand corner, but it's difficult to see. This is because the screen is being written and cleared so fast (by the loop set up between lines 40 and 60) that viewing is almost impossible.

Suppose we don't clear the screen each time the correct time is printed. What happens then? Test this by changing line 60 to:

```
60  GOTO 50
```

Now, the CLS statement is taken out of the loop, so the screen will not be cleared before each printing. Run the program again. Goodness! The time is being displayed, but the numbers span the screen from top to bottom. Would a semicolon help us at this point? Try inserting one at the end of line 50. When you run this program, you will see that this is no help whatsoever. So how do you display time at one spot on the screen?

The answer is simple, and it involves a new statement that is extremely useful in outputting information to the screen in a pleasing format. In GWBASIC, this is the LOCATE statement. With LOCATE, we can specify exactly where we want the computer to print information on the screen. If there's something already at that position, the computer simply writes over it. Remove the semicolon at the end of line 50, then change line 60 and insert the following line as shown.

```
    45  LOCATE 14,38
    60  GOTO 45
```

Now run the program. That did it! After you input the correct time and press Enter, that time is displayed at the center of the screen and you can actually see the seconds ticking away on the electronic clock. This is due to the fact that the LOCATE statement is causing the PRINT TIME$ statement which follows it to always display information at location 14,38 on the screen. There's your electronic clock, and it will be far more accurate than most clocks you've ever had.

UNDERSTANDING THE SCREEN

The LOCATE statement in GWBASIC accesses a certain screen location, which can be used in conjunction with any statement that writes information to the screen. When a LOCATE statement is used, it will determine the exact position for the next write—the very next, but not all others that follow. In the previous program, the LOCATE statement in line 45 was used by the PRINT statement in line 50 to position the value of TIME$ near the center of the screen. Since the LOCATE statement was made a part of the loop between lines 45 and 60, before line 50 was executed, the LOCATE statement had already determined the position where TIME$ would be written. Since this position never changed, the new value of TIME$ was written over the old value.

The numbers immediately to the right of the LOCATE statement determine the position on the screen it accesses. The PC-6300 divides your screen into 2,000 different points. Figure 2-1 shows the screen layout in text mode, which is what we're dealing with right now. The screen consists of 80 character positions from left to right, called columns. There are also 25 rows or lines on the screen. Thus, 25 rows times 80 columns gives you 2,000 positions. While the AT&T PC-6300 has 25 rows, normally BASIC will not print on line 25. (It can be done, but it's a pain and can cause confusion.) Think of the machine as having a practical display of 80 columns by 24 rows—which will be assumed for the remainder of this discussion.

The LOCATE statement uses row and column numbers to indicate the point at which text is to be written on the screen. LOCATE 1, 1 tells the computer to start writing text at row 1, column 1. Figure 2-2 shows this position. LOCATE 2, 1 indicates a write to be performed starting at the first column in the second row. We can generalize this as LOCATE *row, column*, where the row number

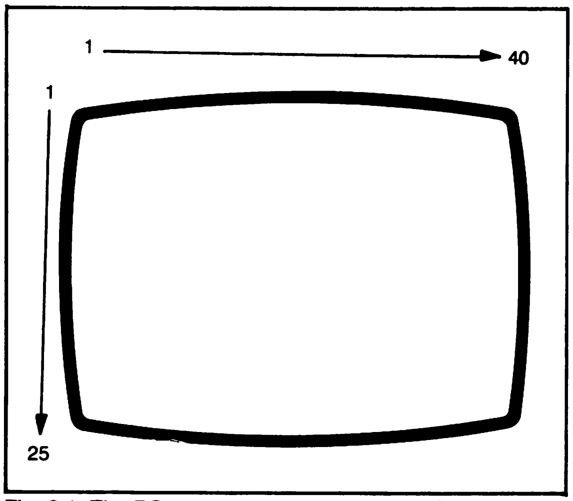


Fig. 2-1. The PC-6300 text screen consists of 25 rows and 40 columns.

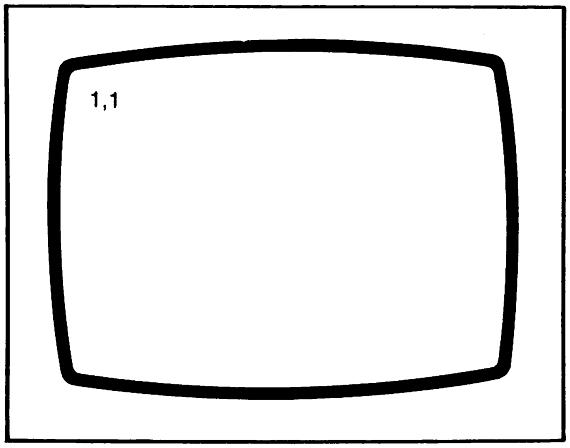


Fig. 2-2. Coordinates 1, 1 identify the upper left hand corner of the text screen, or row 1, column 1.

indicates the vertical positioning, and the column number represents the horizontal. In the clock program, I chose 14 as the row number since it lies just below the middle of the screen, while column 38 is just to the left of center. Why? Because it is often better to choose an optical center (from the viewer's standpoint) rather than exact center (based upon screen coordinates alone.) You can experiment with the coordinates a bit to suit yourself.

ANOTHER BRANCH STATEMENT

Early in this chapter we discussed the GOTO statement, a branch statement that tells the computer where to go within the program, or, more specifically, which lines to execute and which to skip over. Branch statements allow us to use previously written program segments to perform job points in the program past their original location.

There is another, extremely useful branch statement in GWBASIC that may be thought of as a GOTO statement with a user-programmed return. It's called GOSUB and it is more often referred to as GOSUB and RETURN. GOSUB stands for go to subroutine. A subroutine is like a miniature program unto itself, within the larger structure. Like GOTO, GOSUB refers to a line number which tells the computer where to branch. However, a subroutine accessed by a GOSUB always ends with RETURN. This tells the computer to go back to a point *one line past* the line which sent it to the subroutine. The following program demonstrates the use of these statements.

```
10 CLS
20 INPUT "TYPE IN ANY NUMBER TO
BE DOUBLED";X
30 CLS
40 GOSUB 70
50 PRINT Y
60 END
70 Y = X * 2
80 RETURN
```

While this program makes somewhat trivial use of GOSUB and RETURN, it does effectively demonstrate their use. Line 10 clears the screen. Line 20 instructs the user to type in any number to be doubled, which input is assigned to the numeric variable X. The GOSUB statement found in line 40 tells the computer to go to a subroutine that begins at program line 70. It assigns to Y the value of X times 2. Line 80 terminates the subroutine with a RETURN statement.

The RETURN statement tells the computer to go back to the line which follows the one containing the GOSUB statement. Since the GOSUB statement is found on line 40, RETURN causes line 50 to be executed next. Line 50 displays the value of Y (2 times X) on the screen. Note the END statement in line 60. You will recall that this statement was not needed in our other programs, since BASIC will automatically stop execution when there are no other lines to read. In this program, however, if you omit the END statement BASIC would automatically go on and execute the remaining lines, which are our subroutine lines 70 and 80. But remember, a subroutine is a small program unto itself, which is not to be executed unless specifically told to do so by a GOSUB statement.

As a test, let's remove the END statement in line 60 altogether. In direct mode, simply type 60 and hit Enter. (If you can list your program at this point, you will see that line 60 no longer exists.) Run the program again. You can still input a value and the GOSUB will still access the subroutine. The RETURN statement in line 80 causes the computer to go back to line 50, where the value of Y is printed.

However, you are now faced with an error message: the computer is telling you that it encountered a RETURN without GOSUB. Since the END statement was omitted, after line 50 was executed the computer executed lines 70 and 80, where the computer became distressed. It read a RETURN statement and said, "Return where? I'm not supposed to be here because a GOSUB hasn't instructed me to do so." It worked fine during the first entry to the subroutine, because the GOSUB in line 40 sent it there. But when it got there the second time, it had no GOSUB instruction and told you so through the error message. In this case the END statement is crucial to proper execution of the program.

MULTIPLE-STATEMENT LINES

This portion of the text will tell you about some ways you can make your programs more economical. All previous programs have used only one statement, function, or assignment per program line; with the PC-6300 it is quite easy to include several statements, functions, or assignments on a single program line. Take the following program, for example:

```
10  CLS
20  C=10
30  Y=20
40  Z=X*Y
50  PRINT Z
```

This is just like our previous examples. However, it can be rewritten as follows:

```
10  CLS
20  X=10:Y=20:Z=X*Y
30  PRINT Z
```

Here, line 20 includes all the assignments made in lines 20 through 40 in the previous program. Notice that a colon is used to separate each of the assignments. The PC-6300 sees this program in exactly the same way as the previous one. We could even go one step further and write the program as:

```
10  CLS:X=10:Y=:Z=X*Y:PRINT Z
```

Here, the entire program is contained in a single program line. Notice that colons separate each individual statement on the line. If you omit a colon, you will get an error message. You can use as many as 255 characters per line, but if you go past this point the computer automatically knocks off the additional characters.

For the sake of clear programming, you will not often see a tremendously large number of statements committed to any one line. From a memory savings standpoint there is an advantage in using as few lines as possible, because each new line number takes up additional memory. The last example requires slightly less memory storage than the first. Sometimes, however, clear programming can be better effected by committing only a group of similar statements or assignments to a single program line. The second example shown is getting close to this; line 20 made all assignments to X, Y, and Z. However, CLS is put on a separate line, as was PRINT Z. From the standpoint of program clarity it is sometimes worth sacrificing a few bytes of memory space.

LOGICAL OPERATORS

Logical operators perform special operations using numeric values. The ones we will be concerned with here are OR and AND. In text mode programming, we often use logical operators in IF-THEN statements to bring about special branches. The following program is a good example:

```
10  CLS
20  INPUT "TYPE ANY NUMBER FROM
0 TO 10";X
30  INPUT "TYPE ANY NUMBER FROM
11 TO 20";Y
40  IF X=5 AND Y=18 THEN PRINT
```

```
"THOSE WERE THE NUMBERS I WAS
LOOKING FOR"
50 END
```

Line 40 tells the computer to print the quoted phrase only when X is equal to 5 and Y is equal to 18. If X is equal to 5 and Y is equal to a number other than 18, the phrase will not be printed. Conversely, if X is not equal to 5, the phrase will not be printed even if Y is equal to 18. When the AND operator is used, both conditions must be true. In this case, X must be equal to 5 and Y must be equal to 18 before the phrase will be printed. You can go further if you want. After adding a few more INPUT statement lines, line 40 might read:

```
40 IF X=5 AND Y=18 AND Z=40 AND
ZZ=55 THEN PRINT "THOSE WERE
THE NUMBERS I WAS LOOKING FOR"
```

Now let's discuss the OR operator. Change line 40 in the original program to:

```
40 IF X=5 OR Y=18 THEN PRINT
"BINGO!"
```

Here, line 40 tells the computer to print BINGO if X is equal to 5 *or* Y is equal to 18. If X is not equal to 5, *but* Y is equal to 18, the word will be printed. As long as X is equal to 5 or Y is equal to 18, the word will be displayed. Only one of the tests within a logical OR operation must be true to bring about the proper result. If both are true, that's fine too.

You can also use OR and AND together. The following program demonstrates:

```
10 CLS
20 INPUT "TYPE ANY NUMBER FROM
0 TO 10";X
30 INPUT "TYPE ANY NUMBER FROM
11 TO 20";Y
40 IF X=5 AND Y=15 OR X=3 THEN
PRINT "BINGO!"
```

Here is the condition set up in line 40. If X is equal to 5 and Y is equal to 15 the computer will print BINGO. However, if X is not equal to 5 and Y is not equal to 15, but X is equal to 3, the word will also be printed. You can think of this as two IF-THEN tests on the same line. The first tells the computer to print the word if X is equal to 5 and Y is equal to 15. The second tells the computer to print the word if X is equal to 3.

RELATIONAL OPERATORS

A relational operator is a symbol that causes the computer to compare two values. We are already familiar with one of these relational operators. This is the equality sign (=). There are other relational signs as well. The inequality sign is used to state that two values are not equal and looks like this:

<> not equal to

While this is treated like a single symbol by the computer, typing it into the program requires two keystrokes, the comma and period keys in uppercase. The inequality sign is used just like the equal sign, as in:

```
A<>B
```

There are two other relational operators that we use quite often in BASIC, the "more than" and "less than" symbols shown below.

< less than
> more than

Here is the format in which they are used:

A<B	A is less than B
A>B	A is more than B

Just think of the wide side as being the larger side, and the pointed side as pointing to the variable that is the smaller.

Now, we can combine the less than/more than symbols and the equal sign, as follows.

A<=B	A is less than or equal to B
A>=B	A is more than or equal to B

We often use the relational operators to make programs foolproof for the user. The program below needs an input number of from 0 to 10. Line 30 makes sure that the number input is not less than 0.

```
10 CLS
20 INPUT "TYPE A NUMBER FROM
0-10";X
30 IF X<0 THEN PRINT "INVALID
NUMBER"
```

Line 30 says if X is less than 0 then print the phrase "INVALID NUMBER." We might also wish to make sure that the number input is not larger then 10. The program below shows how this is handled.

```
10 CLS
20 INPUT "TYPE A NUMBER FROM
0-10";X
30 IF X>10 THEN PRINT "INVALID
NUMBER"
```

Line 30 tells the computer to print the phrase if X is more than 10.

We don't need two programs to do this if we remember the previous discussion on logical operators. This program demonstrates the use of logical and relational operators:

```
10 CLS
20 INPUT "TYPE A NUMBER FROM
0-10";X
30 IF X<0 OR X>10 THEN PRINT "IN-
VALID NUMBER"
```

Line 30 tells the computer to print the phrase if X is less than 0 or if X is more than 10.

The following program also allows us to exclude zero, using the less than or equal to operator:

```
10 CLS
20 INPUT "TYPE A NUMBER FROM
1-10";X
30 IF X =0 THEN PRINT "INVALID
NUMBER"
```

Note that in line 20 we desire a number from 1 to 10. We could also have handled this by specifying that the phrase is to be printed if X is less than 1.

Some of the following chapters will present program listings that make heavy use of relational and logical operators, so be sure to reread any materials you're not quite clear on. The logical and relational operators allow us to make the most efficient use of programming space, and make the programs easier to understand once these operators are understood.

ARRAYS

An array is a group of values referenced by the same name. It may be thought of as a table. Most beginning programmers shy away from arrays because they feel they are extremely difficult to understand, but this is totally untrue. You will find that arrays are a tremendous help in all types of BASIC programming. For now, simply think of an array as a "tank" with a certain name which holds a large number of values. All of the values are contained within this tank in sequential numeric order. If the array is named A, then A(0) contains one value, A(1) contains another, A(2) yet another, and so on.

To set up an array, we use the DIM statement, which stands for dimension. This determines the size of the array, or more appropriately, the number of values it can contain. This is the format for DIM:

```
10 DIM A(10)
```

The DIM statement sets up an array named A which can hold 11 values. Why 11? Because the first element in the array will be automatically specified as A(0) by the computer. (Remember that 0 is a valid number.) You can simply remember that an array can hold one more element than its numeric designator would seem to indicate, for those of us who start counting with 1 rather than 0.

The following program demonstrates some of the workings of an array:

```
10 CLS
20 SCREEN 0
```

```
 30  DIM A(5)
 40  A(0)=10
 50  A(1)=20
 60  A(2)=30
 70  A(3)=40
 80  A(4)=50
 90  A(5)=60
100  PRINT A(2)
```

Here, the array was dimensioned to contain 6 elements. Lines 40 through 90 assign numbers to the various array positions. You will notice that we treat A(0) or any other element in A simply as another variable. Line 100 tells the computer to print the value of the element contained at A(2).

This may seem very awkward, so why not use standard variables here instead of an array? For small numbers of variables this is a good point, but suppose you need a hundred—or a thousand? You would soon run out of discrete names for your variables. There are also other reasons. The next program puts the array to more effective use.

```
10  CLS
20  SCREEN 0
30  DIM A(5)
40  FOR X=0 TO 5
50  A(X)=(X*10)+10
60  NEXT X
70  PRINT A(2)
```

This program fills the array with the same values as before, only it does it much faster. When you run the program, the screen will still display the value of 30, which has been assigned to A(2). Line 50 is the key here. The value of X is substituted for the array element number and, indeed, the value assigned the element is a function of the element number itself. In line 40, the FOR-NEXT loop assigns the variable X a value of from 0 to 5. On the first pass of the loop, A(X) in line 50 is really equal to A(0). On the second pass of the loop, it's equal to A(1). When the loop times out, it's equal to A(5). Line 50 assigns A(X) a value which is equal to ten times X plus the number 10. On the first pass, X is equal to 0, and ten times 0 is still 0. But 0 plus 10 is equal to 10. Therefore, A(X), or A(0) on this pass, is equal to 10. On the next pass A(X), or A(1) is equal to one times ten plus 10, or 20. It is not possible to easily assign a standard numeric variable in this manner. We can only do this with an array, where we can substitute a variable for the array element number.

The arrays discussed thus far are called numeric arrays. We can also have string arrays that work the same way. To designate a string array, we would use the format:

```
30  DIM A$(5)
```

This establishes a string array named A$ which will hold 6 elements. Assignments are made just as they would be with any string variable; i.e.,

```
31  A$(0)="HELLO"
32  A$(1)="GOODBYE"
33  A$(2)="HI"
```

and so on. The following program uses a FOR-NEXT loop and an INPUT statement contained within the loop to assign words or phrases to an array. The second part of the program reprints these words and phrases:

```
 10  CLS
 20  SCREEN 1
 30  DIM A$(5)
 40  FOR X=0 TO 5
 50  INPUT "TYPE IN ANY WORD";W$
 60  A$(X)=W$
 70  NEXT X
 80  CLS
 90  FOR X=0 TO 5
100  PRINT A$(X)
110  NEXT X
```

When this program is run, you will be asked to "TYPE IN ANY WORD." The first word is assigned to W$. Line 60 assigns to A$(X), which is also A$(0), the value contained in W$. The loop recycles, and you are again prompted to type in any word. Your new word is committed to W$, and it, in turn, is committed to A$(1) in line 60. This occurs during each of the six cycles of the loop.

After line 80 clears the screen, lines 90 through 110 display the contents of the array A$. The loop in line 90 counts X from 0 to 5 and line 100 prints the contents of A$(X) on the screen as X assumes each new value. Remember, the element *number* represented here by X is always a whole number (integer), and may be represented by a numeric variable—even though the array elements themselves might contain string values.

All of the arrays, numeric and string, discussed thus far are known as single dimensional arrays. You may think of them as a vertical table of values. The A(o) position is at the very top of this table, with A(1) below it, then A(2) below it, etc. We can also make arrays multi-dimensionals using the following format.

DIM A(5,2)

This tells the computer to set up an array that consists basically of rows and columns. These will be 6 rows, each containing 3 columns or elements on each row. Figure 2-3 shows how the element positions are numbered and how they might appear. With a multi-dimensional array, we can think in terms of how the text screen is set up. For instance, 1,1 represents the upper-lefthand corner, while 1,2 represents the second position on the top row. With a multi-dimensional array, the first element in the first line would be at position 0,0. The second element in the first line or row would be 0,1, followed by 0,2, etc. The first element in the second row would be 1,0 and then 1,1, 1,2, etc.

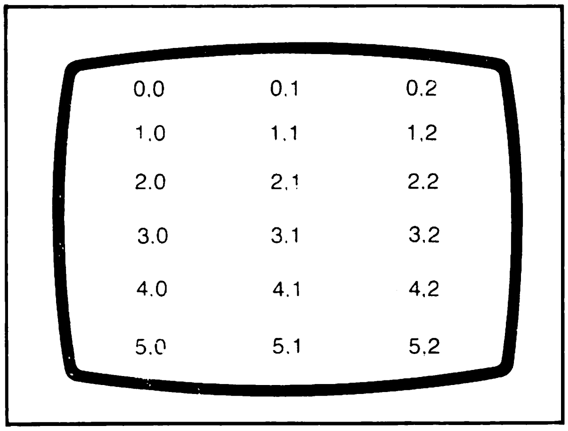


Fig. 2-3. The configuration or alignment of data elements in a multidimensional array.

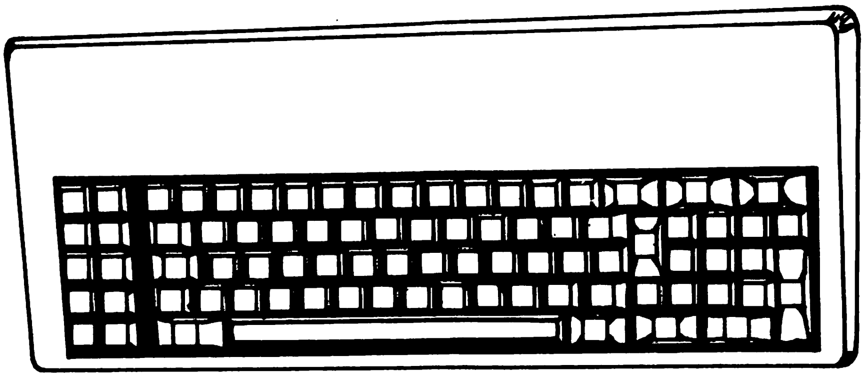
We will not delve further into multi-dimensional arrays at this time since they can be confusing—especially after you've just been introduced to a single-dimensional array. Try experimenting with the single-dimensional arrays more fully. Once you understand their operation, it's easier to move on. The two have very much in common, and the GWBASIC manual fully explains the use of the multi-dimensional arrays.

SUMMARY

This chapter has dealt with many often-used statements and functions in GWBASIC. It by no means has explored the full extent of this very powerful language. For the most part, the topics discussed have involved or are used with text mode operations, i.e., the display of information such as words and numbers on the screen. The next chapter deals with the high-powered graphics capabilities of this computer. Many of the functions and statements discussed in this chapter will also be used heavily in graphics mode. Please don't be under the impression that all of the uses of these functions and statements have been discussed. They certainly have not. As your knowledge and capability increases, you will find more and more ways to make the BASIC language work for you.

If at this point you are partially or completely confused, this is an indication that you may not have studied the contents of this chapter thoroughly enough. Each of the exercise programs is intended to show you exactly how functions, statements, or combinations of both operate within a simple program. If you're not clear on the use of these functions and statements, reread those portions which explain them. It is mandatory that you understand those portions of the BASIC language already outlined before moving further. Rewrite each of the working programs presented in this chapter, but use your own imagination to alter them. Try a few weird ideas and see how the computer responds. This can be the best tutorial of all.

Chapter 3



Graphics Programming

The AT&T PC-6300 is exceptional in graphics, an area where many personal computers fail. This is not to say that sophisticated graphics cannot be programmed on them—only that the process is often an ordeal. Drawing simple shapes such as circles, rectangles, and triangles can involve many, many complex program lines. This does not apply to the PC-6300.

In order to take full advantage of the PC-6300's graphics capabilities, it will be desirable (but certainly not mandatory) to purchase the optional RGB color monitor. The RGB monitor will allow you to see the various colors that can be produced, but if you don't have one, the color graphics programs will still operate. When using the monochrome monitor, all color commands simply cause the screen to display points of light in various shades of gray.

This chapter assumes the use of the color monitor, but again, your monochrome monitor will suffice. If you obtain a color monitor at a later date, the programs that now display various geometric patterns in monochrome will do the same in color.

THE GRAPHICS SCREEN

There are three types of screen formats used in programming graphics on the PC-6300. The one most commonly used is called *medium-resolution graphics.* This screen is composed of 320 horizontal points and 200 vertical points. The second graphics mode, *high-resolution graphics,* is monochrome-only and displays 640 horizontal points by 200 vertical points. These points of light are referred to as *pixels.* In medium-resolution graphics mode, each pixel may be one of four different screen colors at any one time. In high resolution graphics mode, only two colors are available. These are usually referrred to as "black and white," although different monitor screens may display them in green and black or even in amber and black.

The last standard screen mode is called *proprietary graphics mode* and consists of 640 horizontal pixels and 400 vertical pixels. Table 3-1 provides a breakdown of the various screen modes available with the PC-6300 and the SCREEN statement numbers that access them.

The SCREEN statement is used to set the mode of operation. This statement may be used with a number of designators, but for this discussion, I will only include the always-mandatory first one. The SCREEN statement is always followed by a number that indicates the mode to which the screen format is set. This statement may also be used to set the active page, the visual page, the range of colors displayed, and so on. Once you get the hang of graphics programming, these latter designations will be used more and more. For now, we will stick to the absolute basics to avoid confusion. For most discussions in this chapter,the medium-resolution graphics mode (SCREEN 1) is used. This is the standard graphics mode. Fortunately, all three graphics modes work pretty much alike, with the only differences being their rearranged or expanded coordinates.

SCREEN COORDINATES

When programming graphics on the PC-6300, it is usually necessary to locate the point on the screen at which a graphic drawing is to begin. You will recall from Chapter 2 that the LOCATE statement is used to define a point on the screen where a text character is to be written. This statement is not valid in locating a position for a graphics write, but we still use a two-part coordinate system similar to the one used with the LOCATE.

The AT&T graphics screen consists of a coordinate system that first specifies the horizontal position and then the vertical position. This is the reverse of the LOCATE statement, which first specifies vertical position (row), and then the horizontal position (column). Figure 3-1 shows the medium-resolution graphics screen. You will recall that this consists of 320 horizontal points and 200 vertical points. The coordinate numbering system begins with 0 instead of 1, so the horizontal points are numbered from 0 to 319. The same applies in the vertical axis, where the points are numbered from 0 to 199 for a total of 200 points. It is standard practice in graphics programming to think of the horizontal points as falling along the X-axis, with the vertical points making up the Y-axis. By specifying coordinates in terms of X and Y, we can easily arrive at an exact position on the screen where a graphics write is to begin.

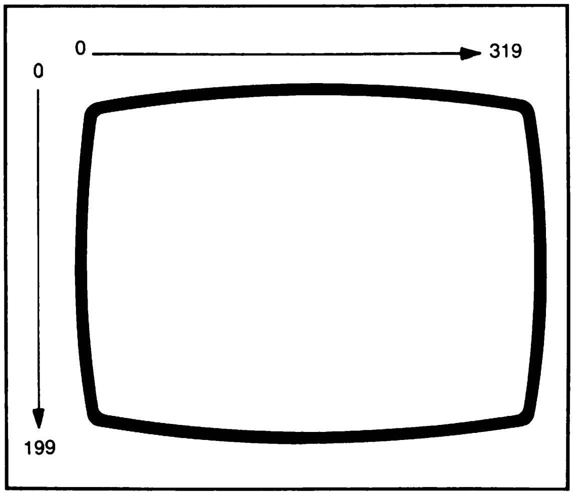


Fig. 3-1. The PC-6300 medium-resolution graphics screen consists of 320 points horizontally (0-319) and 200 points vertically (0-199).

Table 3-1. Screen Modes.

SCREEN 0	Text Mode
SCREEN 1	Medium-Resolution Graphics Mode (Four Color)
SCREEN 2	High-Resolution Graphics Mode (Monochrome)
SCREEN100	Proprietary Graphics Mode (Monochrome)

The upper left-hand corner of the screen is specified by X-, Y-coordinates of 0,0. This corresponds to the text mode coordinates (used with the LOCATE statement) of 1,1. You must remember, however, that text coordinates are given in terms of row and column, whereas graphics coordinates are the equivalent of column and row. If it's easier for you, think of graphics screen as being composed of 320 columns by 200 rows. The pixel position immediately to the right of the upper left-hand corner is specified as 1,0. This indicates horizontal pixel 1 and vertical pixel 0. The 0 in the coordinates indicates the vertical position that is at the very top of the screen, while the 1 represents the horizontal or column position position that is one position to the right of 0. (Remember, 0 is the first number in any row or column.) Thus, the upper *right-hand* corner of the graphics screen in medium-resolution mode would be specified in X-Y-coordinates as 319,0.

Since the screen consists of 320 horizontal points by 200 vertical points, the exact center of the screen would be specified by the X, Y coordinates 160,100. The number 160 is exactly half of 320 (the number of horizontal points), and 100 is half the total vertical points. By specifying these coordinates, the starting position for a graphics write is at the exact center of the screen. The remainder of the discussion will use statements and the POINT function in a medium-resolution screen format, SCREEN 1. These functions will work identically in either low- or high-resolution graphics mode, provided you take into account the increased or decreased horizontal point count. For instance, while 160,100 specifies the center of the medium-resolution screen, the coordinates 320,100 specify the exact center of the high-resolution screen; likewise, the coordinates 80,100 specify the exact center of the low-resolution screen. In each case, the total points horizontally and vertically on the screen are each divided by 2. The vertical resolution remains the same (200) in any of the three graphics mode.

CIRCLE

The CIRCLE statement in GWBASIC allows us to draw circles on the screen using a single program line. For now, the format for using this statement may be thought of as:

```
CIRCLE(X,Y),R
```

Here, X and Y are horizontal and vertical coordinates, respectively of the center of the circle, and R is the radius in screen points. The following program will effectively demonstrate the use of the CIRCLE statement.

```
10  CLS
20  SCREEN 1
30  CIRCLE(160,100),60
```

This program will draw a circle with a radius of 60 pixels (diameter 120 pixels) at the center of the medium-resolution graphics screen. Here's how it works. Line 10 uses the CLS statement to clear the screen. The SCREEN statement in line 20 puts the screen into medium-resolution graphics mode. Following the CIRCLE statement in parentheses are the coordinates which, in this case, specify the center of the circle at the center of the screen. This is where the graphic write will begin. Now, no point is actually written at the center; this just specifies a starting point for the plotting. The radius designation of 60 assures a perfect circle on the screen 60 points in all directions from the X, Y coordinates. When you RUN the program, you will immediately see a large circle at the center of the screen. Now, change line 30 to:

```
30  CIRCLE(100,100),60
```

When you run the program again the same circle appears, but shifted to the left because the X-, Y coordinate values were changed in line 30. The horizontal position is now at X coordinate 100, which is 60 points less than, or to the left of, the previous horizontal coordinate. To move the circle to the right of center, you can change the coordinates in line 30 to:

```
30  CIRCLE(220,100),60
```

This will shift the circle 60 points to the right of screen center. To move the circle up and down, it is necessary to alter the Y coordinate specification. Change line 30 to:

```
30  CIRCLE(160,80),60
```

The Y coordinate of 80 is 20 points higher (i.e., less than) the previous Y coordinate. The circle will now appear in the upper half of the medium-resolution screen. The center of the circle will be at the horizontal center of the screen, but somewhere above the vertical center.

To see what the radius value does, change line 30 to:

```
30  CIRCLE(160,100),30
```

The X-Y coordinates again specify a point at the exact center of the medium-resolution screen. However, the circle radius has been halved, so the circle will be half the size of the previous one.

At this point, try fooling around with the values contained in line 30. You will find that you can change the size of the circle by changing the value of the radius. You can move the circle left and right by altering the value of the X coordinate, and up and down by altering the Y coordinate. You may even specify a value that causes part of the circle to exceed the screen width. This can be done by changing line 30 to:

```
30  CIRCLE(280,100),60
```

This places the circle at the extreme right of the screen. The computer will attempt to display a uniform circle, but there is not enough screen space to do it. If you add 60 points to the horizontal coordinate of 280, you arrive at a total of 340 points horizontal, but the screen can only display points to 319, maximum. No error message occurs here; that portion of the circle beyond the coordinates of the screen is simply not shown.

The CIRCLE statement is a good one to begin with when discussing graphics programming, because it demonstrates how screen coordinates and graphics statements are used. Those readers who have had experience with other types of computers that do not offer the CIRCLE statement will immediately realize its value. Drawing a circle by plotting points one at a time is an extremely difficult program operation. You can do it, but it's definitely not fun.

LINE

The LINE statement does just what its name implies. It draws a line on the screen. (It can also do much more, but this will be saved for later.) The LINE statement can be used in many ways, but for now, let's assume that the format is:

```
LINE(X,Y) - (X2,Y2)
```

Hey! What's that X2,Y2? Don't panic! The explanation is quite simple. Obviously, a line must have a starting point and an ending point. Therefore, the LINE statement uses two sets of X-Y coordinates, the first indicating the starting point and the second set indicating the ending point. The following program demonstrates the use of the LINE statement in GWBASIC:

```
10  CLS
20  SCREEN 1
30  LINE(10,100) - (160,100)
```

Lines 10 and 20 do the same thing they did in the previous CIRCLE demonstration program. In line 30, the LINE statement simply tells the computer to draw a line from screen coordinate 10,100 to screen coordinate 160,100. Since the Y coordinate values (Y = 100) are the same in both sets, this will be a horizontal line, drawn across the center of the screen. The line will be 150 pixels in length (160 − 10 = 150).

Now, let's draw a vertical line on the screen by changing line 30 to:

```
30  LINE(160,50) - (160,150)
```

Here, the first coordinate designation (X = 160) stays the same in both sets, so the horizontal position is fixed. Only the Y or vertical coordinate values change from set to set. When this program is run, a line will be drawn at the horizontal center of the screen from vertical coordinate 50 to vertical coordinate 150. We can also draw diagonal lines simply by changing both sets of coordinates. To demonstrate this, change line 30 to:

```
30  LINE(50,80) - (90,150)
```

This will draw a diagonal line slanting downward from left to right. You can now experiment a bit with the LINE statement to see what happens when coordinate values are switched around. You must be certain that X coordinate values range from 0 to 319 and Y coordinate values range from 0 to 199. If a coordinate is out of this range, an error message will result.

This method of forming lines by specifying coordinates is known as the *absolute* method, in that the LINE statement examples thus far have all specified exact starting *and* ending points of the lines to be drawn. However, there's also another way to draw lines, which the following program demonstrates.

```
10  CLS
20  SCREEN 1
30  LINE(50,10) - (50,100)
40  LINE  -  (150,100)
```

When this program is run, you will see a crude letter "L" drawn on the screen. You have no doubt noticed that line 40 contains another LINE statement, this one with only one set of coordinates, preceded by a hyphen (-). Where the LINE statement in line 30 uses the absolute method of specifying coordinates discussed previously, the LINE statement in line 40 uses the *relative* coordinate method. Here's what happens. The computer first draws the line programmed in line 30. When this first line has been drawn, the graphic coordinate position stops at 50,100. Line 40 tells the computer to draw another line from this *present* graphic position 50,100 to coordinate (150,100). To visualize this, one must imagine an invisible "graphic cursor" much like the text cursor discussed in the previous chapter. When line 30 is executed, the graphics cursor is first positioned at coordinate 50,10 and then travels to coordinate 50,100, where the line ends. When line 40 is executed, the graphics cursor position of 50,100 is already locked into the computer; so it assumes the line start point to be 50,100. Using the relative method of drawing a line, all that is necessary is to specify the line end point. You can run this program another way by removing line 30 altogether. When this revised program is run, a line will be seen starting at the center of the screen (coordinates 160,100) and ending at the specified coordinates of 150,100. The computer always positions the graphic cursor at center when graphics mode is first entered.

We humans are often accustomed to thinking of everything as beginning at the left and ending at the right, but the computer is not at all handicapped by this false assumption. In this case, a line was drawn from right to left rather than from the standard left to right. Since we were using the relative form of coordinate specification, and the graphics cursor was automatically set to coordinate 160,100 when the graphics screen was first initalized, the computer drew a line starting at coordinate 160,100 and moving left to coordinate 150,100. The same line could be drawn on the screen from left to right by specifying:

```
Line(150,100) - (160,100)
```

This line would begin at horizontal coordinate 150 and end at coordinate 160. Either way, the line looks the same even though the computer looks at it in a completely different manner. For the purposes of our discussion, the lines

```
LINE(150,100) - (160,100)
LINE(160,100) - (150,100)
```

produce identical results on the screen. Either way, a line 10 pixels long is displayed *between* coordinates 150,100 and 160,100.

COLOR

The COLOR statement in GWBASIC is aptly named, because it determines the color or colors in which each point on the screen is to be plotted. As was noted previously, four colors may be displayed simultaneously on the medium-resolution screen, and only black-and-white on the high-resolution screens.

The COLOR statement is of the format:

```
COLOR BACKGROUND,PALETTE
```

Table 3-2. Color Background and Palette Designators.

0	BLACK	8	GRAY
1	BLUE	9	LIGHT BLUE
2	GREEN	10	LIGHT GREEN
3	CYAN	11	LIGHT CYAN
4	RED	12	LIGHT RED
5	MAGENTA	13	LIGHT MAGENTA
6	BROWN	14	YELLOW
7	WHITE	15	HIGH INTENSITY WHITE

COLOR	PALETTE 0	PALETTE 1
1	GREEN	CYAN
2	RED	MAGENTA
3	BROWN	WHITE

where the BACKGROUND color is a numeric expression in the range of 0 to 15, and the PALETTE specification is either 0 or 1. These represent two sets of palette colors, both of which are fixed. However, you have complete control over the background colors. Table 3-2 shows the numeric specifications for background color, as well as the palette colors available and their color designators. When you follow the COLOR statement with one of these numbers, this sets the background color for your display. When medium-resolution graphics mode (SCREEN 1) is first entered, the background color is automatically set to 0, but you can change this using the COLOR statement. At this point, you're probably totally confused about the use of palette. Don't worry. We will be returning to the LINE and CIRCLE statements shortly to show just how this unusual combination works. It's quite effective and very easy to use once you get the hang of it—and this doesn't take very long.

First, let's fool around with the COLOR statement by itself. Run the following program:

```
10  CLS
20  SCREEN 1
30  COLOR 1,0
```

When this program is run the screen will immediately change to a blue background, replacing the default black that would normally be seen, because the background color code specified in the COLOR statement is 1. (See the previous background color chart.) For now, don't worry about the second number used with the COLOR statement. This is the PALETTE—which has no effect on background—but is used by other graphics statements to plot points in a specific color.

You can change this program by inserting any other number from 0 to 15 as the first number in the COLOR statement in line 30. This will allow you to see the background colors available on the PC-6300. The following program, however, does this more simply.

```
10  CLS
20  SCREEN 1
30  FOR=0 TO 15
40  COLOR X,0
50  FOR Y=1 TO 1000
60  NEXT Y
70  NEXT X
```

This program helps demonstrate that most of the commands, statements, and functions used in text mode programming are used just as heavily in graphics mode. You can see the familiar FOR-NEXT loops, discussed in Chapter 2, used here in a graphics program that displays all of the possible screen background colors. Line 10 clears the screen, and line 20 puts us in medium-resolution graphics mode. The FOR-NEXT loop that begins in line 30 counts from 0 to 15, the values that represent background colors in the COLOR statement. The value of X is used with the COLOR statement in line 40.

Within this loop is yet another FOR-NEXT loop. This one, which assigns the variable Y, is known as a "nested" loop, since it resides wholly within the outer loop. Each time line 50 is encountered, it counts from 1 to 1000, and is then terminated by the NEXT statement in line 60. The NEXT statement in line 70 causes the (outer) X loop to recycle.

You may be surprised to see that the value of Y is not used anywhere in the program. The loop assigns to Y a value of from 1 to 1000, but because there are no other statements in this inner loop, you might assume that it does nothing, this is incorrect. The loop does something to computer execution: it slows it down, because it takes the computer a few seconds to count from 1 to 1000. This is called a *time delay* loop. One might now ask why you would want slower execution, since a main concern today is with execution speed.

Time delay loops are extremely useful in graphics programs because they give us humans a chance to see what's on the screen before that screen changes to something else. Here's what happens. The COLOR statement in line 40 is assigned a value which changes the background color of the screen. There is then a slight delay while the loop in lines 50 and 60 times out. When the loop times out, the outer loop recycles and the COLOR statement is assigned another background value. To demonstrate the value of the time delay loop, remove lines 50 and 60 from this program and RUN it again. Hey! The screen simply went from black to white and didn't show all the colors in between. Wrong! It showed all of the colors as before, but it showed them so rapidly that your eye couldn't follow the sequence. The time delay loop allowed each background color to be fully established before being changed to another.

Now back to the original program. When it is run, every color from black to high-intensity white is displayed as the background.If you're tired of looking at a black background, the COLOR statement can be used to change it to any of the fifteen other colors.

Now that you understand this, let's discuss the palette colors. These are the colors that can be written over the background you establish. If you opt for palette 0 (as in COLOR 1,0, for instance) then any other graphics statements such as LINE or CIRCLE will draw objects in any one of the three colors assigned to palette 0. Both the CIRCLE and the LINE statement may be used with an additional numeric designator to assign a color from the preselected palette to that line or circle. When you don't supply this designator the computer assumes a default palette color of 3, but we can change all this by using the COLOR statement and palette assignment numbers with our graphics commands.

In palette 0, the number 1 will give us a green screen write, the number 2 will bring about a red write, and the number 3, brown. In palette 1, the same numbers give us cyan, magenta, and white, respectively. Now, let's go back to the CIRCLE statement and see what can be done when this is combined with the COLOR statement. You will remember that the format for the CIRCLE statement is:

```
CIRCLE(X,Y),R
```

However, to use color with the CIRCLE statement, the format is:

```
CIRCLE(X,Y),R,C
```

In this case, C represents a palette color number from 1 to 3. We cannot assign a palette here; this must be done using the COLOR statement. What we do specify is which color within the preassigned palette the circle is to be drawn. The following program will demonstrate this:

```
10  CLS
20  SCREEN 1
30  COLOR 0,0
40  CIRCLE(160,100),60,2
```

Here, line 30 assigns a background color of black, represented by the first zero, and a palette of 0, represented by the second zero. When this program is run, a red circle is drawn against a black background because the number 2 has been used as the color designator in the CIRCLE statement in line 40. Referring to the chart, you will find that the number 2 represents the color red in palette 0. If you change the color designator in line 40 to 1, the circle will be drawn in green; changing the color designator to 3 produces a brown circle. (This will appear on most monitors more as a greenish-yellow.)

Now, change line 30 to:

```
30  COLOR 0,1
```

You will now be able to display your circle in any of the three colors available in palette 1. This means that a 1 will produce a circle in cyan, which appears as a light blue against a black background. Colors 2 and 3 will bring about magenta and white, respectively. Magenta appears as a medium purple against a black background. By changing the screen background color, the foreground colors will look a bit different because they are highlighted in a different manner.

The program in Fig. 3-2 will show how a circle looks in each of the six available palette colors (two palettes with three colors each) on each of the sixteen different background colors. This program may be a little complex for the beginning computer hobbyist, so don't worry too much about its contents. Its purpose is simply demonstrate colors. As the program runs, the BACKGROUND, PALETTE, and COLOR numbers are displayed on the screen. you will note that in some instances the circle seems to disappear altogether, due to the fact that it is written in color which is a lighter shade of the background color. In other words, the circle is there but it cannot be seen, because the background color overpowers it.

```
10 CLS
20 SCREEN 1
30 FOR X=0 TO 15
40 COLOR X,0
50 LOCATE 1,1
60 PRINT"BACKGROUND =";X
70 LOCATE 2,1
80 PRINT"PALETTE = 0"
90 FOR Y=1 TO 3
100 LOCATE 3,1
110 PRINT"COLOR =";Y
120 CIRCLE(160,100),60,Y
130 FOR Z=1 TO 1000
140 NEXT Z
150 NEXT Y
160 COLOR X,1
170 LOCATE 2,1
180 PRINT"PALETTE = 1"
190 FOR YY=1 TO 3
200 LOCATE 3,1
210 PRINT"COLOR =";YY
220 CIRCLE(160,100),60,YY
230 FOR ZZ=1 TO 1000
240 NEXT ZZ
250 NEXT YY
260 NEXT X
```

Fig. 3-2. Program to display PC-6300 background colors.

This points to another designation that can be used with the CIRCLE and LINE statements and with most other graphics statements as well. While there are three palette colors numbered 1 through 3 in each palette, you can also specify a color designation of 0 with either CIRCLE or LINE. This means that the object will be written on the screen in the same color as the screen background, and therefore will be invisible. This is demonstrated by the following program:

```
10 CLS
20 SCREEN 1
30 COLOR 0,1
40 CIRCLE(160,100),60,2
50 FORY=1 TO 1000
60 NEXT Y
70 CIRCLE(160,100),60,0
```

When this program is run, you will see a circle at the center of the screen drawn in magenta. After a few seconds, the circle will suddenly disappear, because the same circle is drawn again with a color 0 specification, which is the screen background color. As far as the computer is concerned the circle is still there, but we can't see it because it has blended with the background.

The 0 designator is handy for erasing previously drawn images from the screen. We could use a CLS statement to clear the screen with a lot less input time—but suppose we wanted to draw two circles on the screen in different locations, and then erase one of them. Using CLS, the entire screen would be cleared, but using a color designator of 0 for the circle to be erased, along with the same screen coordinates and radius value used when it was originally drawn, we could easily erase one circle while preserving the other. The following program demonstrates just this:

```
10  CLS
20  SCREEN 1
30  COLOR 0,1
40  CIRCLE(160,100),60,2
50  CIRCLE(20,100),20,3
60  FOR X=1 TO 1000
70  NEXT X
80  CIRCLE(160,100),60,0
```

This is the same program as before, except an additional CIRCLE statement has been added to draw a smaller circle at the left side of the screen. The CIRCLE statement in line 80 effectively erases the larger circle while preserving everything else on the screen.

More than anything else in graphics programming, the combinations of colors are the subject of a great deal of experimentation. You should play with the various combinations for several hours in order to obtain a full grasp of all that is available to you. The discussion to this point should have given you the ability to know *how* to experiment, but this cannot replace the actual experiment itself. You must put in these hours of time.

From a color standpoint the LINE statement works just like the CIRCLE statement. The format is:

```
LINE (X,Y) - (X2,Y2),C
```

Here, C is the palette color (1-3) in which the line is to be written. You can also use 0 for the color number, and the line will be written invisibly in the background.

At this point, we can go a bit further with the LINE statement and use it to draw boxes. To do this, the format is simply:

```
LINE (X,Y) - (X2,Y2),C,B
or:

LINE (X,Y) - (X2,Y2),C,BF
```

Again, the C represents the color number. However, the new designators B and BF stand for *box*, or *box fille*, respectively. The B forms a box ed with the screen background color. Assuming that you have a black background and the lines are being drawn in white, the box will have a black fill. However, if you use the BF designator, the box is filled with the same color used to draw the outline of the box. When using the LINE statement to draw boxes, the first set of coordinates as represents the upper left-hand corner of the box while the second set of coordinates represents the lower right-hand corner of the box. The following program demonstrates this:

```
10  CLS
20  SCREEN 1
30  COLOR 0,1
40  LINE(120,50) - (200,150),3,B
```

Here, the background color is designated as 0 (black) by the COLOR statement in line 30, and the palette is 1. (This is the default color mode when medium-resolution graphics is first entered, so line 30 is really unnecessary. It is included here to demonstrate that a COLOR statement is always in effect, whether specified or not.) The LINE statement in line 40 uses the B designator, so you know a box is going to be drawn. the upper left-hand corner of the box will be at screen coordinates 120,50. The lower right-hand corner of the box will be at screen coordinates 200,150. The 3 designates the color white while operating in palette 1. When the program is run you will see a rectangle drawn at the center of the screen, and, if you could lay a grid over the screen, you would find that the upper left-hand corner of the box is indeed at coordinates 120,50 and the lower-right hand corner is at coordinates 200, 150.

As an experiment, edit line 40 to remove the B designator and the comma that precedes it. When the revised program is run, a diagonal line will be drawn from coordinate 120,50 to coordinate 200,150. The B designator, however, tells the computer that this is not to be a line. Certainly, we can make boxes using the LINE statement alone. Without the B designator, the same box could be drawn on the screen by altering the program as follows:

```
40  LINE(120,50) - (200,50),3
50  LINE(120,50) - (120,150),3
60  LINE(120,150) - (200,150),3
70  LINE(200,50) - (200,150),3
```

You can see that it took four program lines instead of one to draw the box when not using the B designator. For this reason, any time you wish to draw a square or rectangle of any dimensions, you should always use the B designator for the most efficient program in the shortest amount of programming time.

Let's go back to our original program now and change line 40 to:

```
40  LINE(120,50) - (200,150),3,BF
```

The BF designator is used in this case to indicate that the box is to be filled with the current palette color (white, in this case). When this program is run, you will see a solid rectangle at the center of your screen, of the same dimensions as the previous rectangle. Alter the color numeral in line 40 any way you want, so long as it includes a number from 1 to 3, and you will see how the box fill is affected.

PAINT

The PAINT statement is used to fill in an enclosed area on the screen. Put simply,the PAINT statement fills in an enclosed object such as a circle, square, etc., with one of the three colors available in the current palette. Recall here that when LINE is used with the BF designator, a box is formed and filled with the current color designated by the color numeral included in LINE. In every case, the box is filled with the same color used to draw the box. The PAINT statement, however, allows us to choose any of the palette colors to be used as a fill, keying on the color of the lines which make up the object it is filling. For example, if a box we wish to PAINT is drawn with white lines, we locate a point within the box, and then tell the computer the color in which to start painting, as well as the color representing the boundaries of the object it is to fill. The PAINT statement wildly fills in the area that is completely surrounded by the object lines. It stops painting only when it reaches a line of the color designated to be the edge of the object.

Confused? The PAINT statement is easier to demonstrate within a program than it is to explain in words. The following program will draw a circle on the screen, and then fill it with another color using the PAINT statement.

```
10  CLS
20  SCREEN 1
30  COLOR 0,1
40  CIRCLE(160,100),60,3
50  PAINT(160,100),2,3
```

After the screen is initialized, the background color is set to black (0) and the palette to 1 by the COLOR statement in line 30. Line 40 draws a circle with a radius of 60 pixels at the center of the screen. The circumference is drawn in palette 3, which is white. Line 50 uses the PAINT statement; coordinates are specified here, but they are really unimportant as long as they specify any point *within* the circle. Since the circle spans from coordinates 100,40 to 200,160—the coordinates plus the radius value—any coordinate specifications will work if they lie within these sets of coordinates but do not include them. Immedately following the coordinates is the color numeral which specifies the color to be used for fill. (In palette 1, the number 2 specifies magenta.) The next numeral is the same as the numeral used to draw the circumference of the circle in the previous line. This tells the computer to paint until it hits a white pixel, at which point it stops in that direction.

If you watch closely when the program is first run, you will see that the circle is drawn first and then filled in. You can still see that the circumference of the circle is white, while the interior is filled with magenta. Now, change line 40 to read:

```
40  CIRCLE(160,100),60,1
```

Here we are specifying that the circumference be written in palette color 1, which is cyan. Run the program again.

Hey! What happened? I tricked you on this one to demonstrate what happens if you don't give the PAINT statement a correct boundary color at which to stop. By changing line 40 to draw the circle in cyan, it was also necessary to change the boundary numeral in the PAINT statement to reflect this. If you followed my instructions, you made no changes to the PAINT statement. The computer started painting—and kept on painting because it encountered no white dots (color number 3) on the screen. It stopped when the entire screen was filled with magenta. Let's correct this situation by changing line 50 to:

```
50 PAINT(160,100),2,1
```

Here, the paint color is still magenta (color number 2), but the boundary color numerical has been changed to the one used to draw the circumference. When you run the program now, the cyan circle is drawn on the screen and then filled in with magenta.

You can change the PAINT color number in line 50 to either 1 or 3 to bring about a different color of fill. The critical factors are the boundary number, which *must* reflect the object line color, and the coordinates. Change line 50 to:

```
50 PAINT(130,60),3,1
```

Now run the program. It still works fine, because the point on the screen represented by coordinates 130,60 still lies within the circle's boundaries. Now, change line 50 to:

```
50 PAINT(110,60),3,1,
```

When this program is run the entire screen is filled, *except* for the area encompassed by the circle. The point represented by coordinates 110,60 lies outside of the circle. Since the circle is composed of points of color numeral 1, which the PAINT statement cannot paint over, the computer keeps painting the screen until it is filled from border to border by the erroneous PAINT statement. You must specify a coordinate value that lies within the object to be painted, and also make certain the boundary number designator in the PAINT statement agrees with the color of the object being painted.

Remember, the PAINT statement can be used to fill any enclosed object, including boxes or rectangles, as is demonstrated by the following program:

```
10 CLS
20 SCREEN 1
30 COLOR 0,1
40 LINE(100,50) - (220,150),3,B
50 PAINT(160,100),2,3
```

When this program is run, a box will appear at the center of the screen, drawn in white and then filled by magenta. Note again that coordinates used in the PAINT statement are inside the box, and that the boundary designation is the same as the line color used to draw it.

A PAINT statement can be used to fill only *enclosed* objects. Painting an enclosed object is not unlike blowing air into a balloon: if the balloon is not completely enclosed, the air is going to leak out. The following program demonstrates what happens when you try to use a PAINT statement with an object what is not fully enclosed.

```
10 CLS
20 SCREEN 1
30 COLOR 0,1
40 LINE(50,50) - (150,50),3
50 LINE(50,150) - (150,150),3
60 LINE(150,50) - (150,150),3
70 LINE(50,50) - (50,148),3
80 PAINT(140,100),2,3
```

This program will draw an almost-enclosed box on the screen. Line 70 specifies an ending point that does not quite reach the bottom horizontal line of the box; a two-pixel gap is left. Now, RUN the program. The box is indeed drawn and the filled, but the paint "leaks out" through the hole in the bottom left corner, and the remainder of the screen is filled in as well.

The following program uses two LINE

statements with B designators to draw two rectangles on the screen, one inside the other. Both are drawn in a palette color of white, represented by the color numeral 3. Two PAINT statements are used to fill in these boxes with different colors, and the effect is quite pleasing.

```
10 CLS
20 SCREEN 1
30 COLOR 0,1
40 LINE(110,60) - (210,140),3,B
50 LINE(130,90) - (190,110),3,B
60 PAINT(111,70),2,3
70 PAINT(160,100),1,3
```

The first PAINT statement (line 60) specifies a point within the larger box but outside the smaller one, which fills this area with magenta, (color numeral 2). The second PAINT statement names a point within the inner box and paints it with cyan. The result is a cyan box within a magenta box, but both boxes have white borders. At this point, you have learned enough to draw three boxes on the screen. Draw the second within the first, and the third within the second. By using three separate PAINT statements, you can fill each box with a different color. Try writing and running this program now.

PSET/PRESET

The PSET statement is used to write a point or pixel at a certain screen location specified by X-, Y-coordinates. PRESET is almost identical to PSET but is not often used in standard graphics programming; the differences will be discussed later. PSET is used in this format:

```
PSET(X,Y),C
```

Here, X and Y form the horizontal and vertical coordinates, respectively; C is a color numeral from 1 to 3, used in the same way as in the CIRCLE and LINE statements. The following program demonstrates the use of PSET:

```
10 CLS
20 SCREEN 1
30 COLOR 0,1
40 PSET(160,100),3
```

This program will draw a single point of light at the exact center of the medium-resolution graphics screen, displayed in white because of the color numeral 3. (In palette 0, it would have been brown.) Using PSET, we can put a single point of light at this or any other valid coordinate set on the graphics screen.

Remember that the LINE statement simply connects individual points of light to form vertical, horizontal, or diagonal lines. The CIRCLE statement uses the same individual points of light to form circles. Using PSET, however, we can theoretically draw any type of object, regardless of its complexity, by simply knowing where to position the dots. Of course, there's no *simple* way to know where to position dots when drawing a complex object, but PSET does give us some very good possibilities for drawing unusual arcs, sine waves, etc. If a circle is drawn with points of light using a mathematical formula, then, other objects may be drawn by substituting different formulas. If you're into geometry and have access to the formulas used to draw parabolas, hyperbolas, tetrehedrons, etc., you can program these images on the computer, using the screen as you would a piece of graph paper.

The program that follows shows how the PSET statement may be used within a FOR-NEXT loop to draw a horizontal line from left to right on the screen.

```
10 CLS
20 SCREEN 1
30 COLOR 0,1
40 FOR X=0 TO 319
50 PSET(X,100),2
60 NEXT X
```

When this program is run, a line will be drawn from coordinate 0,100 to coordinate 319,100, composed of 320 dots placed side by side. This is the equivalent of:

```
LINE(0,100) - (319,100),2
```

The following program will draw a diagonal line from coordinate 0,0 to 199,199:

```
10  CLS
20  SCREEN 1
30  COLOR 0,1
40  FOR X=0 TO 199
50  PSET(X,X),3
60  NEXT X
```

In both programs, the value of X assigned by the cycling of the loop was used for one or both of the X-Y coordinates. The usage makes PSET terribly handy. You will almost never see large numbers of PSET statements used outside of a loop, as this would be very inefficient programming. However by placing PSET within the loop and changing its coordinates based upon the loop value, the equivalent of hundreds of thousands of PSET statements are executed by a single program line.

Another definite asset of PSET is that it can be used to draw *multicolored* lines. The following program shows such an example:

```
10  CLS
20  SCREEN 1
30  COLOR 0,1
40  FOR X=0 TO 319
50  C=C+1
60  IF C=4 THEN C=1
70  PSET(X,100),C
80  NEXT X
```

This program will display a horizontal line on the screen in cyan, magenta, and white. (The white will not be too obvious, since it is somewhat affected by the two colors that border it, but the color is definitely there.) This program uses a counting routine of the type discussed in Chapter 2. Line 50 increments variable C (for Color) by 1 each time the loop cycles; line 70 then uses it to set the point color. Since the palette colors run from 1 to 3, the IF-THEN statement in line 60 resets C back to a value of 1 when it has counted past 3. The sequence then begins again. Using PSET in a FOR-NEXT loop as shown here can result in some very colorful displays.

As was the case with CIRCLE and LINE, PSET may also be used with a color numeral of 0, which simply plots a point in the background color, making it invisible. If no color numeral is used with PSET, the default is color numeral 3. Now, the difference between PRESET and PSET is that, when used without a color numeral, PRESET's default value is 0—the background color. Used with a color numeral, it is exactly like PSET. PRESET can be used to erase a dot, after it has been written using PSET, by simply writing over it in the screen background color. This can be used to effect some simple animation, as represented by the following program:

```
10  CLS
20  SCREEN 1
30  COLOR 0,1
40  FOR X=0 TO 319
50  PSET(X,100),3
60  PRESET (X,100)
70  NEXT X
```

Here's what happens when this program is run. Line 50 writes a point of light on the screen at the specified coordinate. The PRESET statement in line 60 immediately erases that point of light. The loop cycles, the PSET statement writes a dot at the next location, and line 60 just as quickly erases it. This process continues until program execution is terminated. The effect is that of a fast-moving dot traveling from left to right across the center of the medium-resolution graphics screen. (The same effect could be had by changing line 60 to 60 PSET(X,100),0.)

DRAW

The DRAW statement is used to quickly draw objects on the screen. The object must consist of straight lines, although they can be drawn at 45 degree diagonals. The DRAW statement format is much the same as PRINT, in that quotation marks normally follow DRAW. Within quotations are the commands that tell the computer which direction the write is to be made, and for what length. The movement commands used with the DRAW statement are shown in Fig. 3-3. There are eight of them, all easy to use. The DRAW statement uses the relative form of specifying screen coordinates, although an absolute specification is usually given to set up the starting point. Each command is followed by a numeric expression given in screen points relative to the previous point. The following program shows how the DRAW statement may be used to create a box on the screen.

```
10  CLS
20  SCREEN 1
30  COLOR 0,1
40  DRAW "U30R30D30L30"
```

This program will draw a box slightly to the right of center on the screen. You will notice that no screen coordinates were specified. The reason the box is drawn right-of-center is that the graphics cursor is automatically positioned at coordinates 160,100 when medium-resolution graphics mode is first entered. The DRAW statement in line 40 tells the computer to move up 30 pixels (U30), then move right 30 pixels (R30), then down 30 pixels (D30), and finally, left 30 pixels (L30).

To specify a particular starting point on the screen, we use the BM designator. This tells the computer to move the cursor to a specific coordinate pair, but not to plot points. This command is separated from the other designators by a semicolon. Change line 40 to:

```
40  DRAW"BM100,60;U30R30D30L30"
```

When this program is run, you will see that the box has moved to the upper left of the screen. The lower left-hand corner of the box is at absolute coordinates 100,60, which was where the screen write began.

The box is drawn in white (the default color), but we can also insert a color designator, which is the letter C plus a number in the DRAW statement. Change line 40 to:

```
40  DRAW"BM100,60;C2U30R30D30L30"
```

When this program is run, the box will now be displayed in magenta.

Drawing objects in the relative mode using the DRAW statement allows us to change the size of our box by inserting a single additional designator. The S designator stands for *scale,* with a default value of 4. Since we did not specify an S designator, the computer assumed it to be 4. However, change line 40 to read:

```
40  DRAW"BM100,60;S8C8C2U30R30D3
    0L30"
```

When the program is run, you'll find that the box is twice its previous size, simply because we've doubled the scale factor. Now, each line in the square is 60 pixels on a side.

To go further, let's move the box more toward the center of the screen. Do this by altering the BM designator in line 40 to BM160,100. The bottom left corner of the box will now be at the center of the screen. Now we can expand upon our DRAW statement line by adding another designator. This one is represented by the letter A and stands for angle. The number that follows may range from 0 to 3. Zero represents 0 degrees,while 1 is 90 degrees, 2 is 180 degrees, and 3 is 270 degrees. Change line 4 to:

```
40  DRAW"BM160,100;A2S8C2U30R30D
    30L30"
```

D	n	Move down
E	n	Move diagonally up and right
F	n	Move diagonally down and right
G	n	Move diagonally down and left
H	n	Move diagonally up and left
L	n	Move left
R	n	Move right
U	n	Move up

Fig. 3-3. The commands used with the DRAW statement in GWBASIC.

When this program is run, it looks like the square has changed positions, and indeed it has. Now the upper right-hand corner of the box represents the point at which the box was originally written; the box has simply been rotated 180 degrees—counterclockwise—about the starting point.

The DRAW statement is so versatile that an entire book could be written about its uses alone. To go further would be to simply repeat the information about it in the GWBASIC manual. The explanation of the DRAW statement to this point should be adequate to help you over the humps in getting started. From this point on, it's simply a matter of playing with it until you feel comfortable with its use. A program included in a later chapter uses the DRAW statement and especially the angle designator (A) to arrive a very interesting computer game.

ANIMATION WITH PUT AND GET

Computer animation is really quite simple, although when many objects are animated on the same screen, the program to do this can look quite complex. As is the case with any complex program, however, on closer examination you will find that it is a combination of simple routines. Animation involves writing an object to one location on the screen, erasing it, and then writing it again at a different location. This gives the impression of motion. The following program provides a simple example of animation, making it appear as if the letter R is moving across the screen from left to right. This is not a true graphics program, but it does demonstrate the principle of animation using statements that are already familiar.

```
10 CLS
20 SCREEN 1
30 FOR X=1 TO 40
40 LOCATE 10,X
50 PRINT "R"
60 CLS
70 NEXT X
```

Here, a LOCATE statement is contained within a FOR-NEXT loop. (We are back to text again when using the LOCATE statement. Therefore, the first argument or number represents the vertical or row position, whereas the variable X represents the horizontal or column position.) On the first cycle of the loop, the letter R is printed on the screen by the PRINT statement at position 10,1. As soon as it is printed, the R is erased by the CLS statement in line 60. When the loop cycles again, the R is printed at position 10,2, erased, and then printed at 10,3, and so on. By quickly writing this character to the screen, erasing it, and then printing it again at an advanced position, we simulate movement. (It's not a smooth movement, because we can see the flicker of the image being erased.) The graphics language of the PC-6300 improves upon this principle tremendously, but it is still the same principle.

The PUT and GET statements in GWBASIC can be used in either text or graphics mode. Unlike most statements, PUT and GET mean one thing in text mode, and a very different thing in graphics mode. In text mode, PUT and GET are used to store and retrieve information from random files. In graphics mode, however, PUT and GET are used to rapidly move graphics objects around the screen.

The GET statement is used to retrieve an image from the screen. The PUT statement is used to place this same image at different locations. There are many different actions that PUT can bring about, but only one will be discussed here. This is the default action. In computerese, this is known as EXCLUSIVE-OR, or XOR. At this stage it's not necessary to know all the ins and outs of EXCLUSIVE-OR, only that it is a highly useful tool in graphics animation. In this mode, an image may be put on the screen over a background, and then moved without having permanently disturbed the background. Also, when a image is put on top of itself, the image is erased from the screen. No doubt I'm beginning to confuse you again, so let's discuss the sequence of events that lead up to an animation program. This is the process:

1) Create the image on the screen using CIRCLE, LINE, PSET, DRAW, etc.

2) Retrieve the image from the screen using the GET statement.
3) Erase the original image from the screen using the PUT statement.
4) PUT the image elsewhere on the screen.
5) Erase the image form its new location, by putting it on itself, using PUT.
6) PUT the image at yet another location on the screen.

Still confused? Let's image that a circle is drawn on the screen. Before we can start the animation sequence, it is necessary to GET the circle using the GET statement. (How this is done will be described later.) Once the circle has been GOT, the computer retains the image in memory, whether it's on the screen or not. To erase the original image, we PUT the computer recollection of the image on top of the actual screen image. This causes it to disappear. We then use PUT again to place the image on the screen at a new location. We now have to erase this newly located image to place it elsewhere, so we put the computer recollection of that image onto the actual screen image again. Once the initial image has been GOT and then PUT to erase it, we always have to PUT it twice—the first time to place the image back on the screen and the second to erase it. To do so the second PUT must specify the same coordinates at the first PUT which originally displayed the image.

When we GET an image, the computer remembers exactly what it looks like by placing its contents in an array. (The principle of arrays were discussed in Chapter 2.) Fortunately, we don't have to worry about the intricacies of making assignments to an array when using PUT and GET, because they do it for us automatically. The following program will demonstrate graphics animation in a very simple way.

```
10  CLS
20  SCREEN 1
30  COLOR 0,1
40  DIM (500)
50  PRINT "R"
60  GET(0,0) - (8,8),A
70  PUT(0,0),A
```

Lines 10 through 30 simply initialize the screen. We set up our array in line 40, with its size determined by the size of the image we wish to GET. There is a fairly complex formula for calculating the minimum needed size, but most programmers simply make a stab at it—which is what we're doing with this one. (An array size of 500 elements is far more than adequate to hold the image produced in line 50. If you're short on memory, you may want to gradually shrink the size of the array, but if it gets too small, you'll have an "illegal function call" error. Then you can simply increase it a bit.) For demonstration purposes, the graphic image will be the letter R, produced in the upper left-hand corner of the screen by the PRINT statement in line 50.

The GET statement in line 60 must retrieve a section of the screen that contains the image we wish to animate. Since the R is displayed in the upper left-hand corner, we can retrieve it by specifying coordinates 0,0, which is the point at the extreme top left of the graphics screen. (Think of the retrieval method as pulling from the screen a box containing the object to be retrieved.) In this case, 0,0, represents the upper left-hand corner of the box, while the second parameter specified in GET (8,8) represents the coordinates of the lower right-hand corner. Figure 3-4 shows the box retrieval method.

The letter A following the GET statement in line 60 simply names the array to which this image is to be committed. The PUT statement in line 70 references only the upper left-hand corner of the box; the other coordinate specifications (the lower right-hand corner) are not used. You will notice that the array designation is also included at the end of the PUT statement in line 70.

In this example, the PUT statement is used to put the retrieved image onto itself. This is done by specifying the starting coordinates of 0,0, the same ones used in GET. When a visible image on the screen is PUT on top of itself, the image is erased. Run the program and you will see that the R is printed in the upper left-hand corner, and then immediately erased.

Admittedly this does not demonstrate anima-

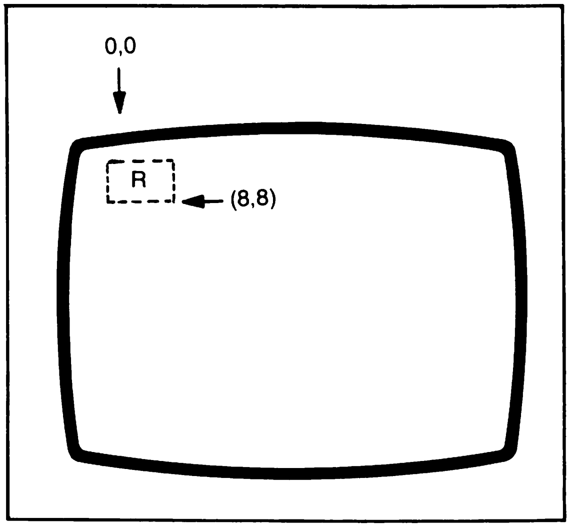


Fig. 3-4. GET captures a block from the screen containing the character to be animated. PUT places the block elsewhere on the screen.

tion very effectively, but let's go further. Add the following line to your program:

```
80  PUT (160,100),A
```

This PUT statement specifies coordinates at the exact center of the graphics screen. RUN the program. You will again see the letter R displayed at the upper left-hand corner. Then it will be erased, only to reappear near the center of the screen. We have effectively moved the letter R by animation to the center of the screen. Now, let's move it somewhere else. You might think we could do this simply by using another PUT statement with a different set of coordinates, and you would be correct—*but* we must first get rid of the R that is already at center screen. Therefore, the sequence is:

PUT an image on the screen for display.
PUT the image onto itself to erase it.
PUT the image somewhere else.

Knowing this, we know that it is necessary to PUT the R that was placed at coordinates 160,100 in line 80 to itself before moving it elsewhere. Add the following lines:

```
 90  PUT(160,100),A
100  PUT(10,50),A
```

When this program is run, the image is initially printed in the upper left-hand corner. It is then erased and PUT at the center of the screen. It is erased again and then PUT at coordinates 10,50.

There is one little problem here, in that PUT and GET work so rapidly that you really have to watch closely to see the image appear at the center of the screen. It is written and erased so quickly that it's hard to see. Once again it is necessary to slow up execution in order for the on-screen display to be recognized by humans. Add the following liners to your program:

```
85  FOR X=1 TO 400
86  NEXT X
```

Now when you run the program, the image that is moved to the center of the screen is held there for a second or so while the computer counts from 1 to 400. The FOR-NEXT loop in lines 85 to 86 is our old friend, the time delay loop. It serves no other purpose than to slow the computer down at a certain point in the program. In animation programming, you will use these delay loops often.

The following program demonstrates animation much more effectively:

```
 10  CLS
 20  SCREEN 1
 30  COLOR 0,1
 40  DIM A(500)
 50  PRINT "R"
 60  GET(0,0) - (8,8),A
 70  PUT(0,0),A
 80  FOR X=30 TO 259
 90  PUT(X,100),A
100  FOR DLAY=1 TO 200
110  NEXT DLAY
120  PUT(X,100),A
130  NEXT X
```

When you run this program, you will see the letter

R move from left to right across the center of the graphics screen. Okay! This looks more like animation. Here's how the program works. Lines 10 through 70 are identical to our first sample program. In line 80, a FOR-NEXT loop is used to count the value of X from 30 to 259; the value is then used as the X coordinate for placing our graphic image (the letter R). The PUT statement in line 90 places the letter R at the location of 30,100, since the first pass of the loop made X equal to 30. A delay loop is then entered in lines 100 and 110. The variable name DLAY is used here to be descriptive of the function of the loop.

The delay loop allows the image to establish itself at location 30,100 for a short time. When line 120 is executed, the image is erased from this location because the image has been PUT onto itself. Line 130 recycles the outer (X) loop by branching back to line 80, where X now becomes equal to 31. The image is placed at location 31,100 and the previous sequence repeats itself.

This program was purposely set up to display a slow-moving graphic image. To speed it up, simply remove our delay loop in lines 100 and 110. Run the program again, and you will see that the letter R travels much more rapidly from left to right. With this rapid movement you will also notice some flickering of the image, but this is one of the things we have to live with when programming in BASIC. The larger the image is and the faster it moves, the more it will flicker.

Is the motion still too slow for you? We can speed it up considerably by changing line 80 to:

```
80  FORX=30 TO 259 STEP 5
```

You will remember the discussion of STEP in Chapter 2. When STEP is coupled with a FOR-NEXT loop, it causes the loop variable (X in this case) to be incremented by the numeric designator included after STEP. Therefore, this loop will no longer count 30, 31, 32, 33, etc. The increments will be in steps of 5, so the loop will count 30, 35, 40, 45, etc. When this value is used as a screen coordinate, the first writing of R will be at location 30,100. The next write will be at 35,100 (as opposed to 31,100), so the screen write will jump 5 positions during each cycle. The letter R will now literally fly across the screen. While these examples have used the letter R rather than what we usually consider to be graphics objects, they have clearly explained the methods by which animation is effected on the PC-6300. Additionally, we have discussed methods of slowing down and speeding up a moving object.

Just so you don't feel cheated, let's make a slight alteration to our program so that we do animate a true graphics object. Change line 50 to:

```
50  CIRCLE(4,4),4,2
```

Here we are specifying that a circle be drawn in the upper left-hand corner. Its center will be at coordinates 4,4, it will have a radius of 4 pixels, and it will be drawn in magenta. Run the program again, and you will see that the R has been replaced by a small magenta circle. the circle moves in exactly the same manner as the R, but the flicker is not as noticeable because fewer points make up the unfilled circle than were used to display the solid R. If you'd like to fill in the circle, add this line:

```
55  PAINT(4,4),1,2
```

The PAINT statement fills in the tiny circle with the color cyan, making it much more noticeable during the animation process. Flicker will also be much more noticeable.

When teaching graphics animation, I'm often asked about why we PUT an image to itself to erase it, rather than simply using the CLS statement to clear the screen. It's a good question, and in the examples discussed thus far all PUT statements used to erase images could have been replaced with CLS—but at quite an expense. First, CLS does not work as quickly as PUTting an object to itself; this is especially true of smaller objects. The main reason for not using CLS, however, is that you may often wish to animate an object against a graphic background that must remain intact. Remember, when we PUT an image to itself it is erased from the screen, but the screen background that lies under that often remains unaffected. Therefore, by using PUT we preserve the background at all times, even though we are erasing the animated image. CLS would erase the entire screen, so it would be necessary to redraw the background after each erasure.

To demonstrate the advantages of PUTting an image to itself for erasure while preserving the screen background, restore line 80 and add another line.

```
80  FOR X=30 TO 259
75  LINE(0,104) - (319,104),3
```

Now run the program. Here, the traveling ball moves along the line, but notice that the line segment which is covered by the ball at any one time is not erased with the ball, but remains intact. Our background is preserved at all times. This is why we use PUT rather than CLS. PUT *can* be confusing. Just remember that PUT is used always to place an image on the screen, and if we PUT the same image on top of itself exactly, then that image is erased. Therefore, PUT is always used *two* times at any one location on the screen if you wish to erase the image before moving on.

POINT

There is only one function specifically devoted to graphics in GWBASIC. This is the POINT function, and it is used to return the color number for a specific coordinate set on the graphics screen. Functions are special subprograms called by name. They accept one or more values ("arguments") enclosed in parentheses, and send back ("return") to the main program a single value. Here, the word "return" simply means that the function name POINT will obtain the color number of a point on the screen, which can then be assigned to a variable. The format is:

```
C=POINT(X,Y)
```

In this case, C is the variable to which the POINT function will assign (return) the color number of the screen point which lies at coordinates X,Y. The value of C will thus be equal to the color number, which will range from 0 to 3. You will recall that in each palette there are three possible colors, represented by the numbers 1, 2, and 3. If 0 is returned to variable C, then this indicates the screen background color, which is always coded as 0, regardless of what color it may actually be.

The following program demonstrates the POINT function:

```
10  CLS
20  SCREEN 1
30  COLOR 0,1
40  PSET(160,100),1
50  C=POINT(160,100)
60  PRINT C
```

When this program is run you will see a cyan point at the center of your screen, and, in the upper left-hand corner, the number 1 will be displayed. The PSET statement placed a point of light at coordinates 160,100 in the color cyan, represented by the color numeral 1. Line 50 then uses the POINT function to read the value of the point of light at location 160,100 and assign its value to C. Line 60 then prints the value of C on the screen. Now, try changing line 40 to:

```
40  PSET(160,100),2
```

When you run the program now, the dot at the center of the screen will be magenta and C will be equal to 2. Therefore, 2 will be printed in the upper left-hand corner when the value of C is displayed.

The POINT function is extremely valuable to the graphics programmer. For instance, if you have drawn a picture on the screen and decide that you would like to change all the magenta dots to white, a simple program can be written to scan the screen. Using the POINT function, you could include a program segment such as

```
250  C=POINT(X,Y)
260  IF C=2 THEN PSET(X,Y),3
```

This is just a program segment, so don't try to run it on your computer. You must assume that the X and Y values are part of a FOR-NEXT loop and represent a valid screen point. These values are stepped through every possible combination of screen coordinates. Each time a coordinate is presented, line 250 reads the color value of that point. Line 260 tells the computer to set a white point (color numeral 3) if C is equal to a magenta point (color numeral 2). A program in another chapter even uses the POINT function to read the points of light that make up a standard character (such as A, B, C, etc.), and then to reproduce that chapter in much larger form on the screen. As you delve deeper into graphics programming, and into the modification of on-screen images, you will find more ways to use POINT.

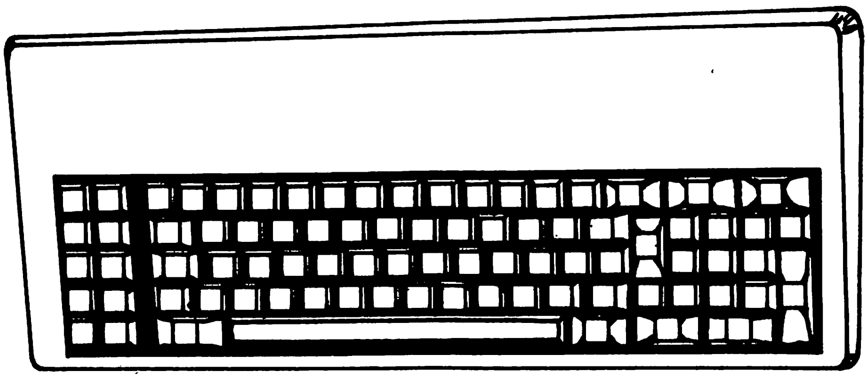
SUMMARY

Just as Chapter 2 addressed text mode programming in GWBASIC, this chapter has overviewed graphics programming using each of the graphics statements (and the single function) available in GWBASIC. I stress the word *overview* here, because his handful of graphics functions can be put to millions of uses. The discussion in this chapter has only touched upon them. However, if you have absorbed the information here, you should be able to advance quickly to PC-6300 graphics without a great deal of difficulty.

It has been my experience, and the experience of many beginners in computer programming, that the major difficulties arise in first learning the absolute basics of programming statements and functions. Even the clearest text can sometimes breed damaging misconceptions. This is why each of the discussions in the chapter has been followed by at least one example of how a particular statement or function is used in a program. Inputting these programs to the computer, seeing how they run, and making your own alterations to see what happens is the most valuable learning experience of all.

If you have absorbed the content of this chapter and the previous chapter on text programming, then the GWBASIC manual will undoubtedly have more meaning to you. You should now be in a better position to learn what all of those reference manuals have to offer.

Chapter 4



General Programs for the AT&T PC-6300

This chapter will present programs of a general nature. These are text mode programs that will do such things as make measurement conversions, convert from one type of temperature to another, and even let you figure mortgage payments on a home. Admittedly, many of these kinds of programs have been presented hundreds of times for other computers; they will not take a terribly different form for the AT&T PC-6300. All programs are run in text mode and are simple enough to serve as good tutorial examples. Each one can be put to direct use around the home or office.

Program 1: Feet to Inches Conversion

How many inches are there in a foot? You don't need a computer to tell you that the answer is 12. How many inches are there in 5 feet? You still probably don't need a computer to tell you that the answer is 60. How many inches are there in 2.36 feet? (Aha! I got you with that one!) While the answer could easily be figured on a calculator, it is simple to write a computer program that will allow you to quickly input many values of feet—and then just as quickly print out the answer in inches.

Lines 10 and 20 contain REM statements simply telling you what the program is and who wrote it. REM statement was not discussed in the first part of this book. REM stands for *remark*, and a line beginning with REM may be considered a nonexecutable line in the program. REM statements are just little notes to people who will be seeing the program listings, but are simply skipped over by the computer during execution. REM statements may appear anywhere in the program.

Line 30 sets the screen to text mode, while line 40 sets the horizontal width to 40 characters. Since this program doesn't use the function keys, the KEY OFF statement in line 50 simply switches off the line of information at the bottom of your screen (the *key*) that tells you what they do. The CLS statement in line 60 clears any screen writes that may have been left over from a previous program.

We get into the meat of the program at line 70.

```
10 REM FEET TO INCHES CONVERSION
20 REM COPYRIGHT FREDERICK HOLTZ
30 SCREEN 0
40 WIDTH 40
50 KEY OFF
60 CLS
70 INPUT"NUMBER OF FEET";F
80 I=F*12
90 CLS
100 LOCATE 13,8
110 PRINT F;"FEET =";I;"INCHES"
120 LOCATE 23,1
130 INPUT"PRESS <ENTER> TO CONTINUE";EN$

140 GOTO 60
```

Program 1. Feet to Inches Conversion.

The INPUT statement prints a prompt "NUMBER OF FEET" on the screen, telling the user that it's time for a keyboard input. The number typed in is committed to numeric variable F. Line 80 contains the mathematical formula for converting feet to inches. You simply multiply the number of feet by 12 (12 inches to 1 foot) to arrive at the inch value. Line 90 uses CLS to clear the screen again, erasing our original prompt and the entered value. The LOCATE statement in line 100 positions the text cursor at a point midway down the screen and eight characters to the right. Line 110 uses the PRINT statement to first display the value of feet (F) on the screen, followed by the quoted phrase "FEET EQUAL". This is followed by the value of the numeric variable I, which gives us the number of inches. This is followed by another quoted word "INCHES". If you input a value of 2 for feet in line 70, line 110 will display:

```
2 FEET EQUAL 24 INCHES
```

This makes for a neat display. Line 120 places the text cursor at the bottom left of the screen, a good spot to print the quoted phrase contained in the INPUT statement in line 130. This tells the user to "PRESS ENTER TO CONTINUE". The input value of line 130 is committed to string variable EN$. This value is not used anywhere else in the program; line 130 simply halts execution and allows the value printed in line 110 to remain on the screen until you're ready to input another value. When you press Enter following the prompt in line 130, the GOTO statement in line 140 is executed and there is a branch to line 60. Here, the CLS statement clears the screen again, and the program begins anew. To exit the program, you must perform a manual halt using the BREAK key.

Program 2: Inches to Feet

It is only natural after having presented a program that converts feet to inches to also include one that converts inches to feet. This program is almost identical to the first, except that the prompt is changed in line 70 and a few switches were made in line 110. Line 80 is the biggest change; we have inserted the formula for converting inches to feet. In this example, numeric variable I contains the number of inches you input at the keyboard. The formula for converting inches to feet simply involves dividing inches by 12. Therefore, line 80 assigns to numeric variable F the value of I (inches) divided by 12. Line 110 prints out this value.

You can modify this program further by get-

```
10 REM INCHES TO FEET CONVERSION
20 REM COPYRIGHT FREDERICK HOLTZ
30 SCREEN 0
40 WIDTH 40
50 KEY OFF
60 CLS
70 INPUT"NUMBER OF INCHES";I
80 F=I/12
90 CLS
100 LOCATE 13,8
110 PRINT I;"INCHES =";F;"FEET"
120 LOCATE 23,1
130 INPUT"PRESS <ENTER> TO CONTINUE";EN$

140 GOTO 60
```

Program 2. Inches to Feet.

ting it to convert feet to yards, yards to feet, inches to yards, or yards to inches. If you want to go further, you can even convert miles to feet, inches, or yards, or vice versa. Of course, pints to gallons, liters to quarts, centimeters to inches, and any number of other conversions are also easily performed using this simple program. All that's really necessary is to change your prompt and print lines and, of course, the formula contained in line 80.

Program 3: Fahrenheit to Celsius

This program performs another conversion. You are asked to input a temperature in degrees Fahrenheit. The computer will then display the same temperature in degrees Celsius. Line 70 prompts you to input the temperature in Fahrenheit, while line 80 contains the formula for the conversion. To convert Fahrenheit to Celsius, subtract 32 degrees from the Fahrenheit temperature and multiply the result by 5/9. The numeric variable X is assigned the Celsius value. The screen is cleared in line 90 and the LOCATE statement in line 100 moves the graphic cursor to a point just left of the center of the text screen. Line 110 prints the Celsius value followed by the iden-

```
10 REM FAHRENHEIT TO CELSIUS CONVERSION
20 REM COPYRIGHT FREDERICK HOLTZ
30 CLS
40 SCREEN 0
50 WIDTH 40
60 KEY OFF
70 INPUT"FAHRENHEIT TEMPERATURE";F
80 X=5/9*(F-32)
90 CLS
100 LOCATE 14,10
110 PRINT X"DEGREES CELSIUS"
```

Program 3. Fahrenheit to Celsius.

```
10 REM CELSIUS TO FAHRENHEIT CONVERSION
20 REM COPYRIGHT FREDERICK HOLTZ
30 CLS
40 SCREEN 0
50 WIDTH 40
60 KEY OFF
70 INPUT"CELSIUS TEMPERATURE";C
80 X=(C/(5/9)+32)
90 CLS
100 LOCATE 14,10
110 PRINT X"DEGREES FAHRENHEIT"
```

Program 4. Celsius to Fahrenheit.

tifier "DEGREES CELSIUS". The program then terminates. You can take a clue from the previous two programs and branch back to an earlier line with an appropriate GOTO statement.

Program 4: Celsius to Fahrenheit

This program does the opposite of the previous program. Here, you input a temperature in Celsius and the computer outputs it in degrees Fahrenheit. The program is almost exactly the same as the previous one, except for the modified input prompt and print designator. The formula in line 80 converts Celsius (C) to Fahrenheit. Numeric variable X then contains the Fahrenheit value.

The four previous programs used very simple formulas to convert from one form of measurement to another. However, the computer is also capable of pushing values through even highly complex formulas, as the next few programs will demonstrate.

Program 5: Mortgage Payment

This program will allow you to input a sum of

```
10 REM MORTGAGE PAYMENT
20 REM COPYRIGHT FREDERICK HOLTZ
30 CLS
40 KEY OFF
50 SCREEN 0
60 WIDTH 40
70 INPUT"ENTER THE AMOUNT TO BE FINANCED
";A
80 CLS
90 INPUT"ENTER THE INTEREST RATE";I
100 CLS
110 INPUT"HOW MANY YEARS";Y
120 Y=Y*12
130 CLS
140 I=I/100
150 I=I/12
160 MA=(I/((I+1)^(Y)-1)+I)*A
170 PRINT"MONTHLY PAYMENT =";MA
```

Program 5. Mortgage Payment.

money to be financed, the interest rate, and the number of years over which payment is to be made. It will then tell you what your monthly mortgage payment will be. This is a highly useful program, because it can be used for any mortgage value, any interest rate, and any number of years. Chances are you've had to sort through mortgage payment books to figure out how much you would have to pay to buy a house, car, boat, computer, or some other expensive thing. This program takes all the trouble out of such calculations.

Here's how it works. Line 70 prompts you to enter the amount to be financed. (Remember, you cannot use commas—as in 3,123,20—to delineate thousands when inputting an amount. Decimals, of course, are OK.) When this value has been input and Enter is pressed, the value is committed to numeric variable A. Line 80 then clears the screen, and line 90 prints another prompt telling you to enter the interest rate. Do not enter the interest rate as a decimal value. In other words, enter an interest rate of 13% as 13, or an interest rate of 13 1/2% as 13.5. (More on this in a bit.)

Again, the screen is cleared upon pressing Enter, and the interest rate is committed to numeric variable I. Line 10 uses another INPUT statement, prompting the user to input the number of years over which the loan is to be amortized. This value is committed to numeric variable Y. Remember, A represents the amount of mortgage, I represents the interest, and Y represents the number of years.

Now it's time for some conversions. Line 120 reassigns, to Y, the value of Y times 12, which simply converts your original input from years to months. The reason interest was input as a whole number is found in line 140. Here, I is reassigned the value of I divided by 100, which automatically converts your interest rate input to a decimal for the computer to use. Line 150 makes a further conversion of I, dividing it by 12 to give the interest rate for each month. The value of I that the computer eventually uses in the formula (line 160) is no longer in the same form it was when you first entered it in line 90.

There is a fairly complex formula in line 160 which takes the value of A, I, and Y and converts them to the amount of your monthly payment. This is assigned to the numeric variable MA (for monthly amount). Then line 170 prints "MONTHLY PAYMENTS EQUAL" followed by MA, which contains the monthly payment.

If you had to perform all of these calculations on a pocket calculator, there would be a good chance of error, and to do it on paper would be quite cumbersome. However, once you have this program input to the computer you can simply enter the values, and almost instantly come up with the monthly amount of money you've got to fork over.

Program 6: Maximum Loan Affordable

This program is similar in some ways to the previous one, except that it will tell you how much money you can afford to borrow, based on the number of monthly payments and the monthly amount you wish to pay. It is necessary to know the interest rate at which the money is to be borrowed, but the program does the rest.

The number of monthly payments is put in via the prompt in line 70. This value is assigned to variable A. The screen is cleared by the CLS statement in line 80 and a new prompt appears asking for the interest rate. This value is assigned to variable I. Once again, the interest rate is not input as a decimal, because line 100 makes a conversion automatically. For instance, it converts 13% annual interest by dividing it by 1200 (100 times 12), effectively yielding the monthly interest in decimal form in one operation. Notice that line 100 simply reassigns the value of numeric variable I.

In line 120, the user is asked to input the maximum affordable monthly payment, which value is assigned to numeric variable M. Line 140 contains a formula that converts these values into a number that represents the amount of money that can be borrowed according to the previously specified user inputs, assigning it to variable X. Since this value will usually come out to a decimal, line 150 is used to convert X to an integer which will simply be the whole number minus any fractional values. For practicality's sake, a maximum affordable amount of 2,001.4388 is best expressed as 2,001.

Line 160 prints the value that can be borrow-

```
10 REM PROGRAM TO CALCULATE THE MAXIMUM
LOAN AFFORDABLE
20 REM BASED UPON CURRENT INTEREST AND A
BILITY TO PAY
30 REM COPYRIGHT FREDERICK HOLTZ
40 CLS
50 KEY OFF
60 SCREEN 0
70 INPUT"HOW MANY MONTHLY PAYMENTS CAN Y
OU MAKE";A
80 CLS
90 INPUT"WHAT IS THE CURRENT INTEREST RA
TE";I
100 I=I/1200
110 CLS
120 INPUT"WHAT IS THE MAXIMUM MONTHLY PA
YMENT YOU CAN AFFORD";M
130 CLS
140 X=M*(1-(1+I)^-A)/I
150 X=INT(X)
160 PRINT"YOU CAN AFFORD TO BORROW UP TO
 $";X
170 END
```

Program 6. Maximum Loan Affordable.

ed on the screen, after which the program terminates. This program is short, sweet, and to the point. Total input time should be about five minutes.

Program 7: Annuity Calculation

An annuity may be thought of as an income (retirement or otherwise) yielded from a principal drawing annually compounded interest. This program will calculate the annuity, based upon the principal, interest rate, and number of years over which the annuity is to be drawn. This program is longer than the previous two which dealt with financial calculations, but only because a few additional user conveniences have been included. In line 80, there is a GOSUB to line 410. Lines 420 through 500 explain the program by printing the instructions on the screen; line 510 contains the mandatory RETURN statement which sends the computer back to line 90. Lines 90 through 110 contain PRINT statements that simply separate the prompt in line 120 from the previously printed instructions. Incidentally, the COLOR statement in line 50 causes the screen background to switch to a blue with black lettering—a colorful effect that is quite pleasing to the eye.

Line 120 tells the user to "press any key" to continue. Previous programs used the INPUT statement with a prompt of "DO YOU WISH TO CONTINUE(Y/N)". This would require the input of either a Y or N, followed by the Enter key. This is fine, but the input letters *must* be capitals as specified in the program. (If you specified lowercase letters in the program, the program would respond only when a lowercase input occurred.) Alternately, we could input the needed lines to branch to the correct points based upon either an uppercase or lowercase letter, as in:

```
10 REM ANNUITY CALCULATION PROGRAM
20 REM COPYRIGHT FREDERICK HOLTZ
30 REM CLS
40 SCREEN 0
50 COLOR 0,1
60 WIDTH 40
70 KEY OFF
80 GOSUB 410
90 PRINT
100 PRINT
110 PRINT
120 PRINT "PRESS ANY KEY TO CONTINUE"
130 F$=INKEY$
140 IF F$="" THEN 130
150 CLS
160 LOCATE 14,8
170 INPUT "ENTER THE INTEREST RATE";I
180 I=I/100
190 CLS
200 LOCATE 14,8
210 INPUT "HOW MANY YEARS ";YR
220 CLS
230 LOCATE 14,8
240 INPUT "WHAT IS THE PRINCIPLE ";PR
250 CLS
260 X=(1+I)^YR
270 Y=I*X
280 Z=X-1
290 ZZ=Y/Z
300 ANNU=PR*ZZ
310 LOCATE 14,10
320 PRINT "ANNUITY=$";ANNU
330 LOCATE 22,5
340 COLOR 31
350 PRINT "DO YOU WANT TO ENTER AGAIN?(Y
/N)
360 COLOR 0,1
370 F$=INKEY$
380 IF F$="Y" THEN COLOR 0,1:GOTO 150
390 IF F$="N" THEN COLOR 7,0:CLS:END
400 GOTO 370
410 CLS
420 PRINT"THIS PROGRAM WILL CALCULATE TH
E ANNUITY"
430 PRINT"YIELDED BY A PRINCIPLE DRAWING
 ANNUALLY"
440 PRINT"COMPOUNDED INTEREST. YOU MUST
INPUT THE"
460 PRINT"INTEREST RATE, AMOUNT OF PRINC
IPLE, AND"
470 PRINT"THE NUMBER OF YEARS OVER WHICH
 THE IN-"
480 PRINT"TEREST WILL BE COMPOUNDED. THE
 COMPUTER"
500 PRINT"WILL PROVIDE YOU WITH THE ANNU
ITY."
510 RETURN
```

Program 7. Annuity Calculation.

```
IF A$="Y" OR A$="y" THEN 1000
```

However, for many purposes this is not necessary. As an aid to foolproofing the program, line 130 uses the INKEY$ variable in GWBASIC to allow any key to be pressed to continue execution, as indicated by the prompt in line 120. Line 130 assigns to F$ the value of the keyboard input. Line 140 simply tells the computer to go back to line 130 if the keyboard input equals "". The double quotes with nothing in between indicate the null string, a value of nothing at all. Lines 130 and 140 will continue to be executed in a loop until there is some type of keyboard input. It makes no difference which key is pressed, since any one will make F$ unequal to "". A later sequence in this same program shows how INKEY$ can be used just like the INPUT statement, but without having to press Enter after each input.

Line 150 clears the screen, and the LOCATE statement is then used to place the text cursor at row 14, column 8. Line 170 uses the INPUT statement to prompt the user to enter the interest rate. This message is printed at the center of the screen due to the previous LOCATE statement. The INPUT statement is necessary here, as opposed to the INKEY$ variable, since the latter is restricted in practical use to a *single character* input from the keyboard. In most cases the interest rate today will consist of a two-digit number!

Line 180 converts the interest to decimal form, lines 200 and 210 print another prompt at the center of the screen asking for the number of years over which the interest is to be drawn, and this value is then assigned to YR. Lines 230 and 240 do the same thing for the principal prompt. Lines 260 through 300 contain the formulas that convert the user-supplied information to the annuity payment. This final value is assigned to the numeric variable ANNU in line 300.

Lines 310 and 320 print the answer at the center of the screen. Now comes another interesting part user appeal feature. The LOCATE statement in line 330 positions the text cursor at row 22, column 5, printing the message (line 350) at the bottom of the screen. Note, however, that line 340 contains the COLOR statement followed by the number 31. In text mode this value will cause any other printed matter to be displayed in white, flashing. That's right. Anything that's printed *after* COLOR 31 is used will flash on the screen, but all previously printed lines will be unaffected. Thus, the message contained in line 350 will flash on the screen for increased user attention. Line 370 uses INKEY$ again, but lines 380 and 390 perform a test, since we are now looking for an input of either an upper case Y or upper case N. If it gets neither of these letters, a continuous loop is formed. Line 360 resets the color to the one used at the start of the program, but this will have no effect on the flashing line printed by program line 350. The program will lock up at this point until a Y or N is input. If the value is Y, there is a branch to line 150, where the program effectively starts all over again. However, if the value is N, color is reset to the original value before the program was run, the screen is cleared, and the program ends. It is necessary to reset the color to white-on-black before ending the program, or the screen will still be in black-on-blue mode and will have to be manually reset.

There you have it—a very interesting and informative program (from a programmer's standpoint) that builds in a lot of quality display features.

The previous three programs have dealt with financing, interest, etc. As your experience increases in programming the PC-6300, you may wish to combine all three programs into a single one. You can build in a menu that will allow the user to select which program portion is to be used.

Program 8: Leap Year Calculation

Why a program to calculate Leap Year? Well, one reason is that most books that include simple programs for any computer usually have one, but my main reason is that it allows me to explain that Leap Year does not occur every four years—contrary to what most of us might suspect. It usually does, but not always. For instance, the year 1896 was a Leap Year and so was 1892, four years

```
10 REM PROGRAM TO CALCULATE LEAP YEAR
20 REM COPYRIGHT FREDERICK HOLTZ
30 CLS
40 SCREEN 0
50 WIDTH 40
60 KEY OFF
70 INPUT"YEAR";Y
80 CLS
90 IF Y/4=INT(Y/4) AND Y/100<>INT(Y/100)
 OR Y/400=INT(Y/400) THEN 100 ELSE 140
100 CLS
110 LOCATE 14,10
120 PRINT Y;"IS A LEAP YEAR"
130 END
140 CLS
150 LOCATE 14,10
160 PRINT Y;"IS NOT A LEAP YEAR"
170 END
```

Program 8. Leap Year Calculation.

earlier. However, the year 1900 was not a Leap Year but 1904 was, so there was an eight-year span (from 1896 to 1904) in which no Leap Year occurred.

A Leap Year is one which is evenly divisible by four and not evenly divisible by 100, or is any year evenly divisible by 400. (Most people think it's any year evenly divisible by four, but this is not true.) Leap Year is used to make our calendars match the Earth's orbit around the sun, which takes a little more than 365 days. Usually we make an adjustment to correct this every four years, but to avoid over-correcting we skip a Leap Year once in a while. The year 1896 is a Leap Year because it is evenly divisible by four and not evenly divisible by 100. However, the year 1900 is evenly divisible by four, *but* it is *also* evenly divisible by 100. Therefore, 1900 would not be a Leap Year *unless* it were also evenly divisible by 400—and simple math will tell you that it is not. One might also think that this occurs every century, but it doesn't. For instance, 1996 will be a Leap Year, and so will the year 2000. (Here we go again.) The year 2000 is evenly divisible by four and is also evenly divisible by 100—*but* the year 2000 is also evenly divisible by 400, and any year that is evenly divisible by 400 is always a Leap Year regardless of any previous criteria. The next time a Leap Year skip will occur, then, will be between 2096 and 2104.

If you're totally confused by now, that's all right, because this program will do all the work for you. Line 90 contains the formula, using the IF-THEN statement plus the logical operators AND and OR. Line 90 says that IF the year Y divided by 4 is equal to the *integer* of the year Y divided by 4 (i.e., the division comes out even) AND IF the year divided by 100 is not equal to the integer of the year divided by 100 is not equal to the integer of the year divided by 100—OR the year divided by 400 is equal to the integer of the year divided by 400—THEN GOTO line 100, which sets up the "LEAP YEAR" screen print. The ELSE portion of the statement in line 90 simply tells the computer to branch to line 140 if this set of conditions is not true. Line 140 sets up the routine to identify the year as one that is not a Leap Year.

Let's discuss line 90 further. The AND logical operator means that the statement preceding it and the one following it must both be true. The OR operator means that, regardless of the previous two tests, if the statement following it is true, then the branch takes place. In other words, for a branch to line 100 to print LEAP YEAR on the screen, the first two tests must be true OR the last test must be true. This is an excellent example of the use of logical operators in GWBASIC. As your experience develops, you will find that you will be using the logical operators for many purposes.

Program 9: Countdown Timer

We now know that the PC-6300 contains an internal clock that can be used to display the time of day and also the date. This program, however uses the clock as a countdown timer that can come in handy for games and other similar activities that require the user to perform some function within a set period of time. This program will display a preset minutes value on the screen, and then count second-by-second to this preset value. When time runs out, a siren-like audio effect will be emitted from the computer.

Here's how the program works. Line 70 prompts you to input the number of minutes you wish to use for your count. If you prefer to use seconds, simply input minutes as a decimal fraction (i.e., 0.5 minutes equals 30 seconds). When this value has been input, the screen is cleared and line 90 assigns X$ the value of "00:00:00." Line 100 resets the computer's clock to X$, or zero. Line 110 begins a FOR-NEXT loop that counts from 1 to 3450 times I. (By experimentation I found that this value for the loop would cause line 130 to be reprinted approximately every second. I'll explain this further a little later.)

Line 120 positions the text cursor near the center of the screen, and line 130 prints the value of TIME$. Remember, it was initially set to zero, but every second it will count upward. Line 140 causes the loop to recycle. Now, with a top value of 3450 times I (the number of minutes), lines 120 and 130 will be executed about every second. This causes the clock display at the center of the screen

```
10 REM TIMER PROGRAM
20 REM COPYRIGHT FREDERICK HOLTZ
30 SCREEN 0
40 CLS
50 KEY OFF
60 WIDTH 40
70 INPUT"HOW MANY MINUTES";I
80 CLS
90 X$="00:00:00"
100 TIME$="11:11:11":TIME$=X$
110 FOR X=1 TO 3450*I
120 LOCATE 14,17
130 PRINT TIME$
140 NEXT
150 FOR XX=1 TO 15
160 FOR X=1200 TO 1500 STEP 40
170 SOUND X,1
180 NEXT X
190 NEXT XX
200 CLS
210 GOTO 70
```

Program 9. Countdown Timer.

to be updated once each second; the numerals, therefore, count upward in increments of one. You may find that it's necessary to make some slight adjustments to the 3450 value for your computer, but this value should be close. The loop times out at the time specified in line 70.

When this occurs, lines 150 onward are executed. Line 150 sets up a FOR-NEXT loop that counts from 1 to 15. Line 160 contains a nested loop that counts from 1200 to 1500 in steps of 40. The GWBASIC SOUND statement outputs the frequencies set by the loop in line 160. As the loop is cycled, the frequency steps up. Each time the nested loop is completely executed, a "whoop" is produced. Since the outer loop counts 1 to 15, 15 whoops are heard.

When the siren times out, line 200 clears the screen and the program starts all over again, due to the branch contained in line 210. You can now input another time value.

This program is an interesting study in itself, but it will rarely be used alone. It will most often be incorporated as a part of a larger program.

Program 10: Backward Screen Program

This program has no practical value (that I know of), but it does contain some entertainment possibilities and is interesting from the standpoint of methodology. On the PC-6300, like any other computer, any information typed in via the keyboard is displayed from left to right starting at the top of the screen. This program, however, will cause any keyboard input to be displayed from right to left; Fig. 4-1 shows an example.

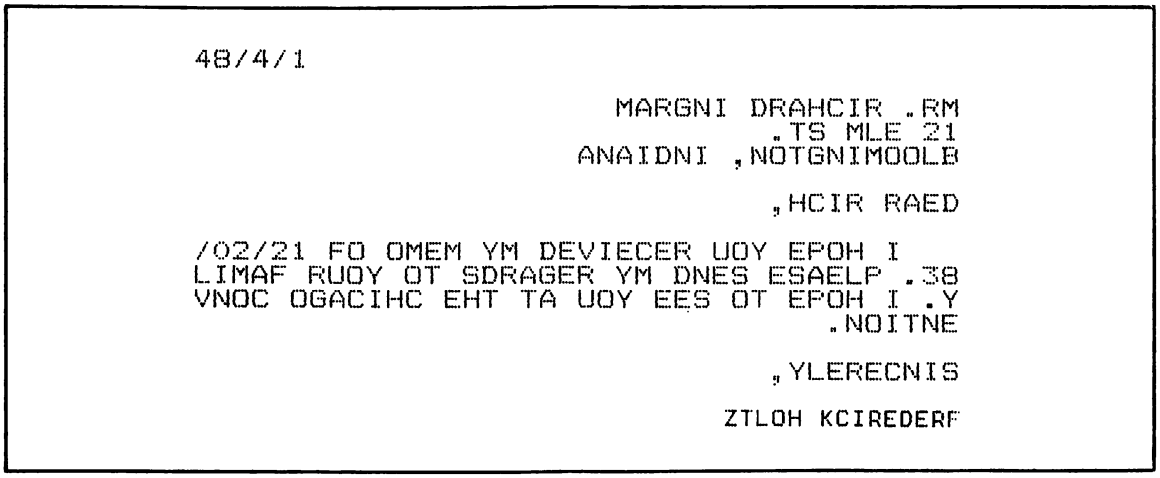
Here's how the program works. The screen is first initialized, and then line 60 assigns the variable Y the initial value of 1. Lines 70 and 80 set up an "ANY KEY" routine as described in the annuity program, forming a continuous loop until any key is pressed. When this is done, line 90 first sets the text cursor at Y,40-X. For the first letter of input, Y is equal to 1, since Y was initially assigned this value in line 60. Since no X value has been assigned X is automatically equal to 0. The first input character will therefore be displayed in row 1,

```
80 10 REM BACKWARD SCREEN PROGRAM
20 REM COPYRIGHT FREDERICK HOLTZ
30 CLS
40 KEY OFF
50 WIDTH 40
60 Y=1
70 A$=INKEY$
80 IF A$="" THEN 70
90 LOCATE Y,40-X
100 PRINT A$
110 IF A$=CHR$(13) THEN Y=Y+1:X=-1
120 X=X+1
130 IF X=40 THEN X=0:Y=Y+1
140 IF Y=24 THEN END
150 GOTO 70
```

Program 10. Backward Screen Program.

column (40 minus 0), which is the upper right-hand corner of the screen. Line 100 prints the value of A$, which is the letter you input. Line 110 checks to see if A$ was a carriage return (Enter), which in the character set is ASCII code 13. The CHR$ function converts ASCII code 13 to what the computer recognizes as a carraige return. If you input a carriage return via the keyboard, A$ is equal to CHR$(13). Line 110 tells the computer to increase Y by 1 if A$ is a carriage return, and to assign X a value of -1. Line 120 then increments X by +1. When this value is subtracted from 40 (line 90), the text cursor is positioned one step toward the left anytime a key is pressed—unless the key is the carriage return. In that case, the -1 assignment to X (line 110) combined with the +1 assignment to X (line 120) causes the value of X to revert to 0. In other words, whenever you hit enter, the text cursor goes down one line and returns to the right hand side of the screen—just the reverse of normal operation.

Lines 130 and 140 perform other tests to keep



```
48/4/1

                    MARGNI DRAHCIR .RM
                             .TS MLE 21
                 ANAIDNI ,NOTGNIMOOLB

                             ,HCIR RAED

/02/21 FO OMEM YM DEVIECER UOY EPOH I
LIMAF RUOY OT SDRAGER YM DNES ESAELP .38
VNOC OGACIHC EHT TA UOY EES OT EPOH I .Y
                                .NOITNE

                            ,YLERECNIS

                        ZTLOH KCIREDERF
```

Fig. 4-1. Screen display of the Backward Screen Program.

the cursor from running off the screen. (Actually it can't; an error message is generated when an illegal cursor value is used, causing the program to end.) When the cursor has counted all the way to the far left hand side of the screen, the value of X will now be equal to 40. Line 130 tells the computer to reassign X to 0 when it is equal to 40. At the same time, Y is incremented by 1. Line 140 checks for an end-of-page condition. When Y is equal to 24 (near the bottom of the screen), the program ends.

With this program, you can display any member of the standard character set on the screen. Don't try to use the numeric keypad to the right of the keyboard or the backspace key, however, or unusual characters will be printed. In other words, if you make a typing mistake you're out of luck.

Experiment with this program a bit to see what you can come up with. Here's an interesting switch. Change lines 60, 110, 130, and 140 as follows:

```
 60  Y=22
110  IF A$=CHRS(13) THEN
     Y=Y -1:X=-1
130  IF X=40 THEN X=0:Y=Y-1
140  IF Y=0 THEN END
```

When you run the revised program, the text will begin appearing at the bottom right of the screen, still advancing from left to right. Each time you reach the end of a line at the far left of the screen or press a carriage return, the text cursor will advance upward by one line and reset to the right side of the screen. By making a few simple changes, you can also create a program that prints letters from top to bottom, or vertically instead of horizontally. The key is the LOCATE statement found in line 90 and the proper assignment of X,Y values.

Program 11: Printer/Typewriter

If you have a printer, this program should be interesting. It will display text in normal fashion (left to right) on the screen *and* on your printer. Line 60 sets the printer width to 40 characters to match the screen width. Lines 70 and 80 contain the familiar "ANY KEY" routine. When any key is pressed, it is printed on the screen in line 90. Line 100 checks for a carriage return. Then, line 110 uses the LPRINT statement to cause the printer to type A$. Line 120 branches back to line 70 so that another character may be input. The LPRINT statement is used just like PRINT, except the output goes to the printer instead of to the screen. Here line 90 outputs A$ to the printer. If you have the extra memory option and the display enhancement that comes with it, you can change line 50 to WIDTH 80 and also change the value 40 (in line 60) to 80 for a full 80-column display on both screen and printer.

```
10 REM PRINTER/TYPEWRITER
20 REM COPYRIGHT FREDERICK HOLTZ
30 CLS
40 KEY OFF
50 WIDTH 40
60 WIDTH"LPT1:",40
70 A$=INKEY$
80 IF A$="" THEN 70
90 PRINT A$;
100 IF A$=CHR$(13) THEN PRINT
110 LPRINT A$;
120 GOTO 70
```

Program 11. Printer/Typewriter.

Program 12: Trip Calculator

This is a program designed to supply you with trip planning information—specifically trip time, fuel consumption, total mileage traveled, and the number of times it will be necessary to refill your tank. (It assumes you are starting out with a full tank.)

Lines 70 through 130 tell you about the program by printing instructions. As the program moves on, lines 200-210 ask for your average fuel consumption for in-town driving, in miles per gallon. At line 220 the INPUT statement assigns this figure to variable A. Lines 250 through 270 allow you to input your MPG figure for highway driving, assigning it to variable B.

After line 280 clears the screen, lines 290 through 310 allow you to input your fuel tank capacity in gallons. This is committed to numeric variable C. In line 320 the BEEP statement causes a short 1000 Hz tone to be produced, telling the user to examine the information printed by lines 340 through 370 for correctness; lines 400-430 prompt the user to verify. If it is not (Enter N), there is a branch to line 190, where you are asked for the information again. If it is correct there is a branch to line 490, and the program continues. Line 140 checks for either an uppercase N or Y. If you've input something else, lines 440 through 480 indicate an incorrect input. Line 480 branches to line 310, where the previously input information is displayed again.

Assuming the information is correct, lines 490 through 550 clear the screen and explain the information the user will be asked for next. Line 570

```
10 REM PROGRAM TO DETERMINE TRIP TIME,FU
EL CONSUMPTION,TOTAL MILEAGE TRAVELED.
20 REM COPYRIGHT FREDERICK HOLTZ
30 CLS
40 KEY OFF
50 SCREEN 0
60 WIDTH 40
70 PRINT"THIS PROGRAM WILL ACT UPON INPU
T INFOR-"
80 PRINT"ION TO ALLOW YOU TO PLAN A TRIP
."
90 PRINT"IT WILL BE NECESSARY FOR YOU TO
 KNOW"
100 PRINT"YOUR PLANNED ROUTE, THE MILEAG
E OF EACH TRIP LEG, YOUR AUTOMOBILE'S IN
-TOWN"
110 PRINT"AND HIGHWAY FUEL CONSUMPTION F
IGURES"
120 PRINT"AND A FEW OTHER FACTS. THE FIN
AL SCREEN PRINT OUT WILL SERVE AS A USEF
UL GUIDE"
130 PRINT"FOR TRIP PLANNING."
140 PRINT
150 PRINT
160 PRINT"PRESS ANY KEY TO CONTINUE"
170 A$=INKEY$
180 IF A$=""THEN 170
190 CLS
200 PRINT"WHAT IS YOUR AVERAGE MPG FIGUR
E FOR"
```

Program 12. Trip Calculator. (Continued to page 73.)

```
210 PRINT"IN-TOWN DRIVING"
220 INPUT A
230 PRINT
240 PRINT
250 PRINT"WHAT IS YOUR AVERAGE MPG FIGUR
E FOR"
260 PRINT"HIGHWAY DRIVING"
270 INPUT B
280 CLS
290 PRINT"HOW MANY GALLONS OF FUEL WILL
YOUR TANK HOLD ON FILL-UP "
300 INPUT C
310 CLS
320 BEEP
330 LOCATE 11,25
340 PRINT TAB(25)"YOUR IN-TOWN GAS CONSU
MPTION IS"A"MPG"
350 PRINT TAB(25)"YOUR HIGHWAY CONSUMPTI
ON IS"B"MPG"
360 PRINT TAB(25) "YOUR AUTOMOBILE WILL
HOLD A"
370 PRINT"MAXIMUM OF"C"GALLONS."
380 PRINT
390 PRINT
400 INPUT"IS THIS INFORMATION CORRECT (Y
/N)",B$
410 IF B$="N" THEN 190
420 IF B$="Y" THEN 490
430 IF B$<>"N" AND B$<>"Y" THEN 440
440 CLS
450 PRINT"YOU HAVE NOT ENTERED CORRECTLY
!!!"
460 PRINT
470 PRINT
480 GOTO 310
490 CLS
500 PRINT"YOU WILL NOW BE ASKED TO SUPPL
Y INFORM-"
510 PRINT"ATION ABOUT YOUR TRIP. YOU WIL
L NEED"
520 PRINT"TO EXAMINE YOUR INTENDED ROUTE
 ON AN"
530 PRINT"APPROPRIATE ROAD MAP."
540 PRINT
550 INPUT"PRESS ENTER WHEN READY TO CONT
INUE",Q$
560 CLS
570 PRINT"FROM WHICH TOWN OR CITY WILL Y
OUR TRIP"
580 PRINT"BEGIN"
590 INPUT D$
600 PRINT
```

```
610 PRINT
620 PRINT"AT WHICH TOWN OR CITY WILL YOU
R TRIP END "
630 INPUT E$
640 CLS
650 PRINT"FROM YOUR ROAD MAP, TYPE IN TH
E TOTAL"
660 PRINT"NUMBER OF TRIP MILES."
670 INPUT M
680 PRINT
690 PRINT
700 PRINT"HOW MANY MILES INVOLVE INTERST
ATE HIGH-"
710 PRINT"WAYS OR LIMITED ACCESS PRIMARY
 ROADS"
720 INPUT I
730 PRINT
740 PRINT
750 PRINT"HOW MANY TRIP MILES WILL BE SP
ENT ON"
760 PRINT"SECONDARY ROADS "
770 INPUT S
780 PRINT
790 PRINT
800 PRINT"HOW MANY TRIP MILES WILL INVOL
VE DRIVING"
810 PRINT"THROUGH TOWNS AND CITIES"
820 INPUT R
830 IF (I+S+R)<M THEN 860
840 IF (I+S+R)>M THEN 920
850 IF (I+S+R)=M THEN 950
860 CLS
870 PRINT"YOU HAVE ENTERED LESS THAN THE
 TOTAL"
880 PRINT"TRIP MILEAGE!!!"
890 PRINT
900 INPUT"PRESS <ENTER> TO TRY AGAIN.";A
$
910 GOTO 640
920 CLS
930 INPUT"YOU HAVE ENTERED MORE THAN THE
 TOTAL TRIP MILEAGE!!! PRESS ENTER TO TR
Y AGAIN",A$
940 GOTO 640
950 Y=(I/55)+(R/18)+(S/45)
960 W=(I/B)+(S/B*1.1)+(R/A)
970 Z=W/Y
980 X=W/C
990 IF X<1 THEN X=0
1000 CLS
1010 BEEP
```

```
1020 LOCATE 8,1
1030 PRINT"YOUR TRIP FROM "D$" TO "E$
1040 PRINT"WILL TAKE"Y"HOURS."
1050 PRINT
1060 PRINT
1070 PRINT"TOTAL FUEL CONSUMPTION WILL B
E"W"GALLONS."
1080 PRINT
1090 PRINT
1100 PRINT"YOU WILL HAVE TO REFILL"X"TIM
ES."
1110 PRINT
1120 PRINT
1130 PRINT"YOUR AVERAGE FUEL CONSUMPTION
 WILL BE"
1140 PRINT Z"GALLONS PER HOUR."
```

asks for the town or city of departure, which is committed to variable D$. Line 620 then asks for the name of the town or city of destination, and commits to variable E$.

From this point on, the user is asked to input the total number of trip miles, as well as a breakdown of how many are highway, secondary road, and in-town driving. These values are committed to numeric variables M, I, S, and R, respectively; lines 830 through 850 then make certain the number of miles in the breakdown equals the total number of trip miles represented by variable M. If the figures don't agree there is a branch back to line 640, where you are prompted to input the information again. Since this is a practical program producing important information, it is necessary to program adequate check functions to assure accurate input. (Providing check functions for your programs is a useful habit to acquire.)

Assuming all information is correct, there is a branch to line 950, where the formulas come into play. Line 950 assigns Y a value equivalent to the total time for the entire trip. Here I have assumed that the average speed will be 55 mph for highway driving, 18 mph intown, and 45 mph for secondary roads. You can adjust this line to match your own personal driving experience, but these figures should suffice. Line 960 assigns to W the total fuel consumption. Line 970 divides W by Y, which yields fuel consumption in gallons per hour. Line 980 assigns to X the total gallons divided by the number of gallons your tank will hold. This means X will represent the number of times you will have to refill your tank. Line 990 simply assigns X the value 0 if X is less than 1. (You can't refill your tank a fractional number of times.) Finally, line 1000 clears the screen. The user hears another beep, and the information is neatly displayed. Figure 4-2 shows a sample printout on the AT&T PC-6300 screen.

This program is certainly not meant to be used as an *exact* indicator of the trip values

```
YOUR TRIP FROM ELM CITY TO CHICAGO
WILL TAKE 11.49192 HOURS.


TOTAL FUEL CONSUMPTION WILL BE
22.57429 GALLONS.


YOU WILL HAVE TO REFILL 2 TIMES.


YOUR AVERAGE FUEL CONSUMPTION WILL
BE 1.964362 GALLONS PER HOUR.
```

Fig. 4-2. Data printout of the trip calculator.

displayed. No computer can do this because there are too many variables involved. If it rains, for instance, or if you run into construction or get lost, then all factors will change. This program is useful, however, because it can indicate reasonable expectations for a trip starting at point A and ending at point B. By aiding you in your planning, it may make any trip more enjoyable.

Program 13: Alphabetizing

The last program in this chapter, lucky 13, is extremely useful. As a matter of fact, I use it to complete almost every book I write because it handles the alphabetizing necessary for indexes and glossaries. Using this program, you can enter up to 1000 words or phrases in any order. The computer will then sort them and print them on the screen in alphabetical order. Now, from a practical standpoint, it would take even the PC-6300 several hours to alphabetize several thousand items; furthermore, your screen cannot display this many items at one time. This program is more useful for alphabetizing short lists, although you could modify it to dump its output on the printer.

After the screen is initialized, lines 70 through 90 tell the user what to do. Two string arrays are established in lines 120 and 130, each able to contain up to 1000 items. The user is then prompted to input the words to be alphabetized. Line 150 contains a simple count routine that increments I by 1 each time a word is input; line 160 contains the INPUT statement that assigns the word to the A$ array. Line 180 then branches back to line 150, allowing another word to be input. Program line 170 contains the endless loop exit routine. To complete your list, type and enter the word END. This must be in capital letters, although all other words may be input in either upper, lower, or mixed case. When the exit word is input, there is a branch to line 190, which assigns to variable N the value the count variable I, minus 1. This minus 1 is necessary because the word END caused I to be stepped by 1, but it is not a word to be alphabetized.

Line 200 starts a FOR-NEXT loop which simply counts from 1 to the value of N, which is

```
10 REM ALPHABETIZING PROGRAM
20 REM COPYRIGHT FREDERICK HOLTZ
30 CLS
40 SCREEN 0
50 WIDTH 80
60 KEY OFF
70 PRINT"ENTER EACH WORD AS
   REQUESTED."
80 PRINT"WHEN LIST IS COMPLETE,
   TYPE 'EN
D'."
90 PRINT"ENTER UP TO 1000 WORDS"
100 PRINT
110 PRINT
120 DIM A$(1000)
130 DIM J$(1000)
140 PRINT"TYPE WORDS TO BE
    ALPHABETIZED:
"
150 I=I+1
160 INPUT A$(I)
170 IF A$(I)="END" GOTO 190
180 GOTO 150
190 N=I-1
200 FOR I=1 TO N
210 J$(I)=A$(I)
220 NEXT I
230 CLS
240 FOR I=1 TO N
250 PRINT A$(I)", ";
260 NEXT I
270 PRINT
280 PRINT
290 PRINT"PRESS'ENTER'TO CONTINUE"
300 INPUT Z$
310 CLS
320 LOCATE 12,9
330 PRINT"COMPUTING ---PLEASE
    STAND BY"
340 FOR P=1 TO N-1
350 FOR I=1 TO N-P
360 IF J$(I)<= J$(I+1) GOTO 400
370 X$=J$(I)
380 J$(I)=J$(I+1)
390 J$(I+1)=X$
400 NEXT I
410 NEXT P
420 CLS
430 FOR I=1 TO N
440 PRINT J$(I)
450 NEXT I
460 END
```

Program 13. Alphabetizing.

equivalent to the number of words input. Line 210 simply reads all of the words in A$ into J$, so that when the loop is exited in line 220, the items in J$ are identical to those in A$. The screen is then cleared and another loop is entered, with lines 240 through 260 simply displaying the contents of the array on the screen. When you press Enter, the screen is again cleared and a prompt appears at the center of the screen, telling you that the computer is processing the list. For short lists this prompt is not really necessary, but for longer lists it lets the user know that the computer is doing its job and hasn't locked up.

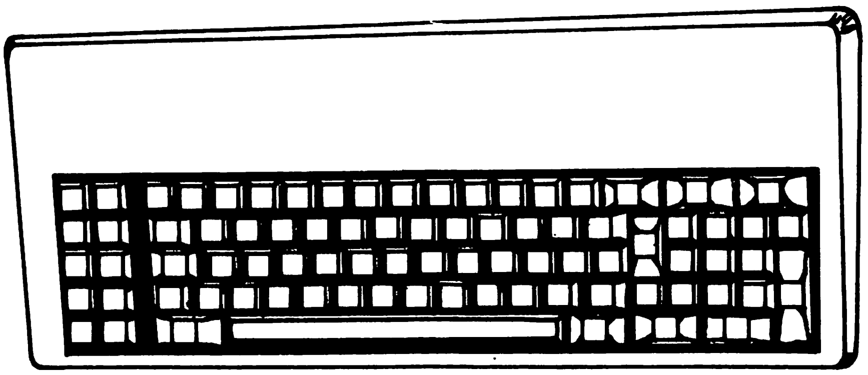
Lines 340 through 410 contain the sorting routine. While the number of program lines is short, the operation is complex. Here's what happens. The computer reads each item in the array, assigning the first item the lowest value. If the next item is of higher value, it's left in place, but if it is of lower value than the first item, the two values are interchanged in the array. The computer considers an A to be smaller than a B, a B smaller than a C, and so on. The computer continues to pull out words and compares them with those read previously—swapping the list around, two items at a time, until the order is complete.

Line 420 then clears the screen and the entire contents of the J$ array are displayed on the screen. This is accomplished by lines 430 through 450, which count I from 1 to the number of elements contained in the array. The PRINT statement in line 440 prints the array element and the loop then recycles. Program execution then terminates. If you wish to output the information to the printer, simply change the PRINT statement in line 440 to LPRINT and the job is done. Especially for long lists, this program is far more useful when the output is directed to the printer.

Summary

The 13 programs included in this chapter were written on and for the PC-6300. For the most part, they are simple programs, although the last two may not fall into this category. Each program can be put to some practical use (with the possible exception of the backward letter program), and each provides a wealth of tutorial information to the beginning programmer. You are encouraged to modify and combine these programs to build software that will do a specific job for you. If you don't understand how these programs work, please reread the information accompanying them. These simple programs encompass a road spectrum of text-mode operations on the PC-6300, and routines similar to these will be used in nearly every program you write.

Chapter 5



Graphics Programs for the AT&T PC

This chapter will discuss some fairly simple graphics programs. Each is presented in a similar manner to the programs discussed in the previous chapter. We will start with fairly low levels of complexity before building up to higher-level programs.

To make this chapter as useful and educational as possible, it is divided into several sections to address different areas of graphics programming. Each of these areas will be explained as fully as possible using actual programs as examples. Each program and discussion assumes use of a color monitor. However, if you are using the AT&T monochrome monitor, the graphics displays will still be attractive, although they will appear in monochrome rather than in the colors described. Again, it is not necessary for you to purchase a color monitor in order to program graphics on the AT&T PC-6300. The optional RGB color monitor is desirable, however, for sophisticated graphics applications and mandatory if you want to actually see the various colors rather than seeing them displayed in shades of gray.

GENERAL GRAPHICS PROGRAMS

This section will discuss some fairly simple graphics programs of a general nature. Most of the programs will concentrate on using the LINE, CIRCLE, PAINT, and COLOR statements. These are good programs for the beginning graphics programmer.

Program 14: Random Boxes

This simple program does just what its name implies. It draws boxes at random on the screen. The screen is not cleared after each box is drawn, so the images accumulate, as shown in Fig. 5-1. If the program is run for a long enough period of time, these boxes will eventually fill the entire screen.

This program initially clears the screen and turns off the key at the bottom of the screen. The screen is then set to mode 1, the medium resolution graphics mode, by line 40. You will notice that line 40 is a multiple statement program line that contains both a SCREEN and a COLOR

```
10 REM RANDOM BOXES
20 CLS
30 KEY OFF
40 SCREEN 1,0:COLOR 0,0
50 LINE -(RND*319,RND*199),
RND*4,B
60 GOTO 50
```

Program 14. Random Boxes.

statement—separated by a colon, which is a must in GWBASIC. The COLOR statement sets the BACKGROUND color to 0 (black) and the PALETTE to 0, allowing the colors green, red, and brown to be used.

The boxes are drawn on the screen using the LINE statement in program line 50. Here, the ending coordinates for the line are set in the relative mode, as explained in Chapter 3. The line will be drawn from the *current* position of the graphic cursor to the coordinates specified in parentheses. This means that, after the first box is drawn, each succeeding box will be drawn from a point that marked the end of the one preceding it. (Remember, the LINE statement, when used with the B designator, creates unfilled boxes on the screen.) As soon as the first box has been drawn in line 50, line 60 branches back to it and this process continues ad infinitum.

The RND function is used as a multiplier to establish the X and Y coordinate values, and also

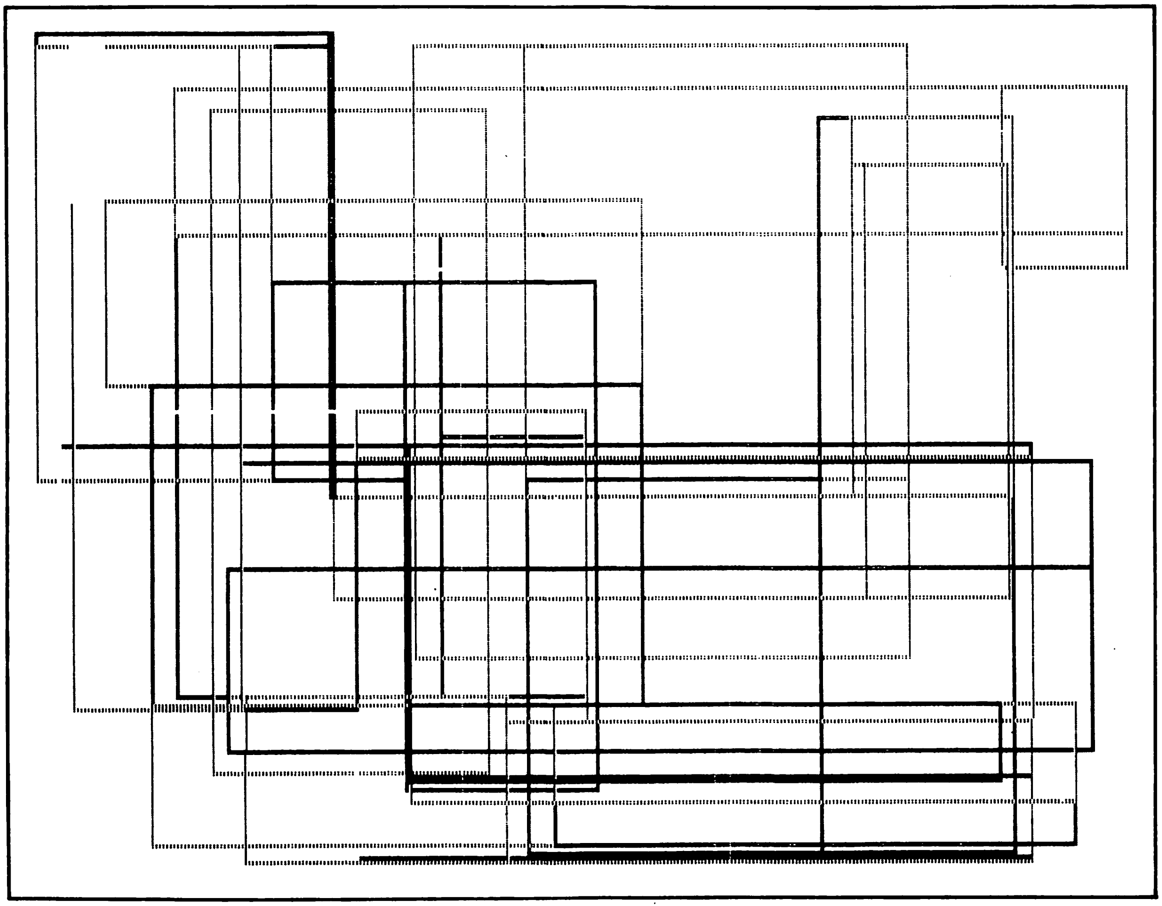


Fig. 5-1. Black-and-white representation of the screen display created by the Random Boxes program.

the color in which the box will be drawn. The X-coordinate will always be equal to a value of from 0 to 318. The Y coordinate may be any number from 0 to 198. (Remember, RND will always be less than 1.) It is not necessary to use the INT function here to specify coordinates; the computer will simply take the nearest integer value for a coordinate and plot from that point.

Any of the three colors in which the box may be drawn using PALETTE 0 are established by the RND*4 designator. This value will always be equal to a number from 0 to 3. If the number is 0, the box is drawn in the screen background and is not seen; if it is 1, the color is green, while values 2 and 3 produce red and brown, respectively.

This program is set up on a continuous loop and will continue to run until a manual halt is brought about at the keyboard. If you want this program to produce filled boxes at random, simply change the B designator in line 50 to BF. If you want it to draw random lines, simply drop the B designator and the comma preceding it. This is an interesting program to watch, but it can become boring after a while. If you want to mix up the pattern a bit, insert the following line:

```
15  RANDOMIZE
```

Now, each time the program is run, the RANDOMIZE statement will cause a prompt to be printed on the screen, telling you to input a random seed number. Type in any number in the specified range and then press Enter. Line 20 will still clear the screen and a whole new pattern of boxes will appear. If you type in a different number each time the program is run, a different box sequence will be generated.

Program 15: Expanding Globe

This program begins with a tiny circle at the center of the medium resolution screen. The circle then expands outward until it fills most of the screen. The circle is painted during each expansion, so there is an unusual pattern formed in the completed image. Figure 5-2 shows the completed circle.

```
10 REM EXPANDING GLOBE
20 CLS
30 SCREEN 1
40 KEY OFF
50 COLOR 0,0
60 FOR X=1 TO 100
70 CIRCLE(160,100),X,3
80 NEXT
```

Program 15. Expanding Globe.

Here's how the program works. Line 20 clears the graphic screen, while line 30 actually puts us in medium-resolution mode. Line 40 again erases the key from the screen, and line 50 establishes a black BACKGROUND and a PALETTE of 0. Line 60 is the start of a FOR-NEXT loop that increments X from 1 to 100. Each time the loop cycles, a circle is drawn about a center at coordinates 160,100. The radius of the circle, however, is determined by the value of X. Therefore, the first circle will have a radius of 1 pixel and the second a radius of 2 pixels. This process continues until the loop times out and the circle has a final radius of 100 pixels. Notice in line 70 that the color designator of 3 is used to draw the circle. This will produce a brown or tan color on the monitor screen, although on some it will appear somewhat yellow. As the loop cycles, the circle grows larger and larger. When the value of X reaches 100, the loop cycles for the last time and the program terminates. If you want to set this program up on a continuous loop add the following line:

```
90  GOTO 20
```

This one-line addition causes the program to run over and over again.

Program 16: Traveling Box

This program uses the DRAW statement (which will be discussed in more detail later) to draw a box-like pattern on the medium-resolution screen. The pattern, drawn at random, is shown in Fig. 5-3.

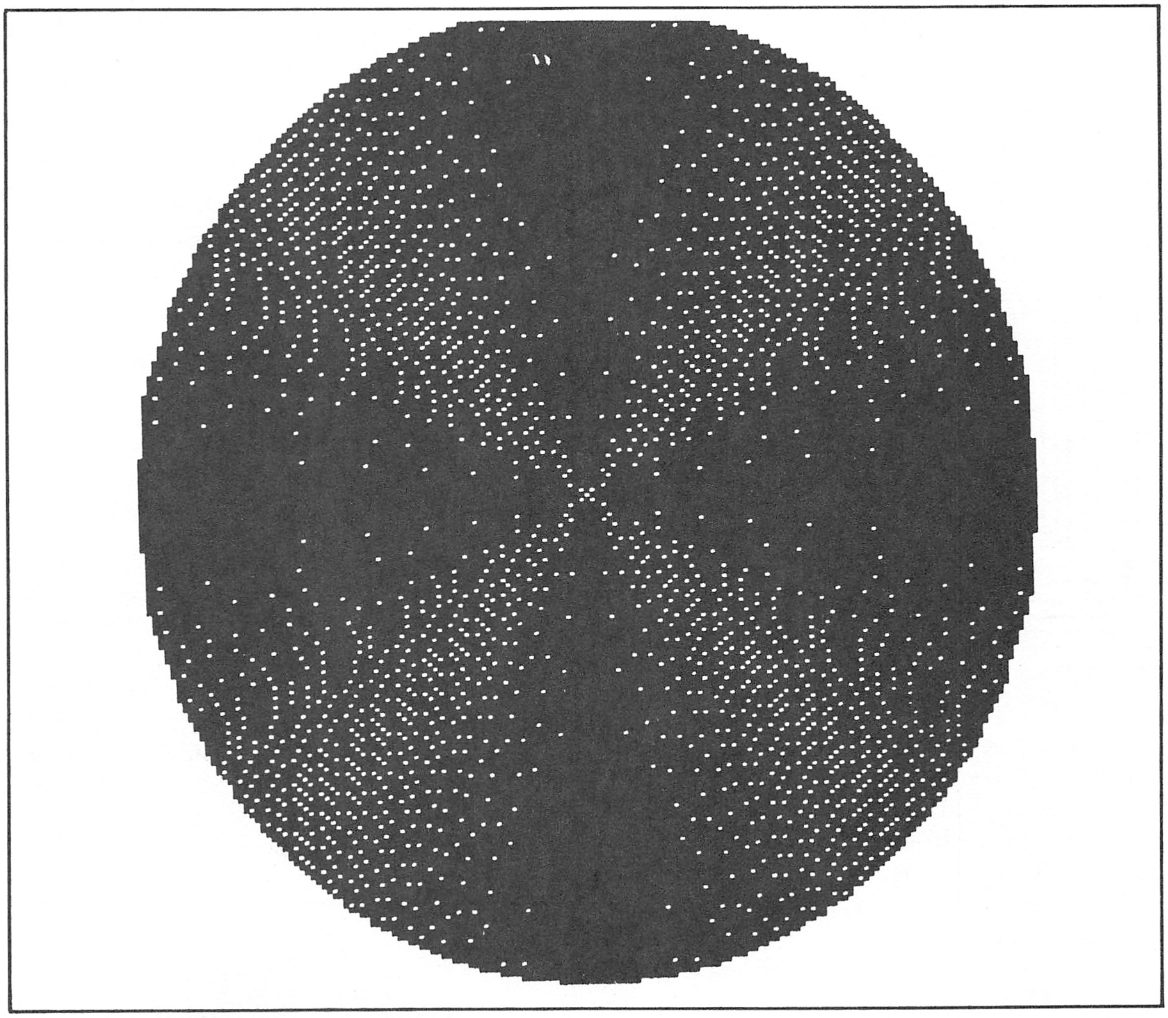


Fig. 5-2. Expanding Globe.

```
10 REM TRAVELING BOX DESIGN
20 REM COPYRIGHT FREDERICK HOLTZ
30 CLS
40 KEY OFF
50 SCREEN 1
60 COLOR 8,0
70 FOR A=-100*(RND) TO 100*(RND) STEP RN D*4+1
80 DRAW "U=A;R=A;D=A;L=A;"
90 NEXT
100 GOTO 30
```

Program 16. Traveling Box.

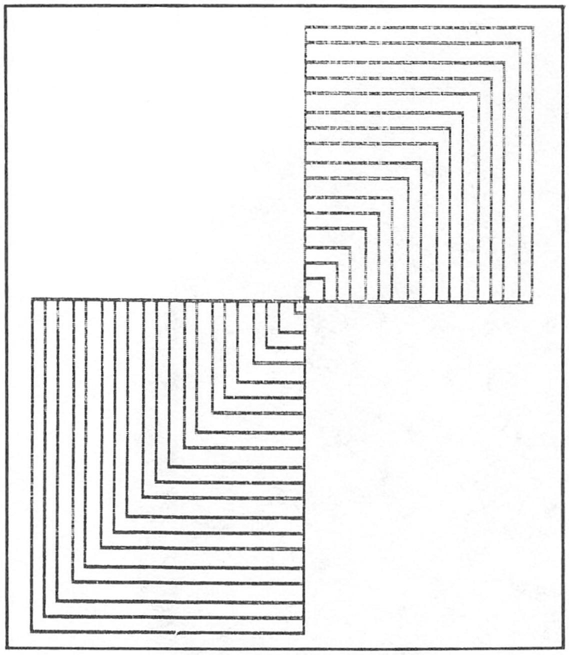


Fig. 5-3. Traveling Box design.

After the screen is initialized, the color is set to a dark gray background and 0 palette in line 60. Line 70 starts a FOR-NEXT loop that counts from a negative number to a positive number, with both the start and end points randomly selected. The STEP value is also selected at random, and will be equal to a value from 1 to 4. Line 80 contains the DRAW statement, which uses the designators U, R, D, L for Up, Right, Down, and Left, respectively. In every case, the box drawn will be symmetrical. As the program runs, the box starts at the bottom of the screen, passes through the center, and then begins making its way to the top right of the screen. This program is set up on a continuous loop, so different patterns will continue to form until the program is halted manually.

Program 17: Pinwheel

The pinwheel program draws a colorful circle at the center of the screen. The black-and-white version, shown in Fig. 5-4, does not do the screen image justice, since it is displayed in a multitude of colors. The program is similar to the expanding

```
10 REM PINWHEEL
20 CLS
30 KEY OFF
40 SCREEN 1
50 CLS
60 COLOR RND*15,RND*4
70 FOR X=1TO 70 STEP RND*6+1
80 CIRCLE(160,100),X,1
90 NEXT
100 FOR X=100 TO 155
110 PAINT(161,X),RND*4,1
120 NEXT
130 GOTO 50
```

Program 17. Pinwheel.

globe program, but here the PAINT statement is used (line 110) to fill in the spaces between circles. You will notice that, in line 60, the COLOR statement designates random values. The screen background may be any number from 0 to 14, while the palette may be any number from 0 to 3. In SCREEN 1 mode there are actually only two palettes open to us, usually numbered 0 to 1. However, any even number value inserted for the pallette number simply sets the computer to PALETTE 0,

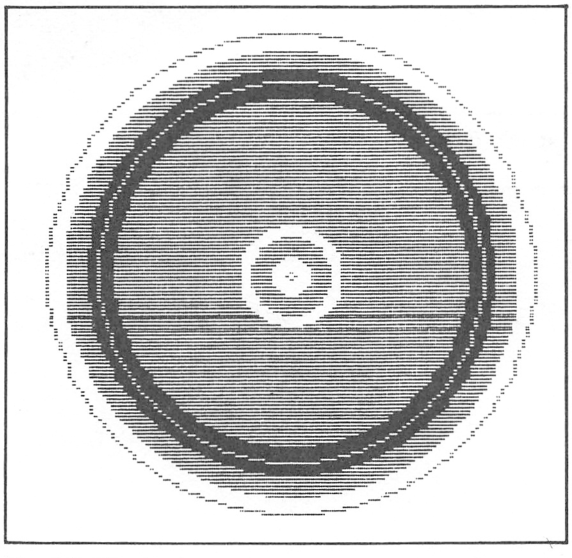


Fig. 5-4. Pinwheel.

while a 3 or any other odd number sets it to 1. In my opinion, using the higher numbers in a random program allows for a better mix of palettes.

Line 70 begins the first FOR-NEXT loop, which counts from 1 to 70 in random steps ranging from 1 to 6, depending on the output of the random number generator. A circle is then written at the center of the screen, with its radius determined by the value of X. Since X is stepped in random increments, the separations between circles will be variably spaced rather than uniformly spaced.

When the first FOR-NEXT loop times out, another steps X from 100 to 155. The value of X is used in the PAINT statement (line 110) to access a screen point that will lie in the bands between circles. If fills in these areas with a palette color from 0 to 3. When this loop cycles out, the entire pinwheel has been drawn and colored. The GOTO statement in line 130 branches back to line 150, where the screen is cleared. The program then begins to draw another pinwheel with newly established, randomly determined background and palette values. This is a very pretty program to view for long periods, because the screen background color is constantly changing and bringing about different palette effects. It's one of those displays you never grow tired of watching.

Program 18: Funnel

Again the CIRCLE statement is used in stepped radius sizes, but this time to produce a funnel effect shown in Fig. 6-5. After the screen is cleared and set up for medium-resolution graphics, line 60 sets the background to a dark gray; a palette value from 1 to 3 is established using the RND function. Here I've elected to include the INTeger function as well, although it's not necessary. Remember, the only two palettes available in SCREEN 1 mode are respectively. Since the numbers possible range from 1 to 3, there is a greater likelihood of achieving an odd palette number (1 to 3) than an even number (2). This was done for visual effect.

Line 70 begins a FOR-NEXT loop which steps X from 51 to 140, in random steps ranging from 1 to 10. The CIRCLE statement in line 80 uses the value of X to establish its coordinates. Notice that the Y coordinate is established by subtracting X from 240, while the circle radius is determined by subtracting 40 from the value of X. The palette color in which the circle is drawn is determined by a random number routine that outputs either 1, 2, or 3. As the loop recycles, various size circles are drawn on the screen, getting wider as they move toward the left. This produces an image that looks much like a funnel. The GOTO statement in line 100 simply locks up the computer in an endless loop, preventing the BASIC end-of-execution prompt (OK) from appearing. To exit this program, you have to bring about a manual halt via the keyboard. Figure 5-5 is shown in black and white, but the funnel will appear on a color monitor in multiple colors.

```
10 REM FUNNEL
20 REM COPYRIGHT FREDERICK HOLTZ 12/20/83
30 CLS
40 SCREEN 1
50 KEY OFF
60 COLOR 8,INT(RND*3)+1
70 FOR X=51 TO 140 STEP INT(RND*10)+1
80 CIRCLE(X,240-X),X-40,INT(RND*3)+1
90 NEXT X
100 GOTO 100
```

Program 18. Funnel.

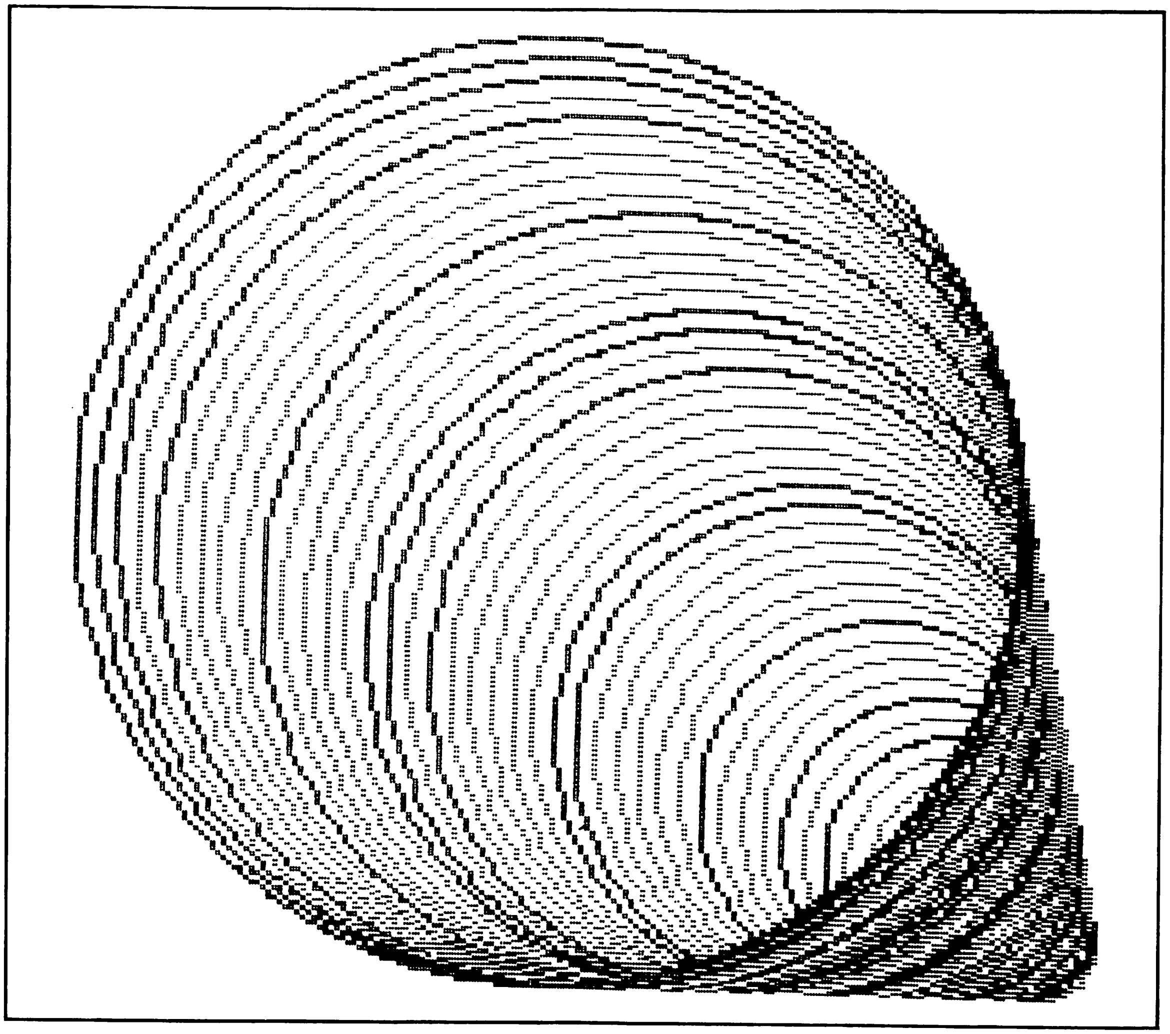


Fig. 5-5. Funnel.

Program 19: Cornucopia

Figure 5-6 shows the unusual display created by the Cornucopia program. This one is very similar to the funnel program, except the SIN function is used to plot the centers of the circles along a curve that starts at a point, travels upward to a maximum value, and then travels slowly downward past the point of origin to a maximum negative value.

After the screen is set up, line 60 begins a FOR-NEXT loop that counts from 1 to 240. Line 80 assigns to Y the value of 50 times the sine of the quantity X divided by 50. The circles created by this program are drawn either in white or black. If drawn in black, they are invisible; this is what produces the rough effect of a cornucopia shell. Line 70 alternates the color palette value between 0 and 3. Notice that A has not been assigned a value prior to the loop which began in line 60. Line 70 says that if A is equal to 0 then reassign A the value 3, but if A is not equal to 0, then reassign A the value 0. A is used with the CIRCLE statement in line 90 to determine the color. On the first cycle of the loop, since A has not been assigned it is

```
10 REM CORNUCOPIA
20 REM COPYRIGHT FREDERICK HOLTZ
30 CLS
40 KEY OFF
50 SCREEN 1
60 FOR X=1 TO 240
70 IF A=0 THEN A=3 ELSE A=0
80 Y=50*SIN(X/50)
90 CIRCLE(25+X,Y+100),1+X/5,A
100 NEXT X
110 GOTO 110
```

Program 19. Cornucopia.

automatically 0. Line 70 changes the value of A to 3, and this is then used as the color designator in line 90. On the next cycle of the loop, however, A is still equal to 3, line 70 changes the value back to 0. On each pass, A will alternate between 0 and 3.

The X and Y coordinates of the circle are determined by adding 25 to the value of X and 100 to the value of Y. The diameter of the circle increas-

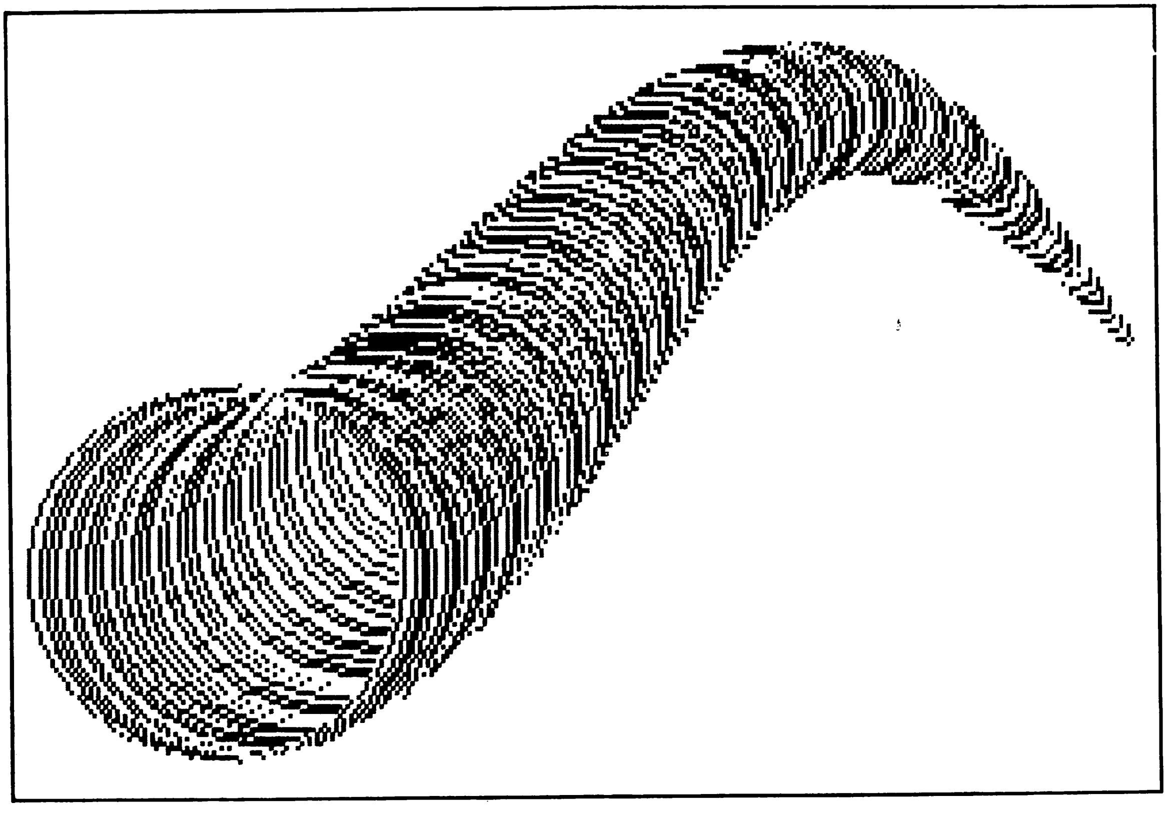


Fig. 5-6. Cornucopia.

es on each pass of the loop, its radius equaling 1 plus the value of X divided by 5. This program will take about 20 seconds to complete its run. When the loop times out, another endless loop lockup is formed in line 110, which keeps branching to itself. It will again be necessary to execute a manual halt at the keyboard to terminate execution. You may wish to alter this program by changing the values in line 70. Doing this can create a multi-colored cornucopia.

Program 20: Saturn

This program depicts a deep-space scene on the computer screen. The black-and-white version in Fig. 5-7 shows Saturn, along with four of its many moons. (What is the count now? Thirteen?) As

```
10 REM GRAPHIC DISPLAY OF SATURN AND ITS MOONS
20 REM COPYRIGHT FREDERICK HOLTZ
30 CLS
40 SCREEN 1
50 KEY OFF
60 CIRCLE(1,100),100,2
70 PAINT(1,100),1,2
80 LINE(0,112)-(160,110),3
90 LINE(0,97)-(160,110),3
100 LINE (0,100)-(160,110),2
110 LINE(0,104)-(160,110),2
120 LINE(0,103)-(160,110),3
130 LINE(0,102)-(160,110),2
140 LINE(0,101)-(160,110),3
150 LINE(0,99)-(160,110),2
160 LINE(0,105)-(160,110),2
170 LINE(0,106)-(160,110),3
180 LINE(0,107)-(180,110),2
190 LINE(0,108)-(160,110),3
200 LINE(0,110)-(160,110),2
210 CIRCLE (280,70),10,2
220 PAINT(280,70),3,2
230 CIRCLE (260,100),5,2
240 PAINT(260,100),2,2
250 LINE (0,40)-(70,40)
260 LINE (0,30)-(55,30)
270 LINE (0,70)-(95,70)
280 LINE(0,120)-(99,120)
290 LINE(0,150)-(80,150)
300 CIRCLE(210,150),3,2
310 PAINT(210,150),1,2
320 CIRCLE (190,170),3,2
330 PAINT (190,170),1,2
340 GOTO 340
```

Program 20. Saturn.

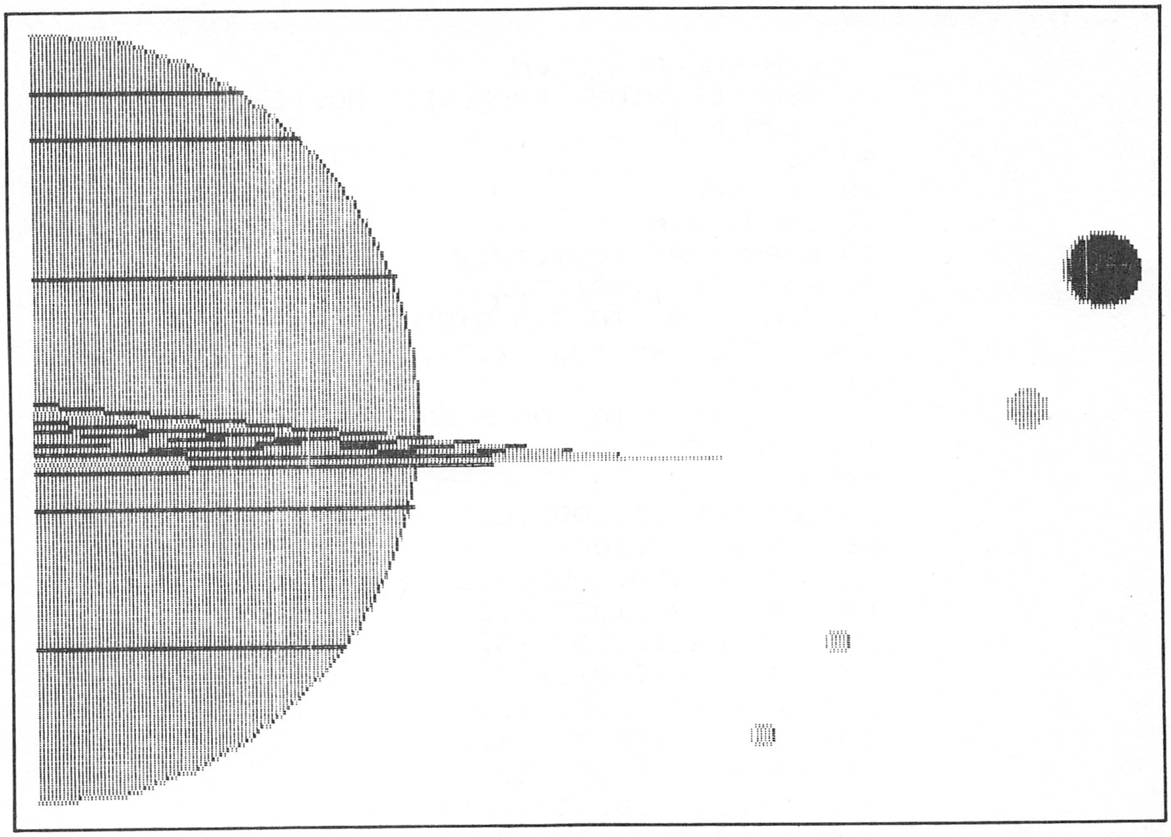


Fig. 5-7. Saturn.

you've probably already guessed, a number of CIRCLE statements are used to produce this display—five CIRCLE statements to be exact, one to draw the planet and four to draw the moons. Here's the way the program works.

The CIRCLE statement in line 60 produces a circle with a radius of 100. However, the center of this circle lies at screen coordinates 1,100. An X-axis value of 1 puts it at the far left-hand side of the screen, so only half of the circle actually appears. Line 70 paints this circle with palette color 1. Lines 80 through 100 draw the lines that make up the ring. Their coordinates were laboriously plotted by trial and error. The CIRCLE statement in line 210 draws the large moon on the far right, which is then filled in with palette color 3. The CIRCLE statement and its accompanying PAINT statement (lines 230-240) draw the moon just beneath the large one. The five LINE statements in lines 250 through 290 draw the lines on the surface of Saturn. The following CIRCLE and PAINT statements draw the remaining two moons. Program execution is again tied up with the GOTO statement in line 340.

While this program was written in four-color mode, some readers will probably want to create this display in sixteen-color mode, since the colors of Saturn number far more than four.

Program 21: Solar System

Staying in deep space, this program will produce a chart of the solar system. The display is shown in Fig. 5-8, and will probably look similar to many astronomical charts you have seen. This program consists almost entirely of CIRCLE and

```
10 REM SOLAR SYSTEM
20 REM COPYRIGHT FREDERICK HOLTZ
30 SCREEN 1
40 CLS
50 KEY OFF
60 COLOR 8,0
70 CIRCLE(45,100),10,2
80 PAINT(45,100),3,2
90 FOR X=150 TO 360 STEP 50
100 CIRCLE(45,100),X,3,,,5/18
110 NEXT X
120 FOR X=30 TO 100 STEP 20
130 CIRCLE(45,100),X,3,,,5/18
140 NEXT X
150 CIRCLE(15,100),3,1
160 PAINT(15,100),2,1
170 CIRCLE(90,105),4,2
180 PAINT(90,105),2,2
190 CIRCLE(100,87),5,3
200 PAINT(99,89),1,3
210 PAINT(100,87),1,3
220 CIRCLE(135,98),4,1
230 PAINT(135,98),3,1
240 CIRCLE(150,130),7,2
250 PAINT(150,130),2,2
260 CIRCLE(249,100),6,3
270 PAINT(250,98),2,3
280 CIRCLE(245,58),5,1
290 PAINT(245,58),3,1
300 CIRCLE(200,29),5,2
310 PAINT(200,29),2,2
320 CIRCLE(160,6),4,3
330 PAINT(162,6),1,3
340 GOTO 340
```

Program 21. Solar System.

PAINT statements. Line 70 draws the sun at the center of the solar system, although in this display it appears toward the left of center. The PAINT statement in line 80 fills in the circle with palette color 3. Lines 90 through 110 form all of the orbit paths on the display screen. X is counted from 150 to 360 in steps of 50. The CIRCLE statement in line 100 uses the center of the solar system, where the sun is displayed, to form the center of most of the orbital path circles. The radius value is determined by the value of X. The 5/18 designation at the end of the CIRCLE statement is the *aspect ratio*, the ratio of vertical screen dimension to horizontal. If the (default) aspect ratio of a true circle is

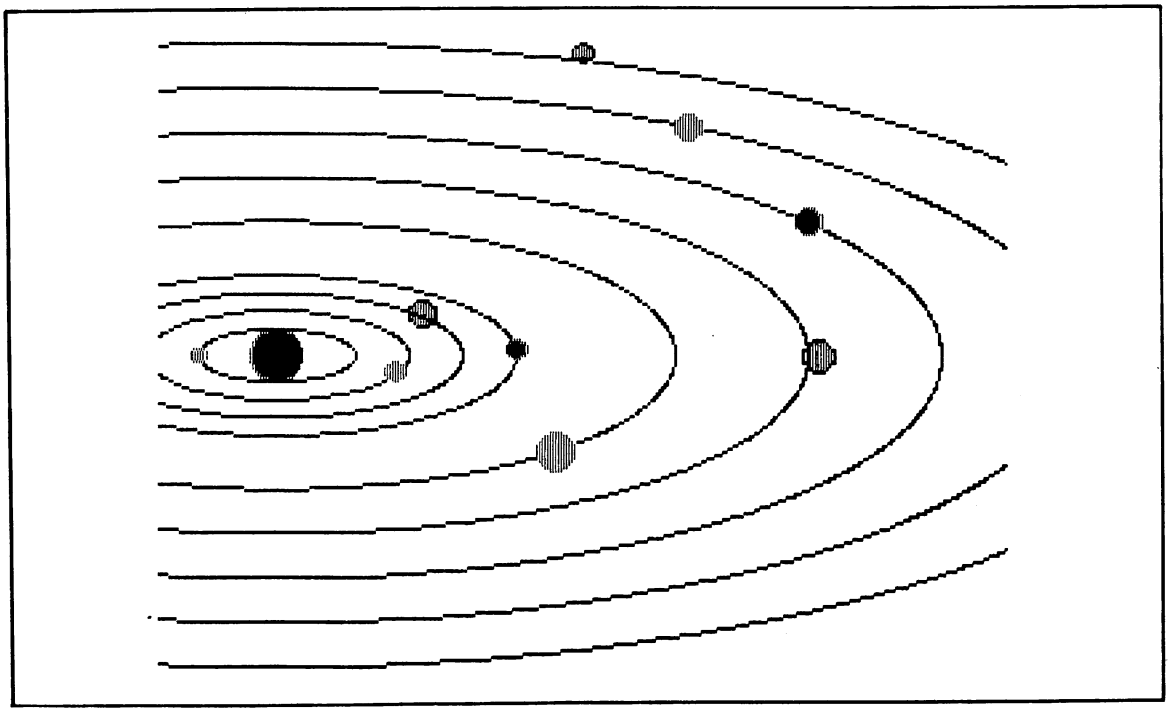


Fig. 5-8. The Solar System.

1/1, i.e., its vertical and horizontal radii are equal, specifying a different ratio (such as 5/18) will flatten the circle into an ellipse whose major and minor axes follow that ratio.

The FOR-NEXT loop begun in line 90 draws the outer orbit path, while another FOR-NEXT loop in lines 120 through 140 draws the inner orbital paths closer to the sun. Once the orbital paths have been created, it is necessary to draw the planets and fill them in, using CIRCLE and PAINT statements. The values contained in lines 150 though 330 were arrived at by trial and error. This is a pretty display to watch, and I have plans to use some animation routines to cause the planets to move along their orbits. Such a program, however, is saved for a later book.

Program 22: Text Animation Demo

Since animation was mentioned in the previous paragraph, it's appropriate to present this program, which provides a simple animation routine for moving text characters about the screen. This program sets up the screen for medium-resolution graphics mode, but it will work just as well with a SCREEN 0 statement in line 30. Line 50 clears the screen and line 60 prompts the user to input an animation speed between 1 and 100; the lower the speed number, the faster the animation. This input value is assigned to numeric variable SP, which is reassigned in line 70 to a value of 4 times SP. Line 80 clears the prompt from the screen. A loop is entered in line 90, which counts X from 1 to 40. The value of X is used in line 100 to determine the column position on the screen. The LOCATE statement places the cursor at row 14, column X. (We already know that X can range from 1 to 40, which happen to be the column values for the medium-resolution screen.) Line 110 prints the letter A at the screen position designated by the LOCATE statement.

Here's where the animation begins. Line 120 counts numeric variable DEL from 0 to SP. The FOR-NEXT statement in line 120 forms a time delay while the computer counts from 0 to the cur-

```
10 REM ANIMATION DEM CLS/LOCATE
20 REM COPYRIGHT FREDERICK HOLTZ
30 SCREEN 1
40 KEY OFF
50 CLS
60 INPUT"ANIMATION SPEED(1-100)";SP
70 SP=SP*4
80 CLS
90 FOR X=1 TO 40
100 LOCATE 14,X
110 PRINT "A"
120 FOR DEL=0 TO SP:NEXT DEL
130 CLS
140 NEXT X
```

Program 22. Text Animation Demo.

rent value of SP; the higher the value of SP, the longer the computer takes to count up to it. As soon as the loop in line 120 times out, the CLS statement in line 130 clears the screen and the outer (X) loop recycles. The LOCATE statement advances the cursor to the right one column, and again, the letter A is printed, one character position to the right of where the previous print occurred. The delay loop is again entered; when it times out the screen is cleared, and the letter A is printed in the next column position. This process continues until A has traveled the entire width of the screen. Each time the letter A is printed, it is erased and reprinted in an advanced position. That is what it's all about. The speed of animation is determined by the amount of time between when a character is written and when it is erased.

Program 23: Graphics Animation Demo

This program accomplishes the exact same screen display as the previous program, but this time the graphic PUT and GET statements are used to effect the animation. The program is the same down to line 90, where an array is established to hold the character to be animated. Line 100 places the cursor in the upper left-hand corner of the screen, and line 100 again prints an A.

The GET statement in line 120 places an array A the screen contents of a box whose upper left-hand corner is at coordinates 0,0 and whose lower right-hand corner lies at coordinates 7,7. This will encompass the entire letter A printed at location 1,1. Line 130 erases the A from the screen by putting the image to itself. Note that the coordinates used with the first PUT statement are the same as those used with the GET statement that identified the upper left-hand corner.

Line 140 starts the animation loop, which steps X from 0 to 309. Note that we are using graphic (not text) coordinates for the X-axis here. Line 150 uses the PUT statement to place the image of the letter A at location 0,100 on the first pass of the loop. Line 160 is the time delay loop, which works just as it did in the previous program. When the loop times out, the PUT statement in line 170 puts the image to itself again, erasing it. The loop then recycles, and again, the letter A travels from left to right across the screen. This program will cause the letter A to move more slowly, since we're dealing here with 309 different coordinates instead of the 40 in the previous program. You can speed the animation up by changing line 140 to:

```
140 FOR X=0 TO 309 STEP 10
```

This increments the X coordinate positions in steps in 10, as opposed to steps of 1.

```
10 REM ANIMATION DEM USING PUT/GET
20 REM COPYRIGHT FREDERICK HOLTZ
30 SCREEN 1
40 KEY OFF
50 CLS
60 INPUT"ANIMATION SPEED(1-100)";SP
70 SP=SP*4
80 CLS
90 DIM A(100)
100 LOCATE 1,1
110 PRINT "A"
120 GET(0,0)-(7,7),A
130 PUT(0,0),A
140 FOR X=0 TO 309
150 PUT(X,100),A
160 FOR Y=1 TO SP:NEXT Y
170 PUT(X,100),A
180 NEXT X
```

Program 23. Graphics Animation Demo.

Program 24: Animated Cigarette

This program shows how animation can be used effectively to produce unusual effects on the screen. When run, the program will show a graphic cigarette that will be "smoked" by the computer. As the cigarette burns, smoke rolls up from the tip and ashes begin to form. Figure 5-9 shows how the display looks when the program is first run.

The smoke is formed by combining sets of parentheses at different screen locations. The parentheses are printed in lines 80 and 90; line 100 establishes an array named A to hold them. The GET statement in line 110 commits the parentheses image to the array, and line 120 erases it from the screen by putting it to itself. The B designator in line 130 draws an elongated rectangle on the screen to represent the cigarette. A vertical line is drawn by line 140 to mark the segment where the filter is located, and lines 150 and 160 color in the various portions. Line 170 starts the animation loop, which counts X from 120 to 0 in negative steps of 6. (Remember, the cigarette body itself is not animated, only the parentheses that make up the smoke.) Line 180 puts the first set at position 80,X (80,120 on the first pass of the loop), a point just above the end of the cigarette. The time delay loop in line 190 slows the computer down a bit, and a simple count routine is set up using the variable R in line 200. The value of R is tested in line 210. If it's not equal to 40, there is a branch that causes the loop to recycle and produce more puffs of smoke.

Notice that the image is not put to itself after the loop times out; we want to produce an entire column of smoke when the entire loop cycles. Line 260 will cause it to cycle again starting in line 170. In other words, when the loop counts from 120 to 0, the column of smoke is formed. After the branch in line 260, the cycling of the loop will cause the column to be erased. Whenever the value of R is equal to 40, R is reset to 0 and the count sequence in line 220 is established. Line 230 then draws a

```
10 REM SMOKING CIGARETTE
20 REM COPYRIGHT FREDERICK HOLTZ
30 CLS
40 KEY OFF
50 SCREEN 1
60 COLOR 0,1
70 LOCATE 1,1
80 PRINT"()"
90 PRINT" ()"
100 DIM A(200)
110 GET(0,0)-(20,18),A
120 PUT(0,0),A
130 LINE(80,140)-(280,160),3,B
140 LINE(230,140)-(230,160),3
150 PAINT(85,144),3,3
160 PAINT(232,150),2,3
170 FOR X=120 TO 0 STEP -6
180 PUT(80,X),A
190 FOR Y=1 TO 25:NEXT Y
200 R=R+1
210 IF R<>40 THEN 250 ELSE R=0
220 C=C+5
230 LINE(76+C,140)-(78+C,230),0,BF
240 IF C=50 THEN 270
250 NEXT X
260 GOTO 170
270 END
```

Program 24. Animated Cigarette.

line near the end of the cigarette to represent the ashes. When C counts up to 50, the cigarette has burnt down far enough for the program to end.

Section Summary

In this section, I have tried to slide gently into graphics programming with some simple routines that have, finally, been built up into some fairly complex programs. Some of them are not terribly easy to understand from explanation alone. At this point, it would be a good idea to go back to any programs you don't fully understand and dissect them a line at a time. Try inputting different values to see what happens. You should understand what is contained in this first section before moving on to the next.

GRAPHICS UTILITY PROGRAMS

This section contains only two programs, both of which are of modest length. Both rely heavily on the POINT function in GWBASIC, and also on the PSET statement. Neither of these programs can produce an image on the screen unless one already exists. Instead, they modify images produced by other programs, such as those discussed in Section I of this chapter. This is why I call them *graphics utility programs.*

Program 25: Mirror Image

This program reads all of the points of light that make up an on-screen image, and then reverse them in mirror-image style. For example, the previous chapter showed a screen print from a program that produced a deep-space scene on the screen. The mirror-image program would simply reverse the image of the deep-space scene so that the moons were on the left and the half-body of Saturn on the right. It can also be used to produce letters from the normal text font that can only be read when held up to a mirror.

Here's how the program works. Line 30 sets up an array (A) to hold 320 elements, the number of points per horizontal line of the medium-resolution graphics screen. Notice that no CLS statement is used here, because it's assumed that an image already exists on the screen. Line 40 assigns Y an initial value of 0, and line 50 then begins the horizontal scanning loop which sequentially places the graphic cursor at every point on the medium-resolution screen. In the first cycle of the loop X is equal to 0, and we know that Y has been assigned to 0. The POINT function in line 60 assigns to numeric variable V the color numeral from 0 to 3 of the point on the screen at coordinates 0,0. (The POINT function, you will recall, returns the color numeral of the point on the screen at which the graphics cursor is placed.) The point value at the X-Y coordinate is assigned to V, and

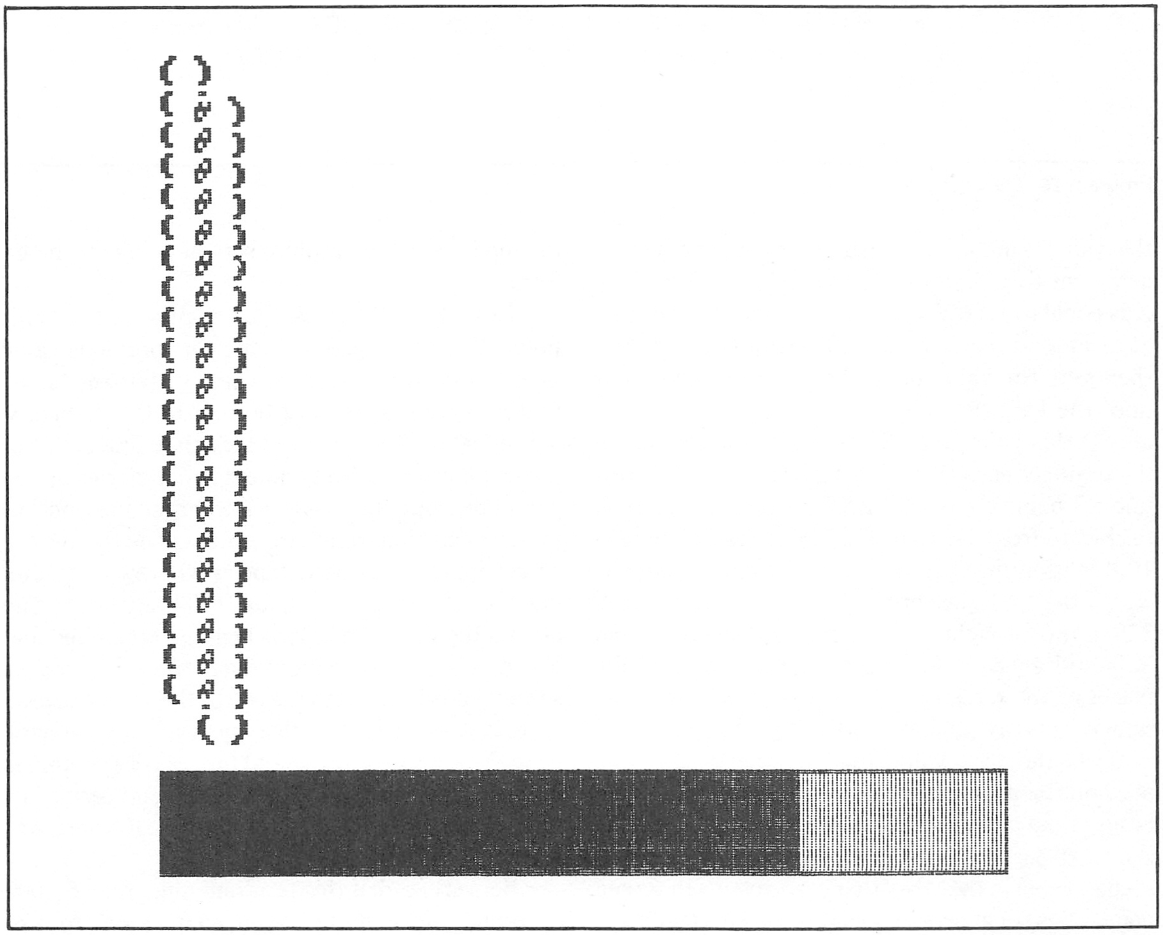


Fig. 5-9. Smoking Cigarette.

```
10 REM MIRROR IMAGE SCREEN GRAPHICS
20 REM COPYRIGHT FREDERICK HOLTZ
30 DIM A(320)
40 Y=0
50 FOR X=0 TO 319
60 V=POINT(X,Y)
70 A(X)=V
80 NEXT X
90 FOR X=319 TO 0 STEP -1
100 PSET(X,Y),A(319-X)
110 NEXT X
120 Y=Y+1
130 IF Y=200 THEN 150
140 GOTO 50
150 GOTO 150 'THIS LINE MAY BE USED TO S
AVE YOUR MIRRORED SCREEN DISPLAY THROUGH
 BSAVE
```

Program 25. Mirror Image.

this value is in turn assigned to the array in the first array position, A(0). On the second pass of the loop X is equal to 1, so V gets the point value of the pixel located at coordinates 1,0; array position A(1) then gets the value of V. This process continues until the loop times out.

At this point, the 320 points on the top line of the graphics screen are contained in array A. Now, line 90 begins a FOR-NEXT loop that counts X backward from 319 to 0. The PSET statement (line 100) is used to place the cursor at coordinates X, Y. On the first pass of the loop X will be equal to 319 at the far right-hand corner of the screen, and Y is still equal to 0. The color value that PSET places on the screen is determined by the value contained in array position A(319-X); 319 minus 319 is 0, so the first value read by the pass of the previous loop (at the left-hand corner of the screen) is set at the far right-hand corner of the screen. On the next pass of this negative counting loop, the second value in the array that corresponds to screen point 1,0 is set at screen point 318,0. This continues until the loop times out. The result is a reprint of the top line on the graphics screen in mirror-image form.

Now, line 120 steps Y by a value of 1. At this point, Y is now equal to 1, since its previous value was 0. Line 130 contains a test statement to see when we've reached the bottom of the screen; if Y is equal to 200, there is a branch to line 150. For now, 150 just forms an endless loop that ties up the computer, but you may wish to branch to another program portion from here, such as one to save the screen image to cassette tape or wherever. Assuming Y is not equal to 200, line 140 branches to line 50 and the program begins again. Remember that Y is now equal to one more than its previous value, so the second line from the top of the screen is read on this pass. It is then displayed in mirror-image form. Y is stepped again, and the process continues until the entire screen is read. The end result is a full screen image that is a mirror of what was previously there.

Be warned that this program must read 64,000 different points on the screen, so it's going to take about 3 or 4 minutes to mirror-image a full screen.

If, however, you wish to read just a portion of the screen, change the X and Y assignment lines to reflect the range of coordinates you wish to access. Figure 5-10 shows an example of the effect of this program.

Program 26: Graphics Screen Shrink

This program performs a function upon the screen image that already exists, just as the previous one did. Instead of creating a mirror image however, it creates an image that is smaller, the reduced image retaining the same symmetry as the original.

Line 40 sets up an array to contain 161 values. Lines 50 through 70 initialize numeric variables Y, R, and G to values of 0. Line 80 begins to read loop, which counts X from 0 to 319 in steps of 2. This means that X will be equal to 0, 2, 4, 6, etc. Every other *point* is thus sampled. As before, the POINT function is used in line 90 to assign the color numeral of the point to numeric variable V which line 100 then assigns to the first position in the array. Since G is equal to 0, A(0) will be equal to coordinate 0,0 on the screen. Line 110 uses PSET twice in a multi-statement line to blot out the screen points that have already been read. First, it blocks out the dot at coordinates X, Y by overwriting it with the background (0) color; the second PSET statement on this line then blocks out the point at coordinate X+1,Y. On the first pass of the loop, this would correspond with coordinate 1,0. Remember, the loop in line 80 increments X in steps of 2, so the point at coordinates X+1,0 was skipped over. Line 120 increments G by 1 and the loop then recycles. On the second pass of the loop, V is assigned a value equal to the color of the dot at coordinates 2,0, which value is committed to the array. On the next pass, the point at coordinates 4,0 is fed to the array, and this process continues until the entire line has been read.

We will later increment Y in a similar manner to that discussed in the previous program. However, Y will be incremented in steps of 2 rather than 1, so line 140 blots out every other *line* on the screen each time the loop cycles. This again gives us a clear slate on which to lay out our reduced image. Let's put the reduced image dots back on the screen. This is done with the loop beginning in line 150. Here X is counted from 0 to 159, and line 160 contains the PSET statement that begins writing the dots to the screen again. When this loop times out, the screen will display a line at the top that is equal to half of the original line. Line 180 in-

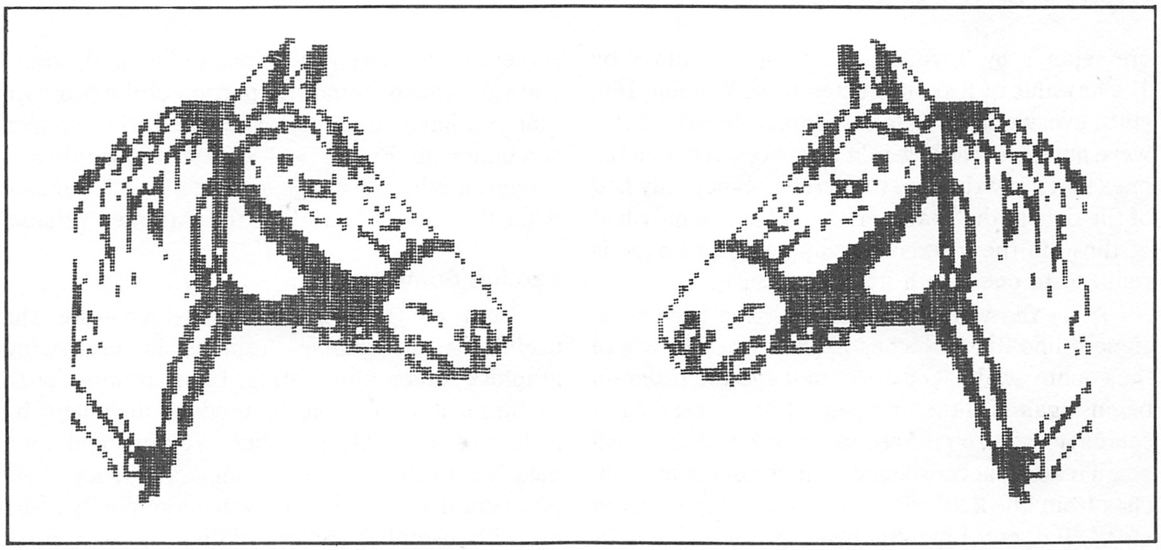


Fig. 5-10. Results of the Mirror Image Screen program.

```
10 REM PROGRAM TO SHRINK THE GRAPHICS SC
REEN
20 REM COPYRIGHT FREDERICK HOLTZ
30 SCREEN 1
40 DIM A(161)
50 Y=0
60 R=0
70 G=0
80 FOR X=0 TO 319 STEP 2
90 V=POINT(X,Y)
100 A(G)=V
110 PSET(X,Y),0:PSET(X+1,Y),0
120 G=G+1
130 NEXT X
140 LINE(0,Y+1)-(319,Y+1),0
150 FOR X=0 TO 159
160 PSET(X,Y-R),A(X).
170 NEXT X
180 Y=Y+2
190 R=R+1
200 IF Y>=199 THEN 220
210 GOTO 70
220 GOTO 220 'THIS LINE MAY BE USED TO S
AVE THE SCREEN IMAGE USING BSAVE
```

Program 26. Graphics Screen Shrink.

crements Y by 2 while line 190 increments R by 1. The value of R is subtracted from Y in line 160, since every other line in the Y-axis is read. (If this were not done, there would be spaces between the lines that are written to the screen.) Since only half of the dots in the X-axis are sampled and only half of those in the Y-axis are sampled, the image is reduced to one-fourth its original size.

After the values of Y and R have been reassigned, line 210 branches back to line 70, where the count variable G is reset to 0 and the program begins again. On the first pass, it began reading at coordinates 0,0; on the second pass, however, it will begin reading at coordinates 0,2. Subsequent branches from line 210 will cause the reading to begin at 0,4, 0,6, etc. Line 200 tests for the value of Y being equal to or more than 199, the bottom of the screen. If so, there is a branch to line 220, where you can branch to another portion of the program that you have added to make use of the reduced screen image. Figure 5-11 shows an example of a screen shrink. The image on the left is the original, while the one on the right is the shrunken version.

Section Summary

Both of these programs demonstrate the usefulness of graphics utilities in processing graphics screen information. Using routines such as these, it is possible to produce an image by means of a simple graphics program and then modify it to a high level of complexity. As you explore graphics further, you will undoubtedly come up with useful graphics utilities on your own. Remember, each of the programs outlined in this section take their input from the graphics screen, so there must be an image here already for any subsequent action to occur.

GRAPHICS TEXT PROGRAMS

While the name of this section may seem to contradict itself, text plays a great part in graphics programming. Text, in this context, only slightly resembles the text produced by the PC-6300 keyboard. Graphics text involves displaying standard text characters, often in an enlarged format. This process often involves screen utilities that read the dots forming a standard text character, and then make modifications to the display. You will be amazed at what can be accomplished by coupling some graphic utility programs that read and redisplay text with the standard character set.

Program 27: Enlarged Lettering Program

This program allows you to type in a short phrase via the keyboard, and then have it displayed at the center of the screen in an enlarged format. Figure 5-12 shows how the computer would display the phrase "WELCOME TO OUR HOME." The information the computer used to print this comes directly from the screen. Any phrase you type in (up to about 19 characters in length) can be displayed in enlarged format on the screen. Due to

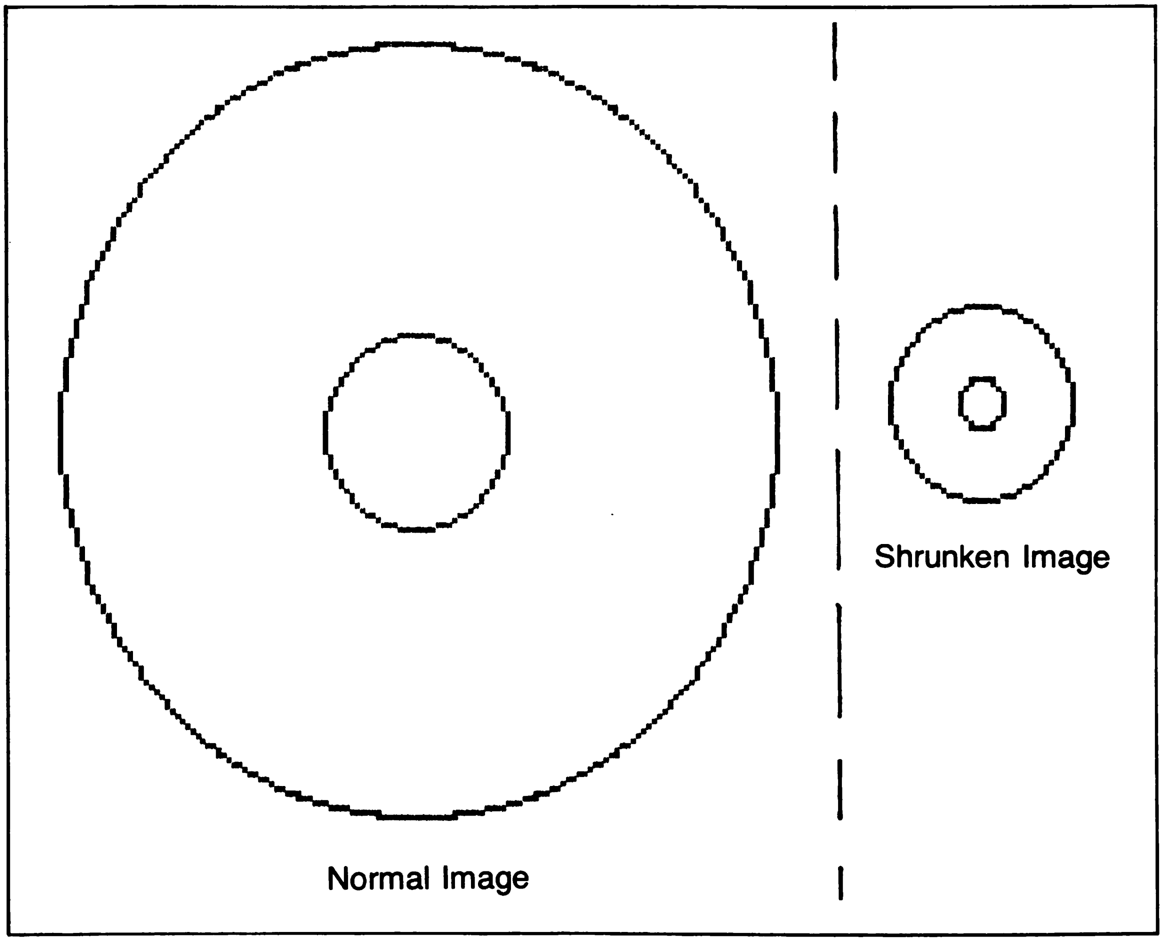


Fig. 5-11. Results of the Screen Shrink program.

```
10 REM LETTERING PROGRAM
20 REM COPYRIGHT FREDERICK HOLTZ
30 KEY OFF
40 CLS
50 SCREEN 1
60 COLOR 1,1
70 INPUT X$:CLS:PRINT X$
80 FOR X=0 TO 150
90 FOR Y=0 TO 7
100 V=POINT(X,Y)
110 GOSUB 150
120 NEXT
130 NEXT
140 GOTO 140
150 A=2*X:B=2*Y
160 PSET(A,B+100),V
170 PSET(X,Y),0
180 RETURN
```

Program 27. Enlarged Lettering Program.

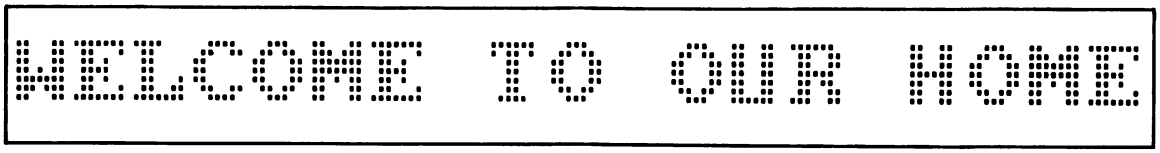
the powerful graphics statements in GWBASIC, programs to accomplish this seemingly monumental task are not monumentally long.

Line 40 clears the screen and line 50 puts us in medium-resolution, four-color graphics. The COLOR statement establishes a deep blue background and sets the palette to 1. Line 70 contains an INPUT statement with no prompt. When you see the question mark on the screen, this is a sign to type in the phrase you want enlarged. As soon as the phrase is input, the screen is cleared and your phrase is reprinted at the top of the screen.

A FOR-NEXT loop in line 80 begins to read the points at the X coordinates on the screen. It will scan from X coordinate 0 to X coordinate 150. Another nested loop in line 90 reads Y coordinates from 0 to 7. This will encompass a full character line 19 characters in length. Line 100 uses the POINT function to read the value of the dot at screen coordinates X,Y. In this mode the dot will be equal either to 0 or 3; 0 represents the blue background, while any white dots (3) are part of the screen write.

Line 110 uses GOSUB to branch to line 150. Here, variables A and B are assigned values of 2 times X and 2 times Y. Line 160 uses PSET to write points in color V at coordinates A, B + 100. Line 170 then blots out the point in the original character that has just been transferred to the center of the screen. Since a multiplier of 2 was used in line 150, the enlarged letter will be twice the height and width of the original characters. Line 180 returns to line 120, where the Y loop is recycled.

It would have been just as easy to include lines 150 through 170 as part of the loop in place of the GOSUB statement. This was done for demonstra-



```
WELCOME TO OUR HOME
```

Fig. 5-12. Enlarged lettering routine from the lettering program.

tion purposes, and also to keep the enlarging routine separate from the reading routine. Many programmers frown on premature or multiple exits from loops, saying that this can be confusing. In this case GOSUB is not really an exit, since it RETURNs to the loop structure; it merely isolates the routine and makes it accessible to other parts of the program. I will agree that loop exits and reentries should be used conservatively—only when there is no other way, and only when you're certain of *all* of the ramifications thereof.

As the Y loop recycles, other points along the Y-axis are read, and just as quickly the enlarged letters appear. When both loops have timed out, line 140 sets up and endless loop to prevent the OK prompt from appearing, but you might use line 140 to branch to another program portion or to save the screen to disk using BLOAD. The letters produced by this program are attractive, although some might object to the spaces that occur between the dots. We can improve on this program, and the next program in this section will demonstrate such a modification.

Program 28: Improved Lettering Program

This program is identical to the previous one, except for a slightly more complex enlarging routine. The enlarged letters produced by this program do not contain spaces like those of the previous program. Here each letter is filled in completely, as shown in Fig. 5-13. To accomplish this, the enlarging routine that begins in line 150 uses four PSET statements instead of one. The values of X and Y are again multiplied by 2, and the PSET statement in line 160 is identical to the one in line 160 of the previous program. However, the three additional PSET statements (lines 170-190) fill in the holes below and to the right of the original A,B coordinates. Again, line 200 blots out the original

```
10 REM INPROVED LETTERING PROGRAM
20 REM COPYRIGHT FREDERICK HOLTZ
30 KEY OFF
40 CLS
50 SCREEN 1
60 COLOR 1,1
70 INPUT X$:CLS:PRINT X$
80 FOR X=0 TO 150
90 FOR Y=0 TO 7
100 V=POINT(X,Y)
110 GOSUB 150
120 NEXT
130 NEXT
140 GOTO 140
150 A=2*X:B=2*Y
160 PSET(A,B+100),V
170 PSET(A,B+101),V
180 PSET(A+1,B+100),V
190 PSET(A+1,B+101),V
200 PSET(X,Y),0
210 RETURN
```

Program 28. Improved Lettering Program.

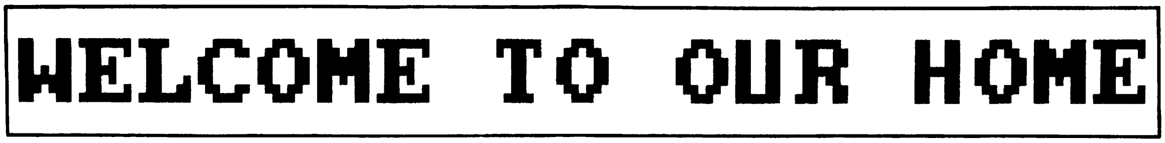


Fig. 5-13. The improved lettering routine results in enlarged letters that are filled.

dot in the character set which was read in line 100. The RETURN statement in line 210 gets us back onto the Y loop and the process is repeated until all of the input letters have been read.

Program 29: Improved Improved Lettering Program

The enlarged letters this program displays are the same as those shown in Fig. 5-13. Here, however, the four PSET statements found in the previous program have been replaced with two LINE statements in lines 160 and 170. Instead of setting the points separately, the LINE statements simply draw the lines in color V from the two sets of coordinates of two other sets that advance A and B appropriately. This program demonstrates the fact that there are several ways to do the same thing with the PBM PC-6300.

Program 30: Snowball Print Program

Here's another program that will modify the on-screen display from the standard character set. It enlarges these letters, but does so in an unusual manner that uses filled circles instead of dots. On the screen, it appears as if the letters are drawn with little snowballs. Figure 5-14 shows the black-and-white version, but the screen display is much more colorful.

This program was really the accidental by-product of a complex lettering program. In many ways it's very similar to the three previous programs, except that graphic PUT and GET

```
10 REM LINE STATEMENT LETTERING PROGRAM
20 REM COPYRIGHT FREDERICK HOLTZ
30 KEY OFF
40 CLS
50 SCREEN 1
60 COLOR 1,1
70 INPUT X$:CLS:PRINT X$
80 FOR X=0 TO 150
90 FOR Y=0 TO 7
100 V=POINT(X,Y)
110 GOSUB 150
120 NEXT
130 NEXT
140 GOTO 140
150 A=2*X:B=2*Y
160 LINE(A,B+100)-(A,B+101),V
170 LINE(A+1,B+100)-(A+1,B+101),V
180 PSET(X,Y),0
190 RETURN
```

Program 29. Improved Improved Lettering Program.

```
10 REM SNOWBALL PRINT PROGRAM
20 REM COPYRIGHT FREDERICK HOLTZ
30 KEY OFF
40 CLS
50 DIM F(30)
60 SCREEN 1
70 CIRCLE(8,8),6,2
80 PAINT(8,8),3,2
90 GET(0,0)-(17,17),F
100 PUT(0,0),F
110 CLS
120 COLOR 1,1
130 INPUT X$:CLS:PRINT X$
140 FOR X=0 TO 150
150 FOR Y=0 TO 7
160 V=POINT(X,Y)
170 GOSUB 210
180 NEXT
190 NEXT
200 GOTO 200
210 A=7*X:B=7*Y
220 IF V=0 THEN 240
230 PUT(A,B+50),F
240 PSET(X,Y),0
250 RETURN
```

Program 30. Snowball Print Program.

statements are used to place the little snowballs on the screen. Line 50 establishes array F; line 60 puts us in medium-resolution , four-color graphics; and the little snowball is made by the CIRCLE and PAINT statements in lines 70 and 80. Line 90 removes the initial snowball from the screen by putting it to itself. The screen is then cleared. Line 130 allows for the characters to be input, here limited

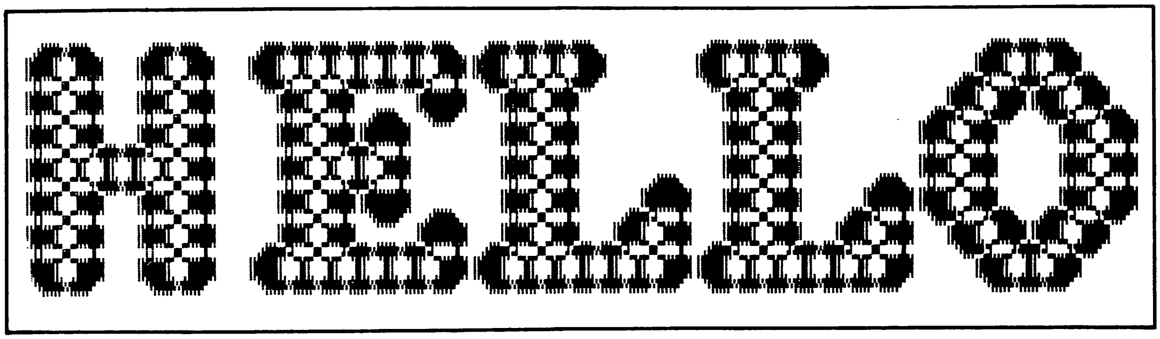


Fig. 5-14. The Snowball Print Program Screen Write.

to about five characters maximum. If you run over, the machine won't lock up; you'll just have to try again. Lines 140 and 150 set up loops to read the various points on the screen that contain the letters, and again the POINT function is used to assign the color number to V. There is a branch out of the loop to line 210, which starts the enlarging routine. Here, a multiplier of 7 is used to assign the values of A and B, so each letter is 7 times the width and height of the original. Line 230 uses the PUT statement to place the snowball at the correct point on the screen. If you need to print more than five letters on the screen, simply reduce the multiplier (7) in line 210. This will shrink the letters a bit.

Section Summary

The programs presented in this section have all acted upon the standard character set. Using these programs, almost any member of the character set displayable on the screen can be modified in many different ways. Though the PC-6300 is a very rapid processor, most of these routines are fairly slow to run in BASIC, because there are so many individual points of light (pixels) to read and write. Therefore, it's always best to write programs that only need to read a small portion of the screen. For improved speed, any of these programs can be used with a BASIC compiler software package. Compiled programs will typically run from three to six times faster than programs processed by the GWBASIC interpreter.

PROGRAMS THAT DRAW

This section deals with programs that use the DRAW statement in GWBASIC. The DRAW statement, discussed in Chapter 3 is used to draw lines up, down, left, right, and at various angles from one point to another. The DRAW statement requires a lot of arguments or numeric designators, and is therefore not especially easy to use—but it is often necessary for producing line drawings that cannot be made with other statements. The DRAW statement is very similar to the LINE statement, but it is not necessary to specify X-Y coordinates, only relative coordinates. For example, if the graphic cursor is located at coordinates 160,100, we can draw a vertical line from this point that is 60 pixels in height by using the LINE statement:

```
LINE - (160,40)
```

This method uses the LINE statement in relative mode to draw a line from coordinates 160,100 to coordinates 160,40. However, using the DRAW statement (and assuming that the graphic cursor is at coordinates 160,100) the program line would look like this:

```
DRAW "U60"
```

This will cause a line to be drawn from the current graphic cursor position (160,100) to a point that is 60 pixels above the original position. To draw a box on the screen, the program line would be:

```
DRAW "U40R40D40L40"
```

this example, similar to one discussed in an earlier chapter, tells the computer to draw a line up 40 pixels, right 40 pixels, down 40 pixels, and left 40 pixels.

In this particular example it might be just as easy to use the LINE statement with the B designator, but one aspect of the DRAW statement has it and all the others beat. The DRAW statement may be used with a *scale factor*—a designator that can magnify or shrink any lines produced with the DRAW statement. For example, if we change the previous DRAW statement example to:

```
DRAW "S8U40R40D40L40"
```

the box would be drawn at 80 pixels on a side instead of 40. The scale factor is derived by dividing by 4 the numeral used with S. In this case, 8 divided by 4 is 2, so the magnification is two times every other command value on the DRAW statement line. Using this scale factor, for example, the U40 command is automatically converted to U80. If we used a scale factor of S2, 2 divided by 4 is .5—so the square would be 20 pixels per side. If we DRAW an object and it turns out to be of an unsuitable size, all we need to do is insert a scale designator in the DRAW statement line, and then run the program again. This is the only statement in GWBASIC that allows for quickly enlarging or shrinking graphic images. Remember, however, that DRAW cannot be used to alter the size of a graphic image produced by statements such as CIRCLE and LINE.

Program 31: Interlace

The Interlace program produces a fairly intricate bit of lacework on the screen. DRAW statements are used to produce the lines required to form what appears to be an interlaced rope pattern. This pattern consists of four loops which cross over or under other image segments.

After the outline is produced by the DRAW statement, PAINT statements are used to fill in the various sections. A randomized COLOR statement changes the screen background and the foreground palette each time the design is drawn. A time delay loop in line 190 allows the filled image to establish itself for a short time before the screen is cleared and the next image is drawn. If you want to lengthen the time and image remains on the screen, simply increase the maximum value of this loop.

If you have a color monitor, you will notice that sometimes the lacework produced by this program will seem to be perfectly flat, while at other times it will appear to be three-dimensional. This effect is dependent upon the random color indexes that are output. In some cases, the foreground and background color may be identical. When this hap-

```
10 REM INTERLACE
20 SCREEN 1
30 A=RND*10+1
40 B=RND*3+1
50 COLOR A,B
60 X=RND*3+1
70 R=20
80 DRAW"S=R;C3 BM100,150U12BU4U12R12BR4R12D12BD4D12L12BL4L
   12"
90 DRAW"S=R;C3 BM100,150U12R8BR4R16D4BL4L12BL4L4D4R4BR12R4
   U4"
100 DRAW"S=R;C3 BM100,150R12U16BU4U8L8BD4D4BD12D4R4U12BU4U
    4L4"
110 DRAW"S=R;C3 BM100,150U12BU8BR4R12BR4R4U4L4D12BD4D4R4"
120 DRAW"S=R;C3 BM100,150BR16U8BU4U16"
130 DRAW"S=R;C3 BM100,150BU16R16BR4R8"
140 X=RND*3+1
150 PAINT(101,38),X,3
160 PAINT(185,48),X,3
170 PAINT(185,145),X,3
180 PAINT(101,145),X,3
190 FOR TS=1 TO 600:NEXT TS
200 CLS
210 GOTO 30
```

Program 31. Interlace program.

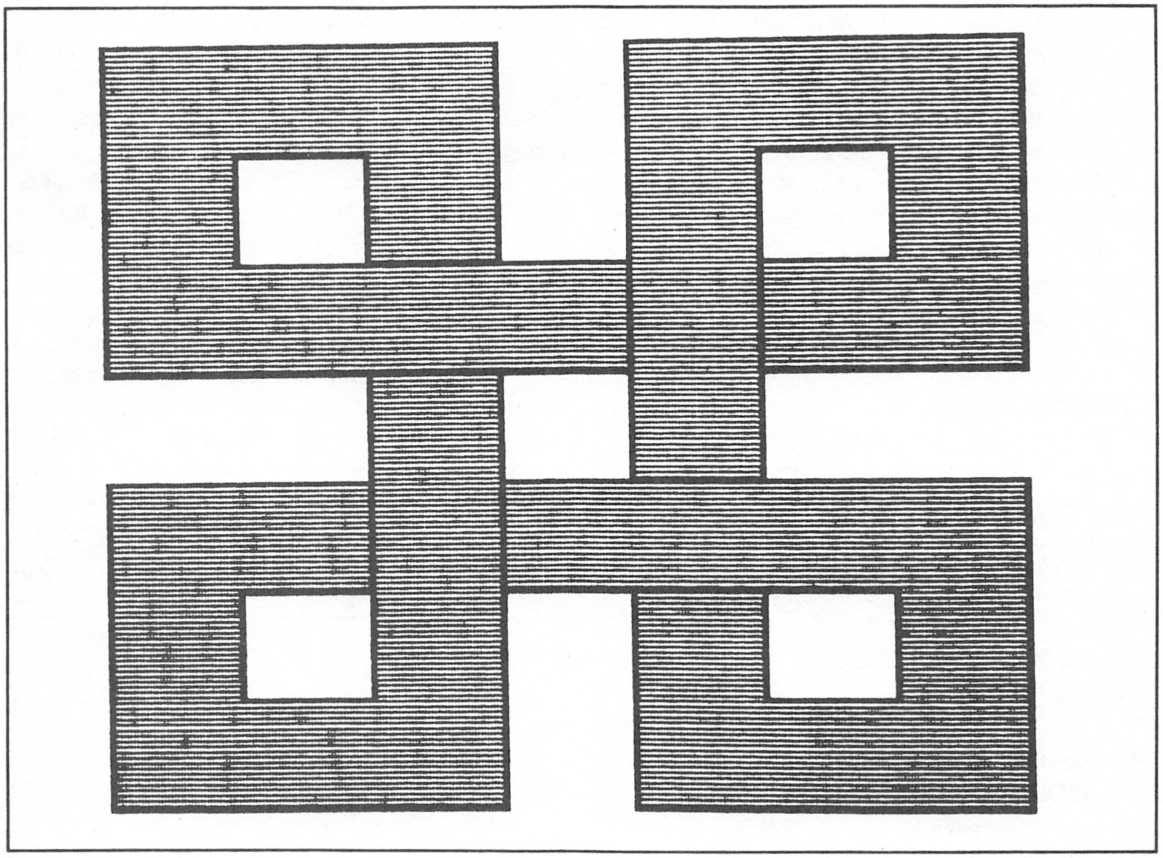


Fig. 5-15. Screen display from the Interlace program.

pens, the screen will appear blank as the foreground image is being drawn in the same color as the background. Figure 5-15 shows an example of the pattern generated by this program.

Program 32: Electronic Graph

An *oscilloscope* is an electronic test instrument that will display visually various electronic operations and parameters. These devices are often used to display the waveform of an ac signal. We can simulate the same effect on the AT&T PC through appropriate programming. The computer does not actually read or measure the output from an electronic device, but we can build in certain parameters by means of software.

This program will generate a sine wave on the monitor screen. A sine wave is constantly chang-

```
10 REM SINE WAVE
20 CLS
30 P=60
40 SCREEN 1
50 LINE(0,100)-(319,100)
60 PI=3.14159
70 A=-11*PI
80 B=7*PI
90 C=638/(B-A)
100 FOR D=A TO B STEP .1
110 X=D*C
120 Y=SIN(D)*P
130 PSET(X+60,100+Y)
140 NEXT D
150 END
```

Program 32. Electronic Graph.

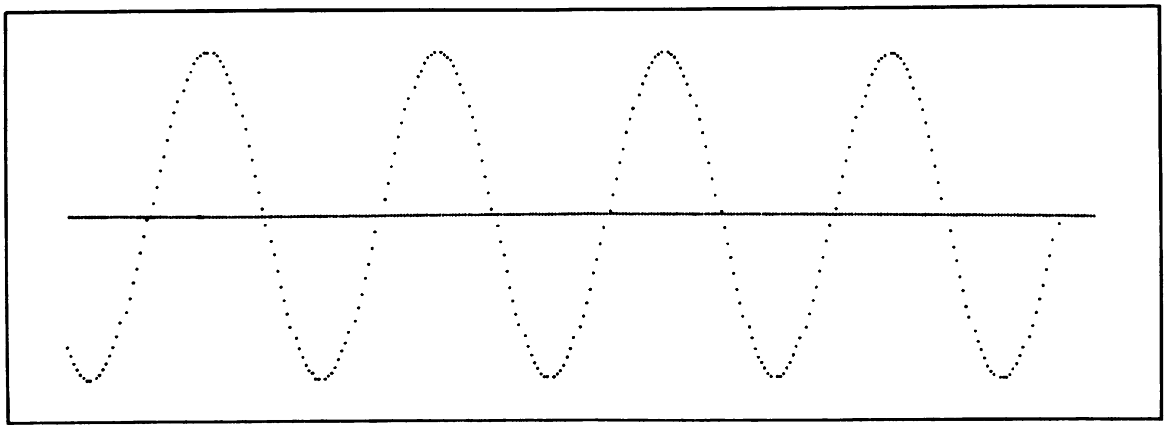


Fig. 5-16. Graphic display of a sine wave produced by the Electronic Graph program.

ing polarity, with equal portions of the curve on the positive and negative sides of the graph. Figure 5-16 shows the graphic display. When this program is run, the solid horizontal line represents the value 0, while the upper portion of the scale is positive and the lower portion is negative. The graph shows a perfect sine wave, in that those wave portions which lie in the upper portion of the scale are mirror images of those which lie below.

The reference line is drawn in program line 50. This line spans the entire horizontal width of the screen and is situated at its center. Line 60 establishes the value for π, while lines 70 and 80 determine the starting and ending points of the sine wave display. Line 90 divides the difference between B and A into 638, which is exactly twice the maximum screen width (319). Line 100 establishes the spacing between each point plotted on the graph. Here I'm using a step of 0.1. Line 120 uses the SIN function to cause the plotted points to be displayed as a true sine wave. The sine of the value of D is multiplied by P. The latter variable determines the maximum value of each wave section, or the distance of each peak from the horizontal line. Line 130 performs the actual plotting function.

Due to the small step in line 100, it will take 30 seconds or so for the entire graph to be completed. You can use larger step factors, although each waveform will not be displayed as finely. We could say that this is a sine wave that represents a value of 60 volts ac, since we chose a value of 60 for P. This assumes that the value of P is to represent potential difference directly. Any other value could be assigned to P, as long as it does not exceed 100. For the display actually shown, the value of P is unimportant, at least in regard to it representing some true electronic value. However, if such a graph were to be used for the comparison of sine waves of different voltage values, P's value would be all-important.

Program 33: Schematic Diagram

This simple program will draw the schematic diagram shown in Fig. 5-17, which electronics enthusiasts will recognize this as a resistor array. The commands contained in lines 70 and 80 each represent a line in this drawing. In line 60, numeric variable X is assigned a value of 8. In line 70, you will see the scale factor used in the format

```
S=X
```

which sets the scale factor to 8, representing a magnification of 2. Therefore, every number used in any DRAW statement line *following* the S designator will represent twice that number of pixels on the screen. At this point in the discussion, I will begin with the DRAW statement in line 70 and work through to the end of line 80, describing the on-screen action of each command.

First, the BM80,90 designation tells the com-

```
10 REM SCHEMATIC DIAGRAM
20 REM COPYRIGHT FREDERICK HOLTZ
30 SCREEN 1
40 CLS
50 KEY OFF
60 X=8
70 DRAW"BM80,90;S=X;R8 U10 R10 E1 F3 E3
F3 E3 F3 E1 R24 E1 F3 E3 F3 E3 F3 E1 R10
 D10 NR8 D10 L10 G1 H3 G3"
80 DRAW"H3 G3 H3 G1 L24 G1 H3 G3 H3 G3 H
3 G1 L10 U10"
90 END
```

Program 33. Schematic Diagram.

puter to move invisibly to screen position 80,90. The scale factor is next, and then the command R8. Refer now to the schematic drawing in Fig. 6-17 to follow the lines that the commands represent, beginning at the extreme left side of the diagram. R8 means move right 8 pixels (16 pixels because of the scale factor); this produces the horizontal line. Following the next series of commands in line 70, we move up 10 and then right 10. This draws the left top corner of the schematic. The next seven commands draw the jagged lines that make up the first symbol for a resistor. E1 means move diagonally up and right one pixel. F3 tells the computer to move diagonally down and right three pixels. These commands are repeated until the resistor symbol is complete.

With the first resistor completed, we hit the R24 command, which draws the top horizontal line between the two resistors. The same E-F sequence is repeated to draw the second resistor, and we now move right 10 and down 10 to draw the right top corner of the schematic. The NR8 command draws the right horizontal line emerging from the schematic box. R8 still means "draw eight pixels to the right," but the N prefix tells the computer to then return to the point where it started. We move down 10 and left 10 to draw the lower right-hand corner of the schematic. The last three commands in this line and the first four commands in line 80 draw the bottom right resistor symbol. These commands are reciprocal to those which drew the top set of resistors, using G to move

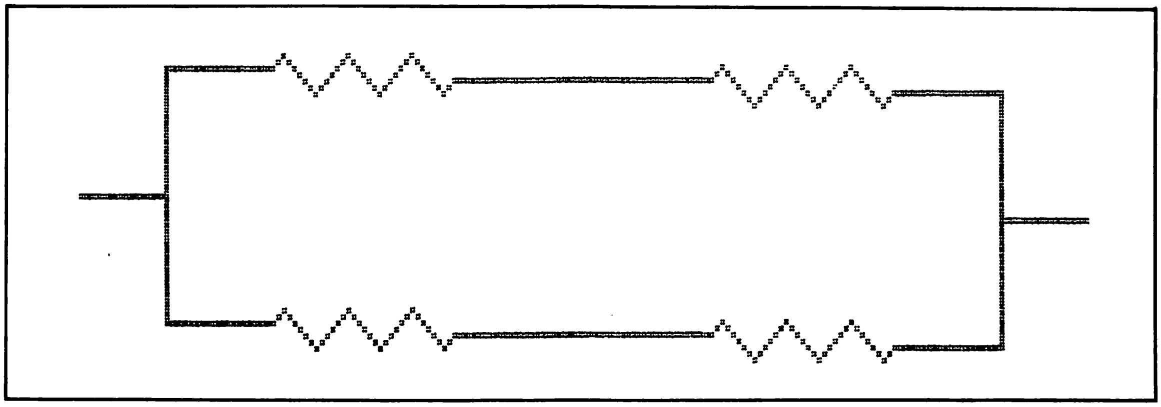


Fig. 5-17. Screen display from Schematic Diagram program.

```
10 REM DIODE ARRAY
20 REM COPYRIGHT FREDERICK HOLTZ
30 CLS
40 SCREEN 1
50 KEY OFF
60 X=20
70 FOR Y=0 TO 3
80 DRAW"A=Y;BM160,100;S=X;U3 BU1 U3 BD3
NR1 NL1 G1 R2 H1"
90 FOR DLAY=1 TO 400:NEXT DLAY
100 NEXT Y
```

Program 34. Diode Array.

diagonally down and left, and H to move diagonally up and left. When this is complete, a line is drawn left 24 pixels to another resistor drawing. The last two commands in 80 draw a line left 10 and up 10 to complete the lower left-hand corner of the schematic; the diagram is now complete.

It takes a bit of memorization to keep track of the meaning of each command, and, of course, you keep very close track of where you are. The DRAW statement is best used to reproduce a drawing on the screen from a textbook or other hard-copy reference source, for reasons you will see in subsequent programs.

Program 34: Diode Array

Here's another drawing, a diode array consisting of four diode symbols connected back-to-back, as shown in Fig. 5-18. In this program a scale factor of 20 is used, which means that each numeric designator will be multiplied by 5. Notice, however, that the commands contained in the only DRAW statement line (program line 80) would seem sufficient only to draw one of the four identical diode symbols. Here's where another DRAW statement command comes in handy. The A designator stands for *angle*, and is assigned any value from 0 to 3, with a default value of 0. If A is 1, any commands given automatically produce lines rotated 90 degrees to the left. In other words, a U20 command after an A = 1 command would really be converted to L20; instead of the line being drawn up 20 pixels, it would be drawn left 20 pixels. A designation for A of 2 rotates the line 180 degrees to the left, and 3 provides a 270 degree rotation.

This program uses the FOR-NEXT loop begun in line 70 to rotate the display through all four angles. Here's how it works. When line 80 is first executed, the A designator is set to 0, since Y is equal to 0. The cursor is set to the center of the screen, and the scale factor is set to X, which is 20.

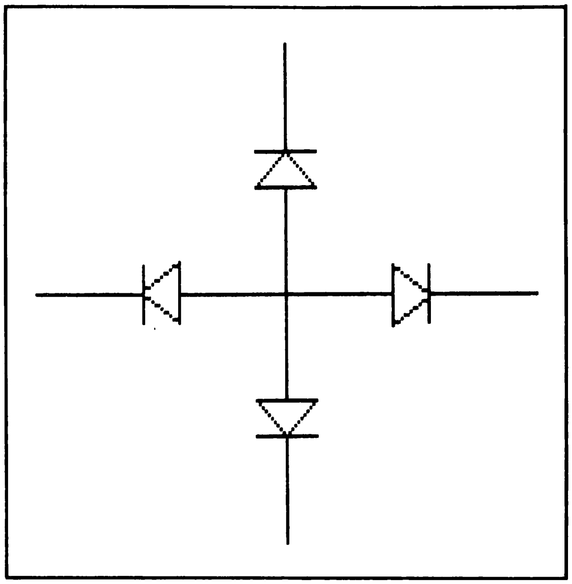


Fig. 5-18. A computer-generated diode array.

The remaining commands draw a single diode, which will appear vertically on the screen because the angle factor is 0 on the first cycle of the loop.

Line 90 is a simple time delay built into the loop for demonstration purposes. The outer loop is recycled in line 100, and Y will now be equal to 1. The angle is now equal to 1, or 90 degrees away from the original. The same DRAW statement line is now used to draw a diode horizontally and to the left of the screen center. When the loop cycles again the angle is 2 (180 degrees left) so a vertical diode symbol is drawn from the center toward the bottom of the screen. On the last cycle of the loop the angle is 3 (270 degrees left of the original), so a horizontal diode is drawn to the right of center. Our diode array is complete. If you now remove line 90, it will appear as through the entire diode array is drawn instantaneously.

Both this and the previous program assigned X a value to be used with the S designator. It is not necessary to assign this value to a variable. A command of S20 would be perfectly adequate, but the assignment method makes it easy to go back and simply change the value of X. For example, change line 60 to:

```
60 X=40
```

Your diode array will now be twice its previous size. If you change the value to 60, the array is even bigger, but portions of it run off the screen. Now change the value of X to 10. The array is quite small and somewhat out of proportion. You can even shrink it down to X=1, but all you will get is a single dot at the center of the screen.

Program 35: Bridge Rectifier Schematic

This program produces a schematic of a bridge rectifier circuit, as shown in Fig. 5-19. Here, I have used the DRAW statement in a program that also prints information to the screen. Lines 50 and 60 create the schematic itself, and lines 70 through 130 print the designations in various screen locations. See if you can figure out this DRAWing yourself, based on what you've learned in the previous program.

Program 36: Block Diagram

Figure 5-20 shows a line drawing known as a *block diagram.* In this case, each of the blocks

```
10 REM FULL BRIDGE SCHEMATIC
20 REM COPYRIGHT FREDERICK HOLTZ
30 CLS
40 SCREEN 1
50 DRAW"S25 BM 150,100 E7 NL15 F7 NR8 G7
 NL15 H7 L2 D5 G1 D3 F1 D3 R24"
60 DRAW"BM150,100 BE4 NH1 NF1 L1 F1U1 BE
3 BF4 NE1 NG1 U1 G1 R1 BF3 BG3 NH1 NF1 D
1 H1 R1 BG4 BH3 NE1 NG1 U1 G1 R1"
70 LOCATE 14,5
80 PRINT "AC INPUT"
90 LOCATE 13,40
100 PRINT"+"
110 LOCATE 22,40:PRINT"-"
120 LOCATE 18,30
130 PRINT "D.C. OUTPUT"
```

Program 35. Bridge Rectifier Schematic.

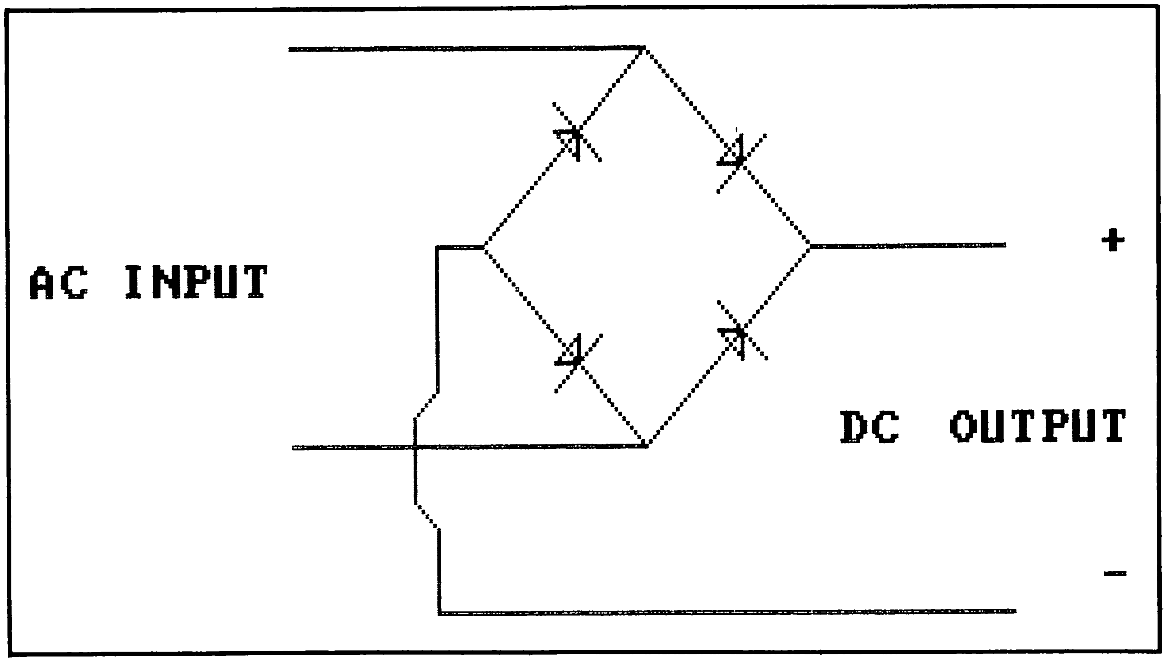


Fig. 5-19. Computer-generated full-wave bridge rectifier circuit.

represents a different portion of an electronic circuit. We have no alphabetic designators in each section because all we're interested in for this exercise is the line drawing itself. This is a simple one-dimensional drawing, but it will take a bit of time to write into a program using the DRAW statement. Admittedly, we could use LINE statements to draw the same picture more quickly. However, when doing more complex artwork, this isn't possible.

First, it is necessary to establish a relationship for each line in the drawing. This is done quite simply by using a finely graduated straightedge to measure the length of each line and line segment between intersections. Figure 5-21 shows my mockup of the drawing in Fig. 5-20; this is just a freehand sketch that serves as a chart for later programming, in much larger form to allow for the insertion of the relative values of each line. These values were obtained by measuring each line in the original drawing. You will want to choose a

```
10 CLS:SCREEN 1
20 DRAW"S10;BL30 R10 U4 R10 BL10 U13 L2
U18 BD18 L8 D17 BR10 BU4 BR10"
30 DRAW"U3 R10 D6 L10 U3 BR10 R10"
40 DRAW"U3 R8 D6 L8 U3 BD3 BR4 D8"
50 DRAW"R6 D6 L12 U6 R6 BU8 BR4 BU3"
60 DRAW"R8 U20 L12"
70 DRAW"U4 L21 D8 R21 U4"
80 END
```

Program 36. Block Diagram.

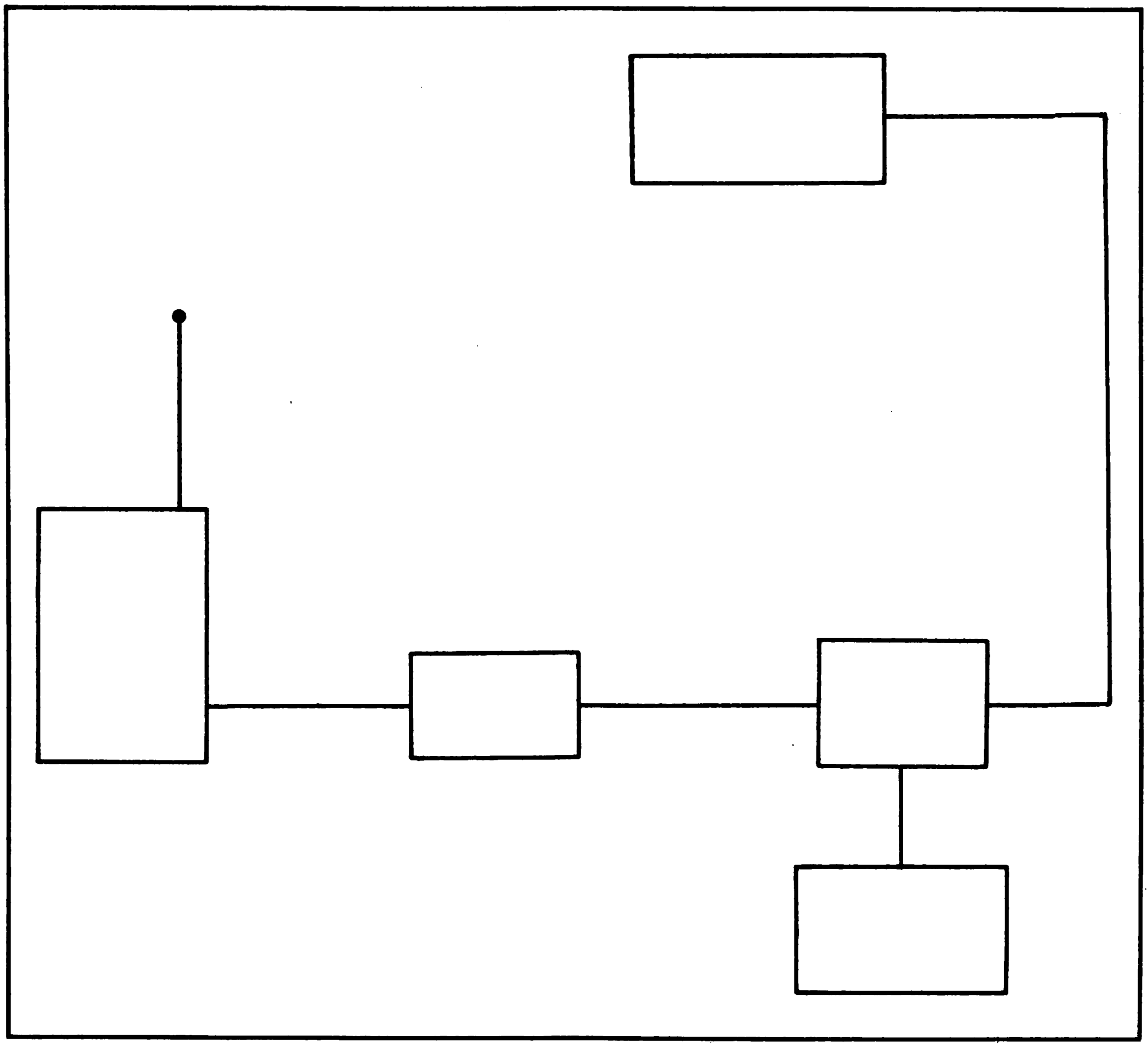


Fig. 5-20. A schematic block diagram such as the one shown here may be accurately reproduced on the computer screen.

straightedge that is graduated in very small values. In this particular case the values given are in millimeters, but since these are relative values, small fractions of an inch will work as well. It is only necessary for each line to be given a value proportional to every other line in the drawing. I chose the starting point for this drawing to be at the left-hand side, as indicated by Fig. 5-21. You could have just as easily chosen the right-hand side, but we are more accustomed to working from left to right.

While it would be possible to include all the commands to reproduce this drawing in a single DRAW statement, I will divide the task up into several DRAW statement lines for clarification. You may wish to proceed in a way that requires fewer program lines—once you understand the process.

Since the first command in the DRAW statement generally starts at the center of the screen, we first insert a BL100 to move the graphic cursor to a point near the left-hand side of the screen. Now, beginning at the point marked START, we use an

R10 command to draw a line 10 relative points to the right. The next command is U4, which moves up 4 relative points to an intersection. My intention here is to draw the intersecting line first, and then proceed. The next command is R10, but I am now out of the first rectangle, which must still be drawn. Therefore, I retrace my steps with a BL10 command, which moves 10 blank places back left to the original point of intersection. This side of the rectangle is 17 relative points in length, the first four of which have already been drawn by a previous command; I used U13 (17-4) to complete the long side. It is now necessary to move 10 points to the left, but first there is another intersection 2 points in; the next command is L2. This is followed by U18, which draws the solitary vertical line. I now arrive back at the intersecting point with a DB18, and complete the remainder of the top horizontal portion with L8. The remaining vertical side is drawn with D17. At this point the graphic cursor is back at the beginning, and we can now draw the second rectangle at the end of the horizontal line drawn earlier. Perhaps the least confusing way to get there is with this combination:

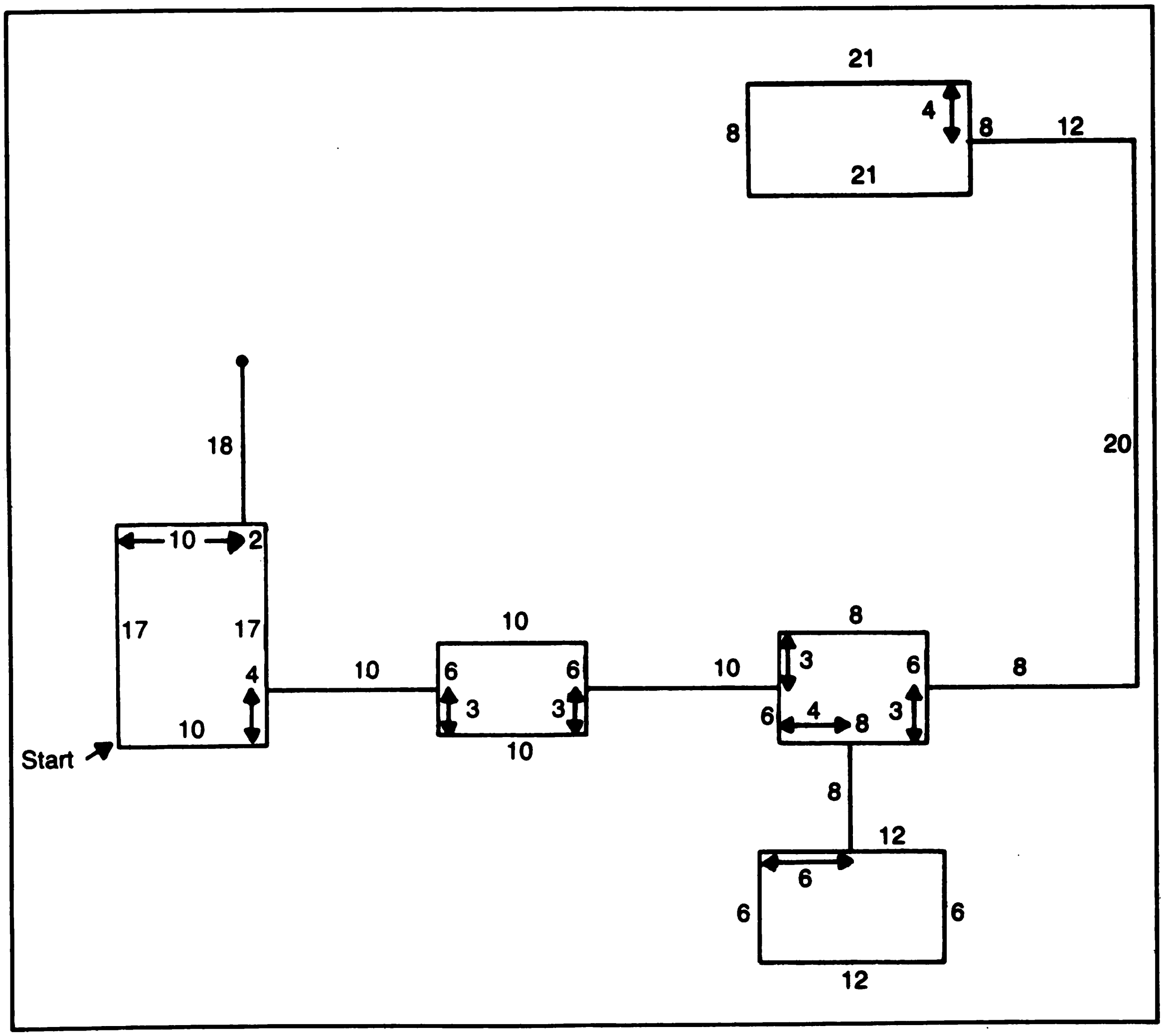


Fig. 5-21. Laying out the block diagram to get coordinates for DRAW Statement Commands.

```
BR10 BU4 BR10
```

The graphic cursor is now at the point on the second block intersected by the connecting line from the first. This side of the new block is 6 relative units in height, with the intersection point at the center. The next combination draws the second block and the outgoing horizontal line that connects it to the third block:

```
U3 R10 D6 L10 U3 BR10 R10
```

What I've done here is draw up 3, then right 10, down 6, left 10, and up 3. This takes me back to the beginning point for this block, but I'm on the wrong side to continue. The BR10 moves the graphic cursor from the center of the leading block edge to the center of the trailing edge, without plotting points. The final command, R10, draws the horizontal line that intersects the next block.

This is a very simple process once you understand the use of the DRAW statement and have mapped out a rough sketch. Let's look at the program itself. Line 20 draws the first rectangle and its connecting line. Line 30 draws the next block and connects it to the third. Line 40 draws the third and connects it to the one on the bottom, which is drawn in line 50. Line 60 draws the line that exits block 3 and extends upward to the overhead block. This last block is drawn in line 70. The entire program in only eight lines long, including the optional END statement.

Admittedly, the original drawing was quite simple, but it would probably take you much longer to draw neatly with pencil and paper. It took me about five minutes to prepare the freehand sketch used to write this program, including measurement time. Once the sketch was completed, it only took a few more minutes to input the program. In a drawing this simple it's usually not necessary to check each DRAW statement line before continuing to input the program. The method is so simple that the program should work the first time, and it often does. Of course, if you make an error or two, the debugging process usually involves shifting a few commands.

You may be thinking that a drawing such as this one is simple, but that it is much more difficult to draw a complex piece of art. This is not really true. As long as the artwork is limited to straight lines and angles of 0, 90, 180, and 270 degrees, the process is handled in exactly the same manner as that shown here. It may take you more time to sketch the drawing, obtain relative lengths, and input the program, but the procedure is the same.

Program 37: Three-Dimensional House

It is impossible to draw a true three-dimensional figure on a flat piece of paper—unless you want to include the microscopic height of the graphite. Three-dimensional drawings, then, are really simulations of three-dimensional objects using a medium limited to only two dimensions. This applies to drawings on paper as well as graphic representations on a computer screen.

Figure 5-22 shows a drawing of a house, with extra rectangles to represent solar collectors. These provide us with more objects to reproduce on the computer screen. At first glance, this may seem like a simple object to draw. To draw the bottom front of the house, you would simply place a graduated straightedge between the beginning and ending of the line, and arrive at a relative figure of about 50 units. Assuming that you start at the left front corner of the house, you would then specify a command of R50 to produce the line. But how about drawing the bottom of the left side of the house? This line is about 35 units long, but it travels away from the front bottom edge at an obtuse angle. If you return to the starting point and input an L35 command, you would simply add length to the first

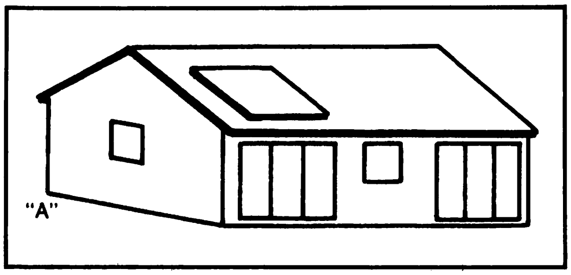


Fig. 5-22. Three-dimensional drawing of a house.

```
10 REM 3-D HOUSE
20 INPUT"TYPE IN THE SCALE FACTOR(1-10):";X
30 IF X>10 THEN 20
40 CLS
50 SCREEN 1,0
60 A=140
70 B=130
80 Y=X/4
90 COLOR 8,1
100 DRAW"S=X;BM140,130 R51 U12 L50 BL2 D12 NU13"
110 LINE(A+2*Y,B-12*Y)-(A-8*Y,B-24*Y)
120 DRAW"S=X;R50"
130 LINE(A+42*Y,B-24*Y)-(A+52*Y,B-12*Y)
140 DRAW"S=X;BM140,130 BL21 BU4 U12 L1"
150 LINE(A-24*Y,B-16*Y)-(A-8*Y,B-24*Y)
160 LINE(A-22*Y,B-4*Y)-(A,B)
170 LINE(A+8*Y,B-1*Y)-(A+20*Y,B-11*Y),,B
180 LINE(A+23*Y,B-6*Y)-(A+28*Y,B-11*Y),,B
190 LINE(A+32*Y,B-1*Y)-(A+44*Y,B-11*Y),,B
200 DRAW"BM140,130 S=X;BR36 BU1 NU10 BR4 NU10"
210 DRAW "S=X;BM140,130 BR12 BU1 NU10 BR4 NU10"
220 DRAW"S=X;BM140,130 BL8 BU7 U5 BL4 BU1 D5"
230 LINE(A-8*Y,B-7*Y)-(A-13*Y,B-8*Y)
240 LINE(A-8*Y,B-12*Y)-(A-12*Y,B-13*Y)
250 DRAW"S=X;BM140,130 BU14 BR7 NR10 BU6 BL4 R10"
260 LINE(A+7*Y,B-14*Y)-(A+3*Y,B-20*Y)
270 LINE(A+17*Y,B-14*Y)-(A+12*Y,B-20*Y)
```

Program 37. Three-Dimensional House.

line drawn. You can't easily use any form of the DRAW statement here, because the angle needed isn't one that can be specified. How is this overcome? The answer is that you don't, at least within the framework of the DRAW statement. However, in GWBASIC we have other statements available.

Let's take another look at what we're faced with by referring to Fig. 5-23. This is the same house, but horizontal lines have been drawn through it parallel to the bottom of the screen. The far end of the left side of the house (A) is elevated a short distance compared to the left corner of the front of the house (B). Once the front is drawn, a LINE statement could be used to draw a line from the coordinates of A to the coordinates of B. We don't have to worry about angles here.

For the sake of discussion, assume that point B is located at coordinates 140,130 on the screen. Assume also that the height from the reference line drawn from border to border through point B is 4 relative units. This means that the vertical coordinate of A will be equal to the vertical coordinate of B minus 4 units. In this example, the vertical coordinate of A would be 126 (130 − 4). The horizontal coordinate is obtained by measuring the distance from point B to the dotted line starting at A, a line drawn parallel to the edges of the frame. In the example, the distance between point B and the dotted line is about 23 relative units. Therefore, the horizontal coordinate of A is equal to the

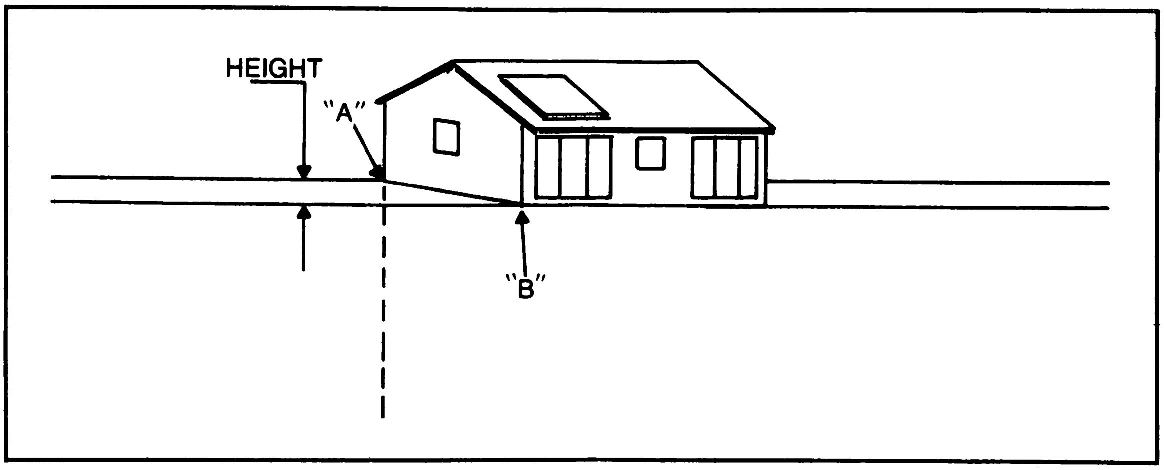


Fig. 5-23. Mapping out the angles and lengths for reproducing the house.

horizontal coordinate of B minus 23. (If we were trying to determine a coordinate to the right of B, the unit figure would be added instead of subtracted.) By using simple math, we can easily arrive at the coordinates of point A, which are 140 - 23,130 - 4 or 117,126.

To draw the bottom line of the left side of the house, then, we would first draw the reference line (the front bottom portion of the house) and then include this statement:

```
LINE (117,126) - (140,130)
```

This produces exactly the desired results. This same method will be used to generate any lines that approach any other lines (which may be produced using DRAW statements) at an angle. Note that it is only necessary to measure physical line lengths with a graduated straightedge when the DRAW statement is used. The length of the lines generated in LINE statements is determined by the distance between the sets of coordinates, and may be calculated with the extrapolation process using the borders as guides.

Of course, there are many objects in this scene that are formed by horizontal and vertical lines. These can be detected by laying a straightedge horizontally or vertically across the drawing and noting which lines fall along it. Each time you find a line that conforms to the straightedge, pencil in a line across the entire picture. This will be used to draw lines using the DRAW statement.

Figure 5-24 shows what the drawing will look like once the search has been completed. You can see that the top of the roof, the lower roof edge, the bottom front of the house, the tops and bottoms of the windows, and the solar collectors conform to horizontal lines. The vertical window sections and the three visible vertical edges of the house conform to vertical lines. All of these may also be drawn using the DRAW statement. Anything excluded from this graph-like pattern must be produced using other graphic statements.

The way I normally proceed at this point is to produce everything I can using DRAW statements. This involves measuring the physical lines of the original drawing. If possible, it is best to make a copy of the original drawing (several, in fact) and then draw the borders. Naturally, you measure the distance from the last point drawn to the start of the next portion to be drawn, using B commands to move the graphic cursor without drawing lines. When all DRAW statements have been completed, you can then run the incomplete program and compare the graphic representation with the original drawing. You should see all the lines that have been marked with horizontal and vertical lines.

Now the hard part begins. To draw the diagonal lines, you must treat each one as a separate entity. Refer back to Fig. 5-23. In order to draw the bottom left-hand edge of the house, it is necessary to position the graphic cursor at point B in this drawing. The statement to accomplish this is:

```
DRAW "BM140,130"
```

which are the original coordinates of point B. The process used to determine the coordinates of A when the coordinates of B are known has already been discussed, but reread these directions if you don't have a clear understanding of the process.

The new DRAW statement has returned the graphic cursor to the coordinates specified, so all that's necessary now is to input the following line:

```
LINE (117,26) - (140,130)
```

Actually, the new DRAW statement just discussed is not mandatory, nor even desirable from a programming standpoint. It is not necessary to return the cursor, since both coordinate sets (A and B) are specified in the new LINE statement; it was used because it better explains what we're trying to do. Assuming you have input the new DRAW statement, you can shorten the new LINE statement to:

```
LINE -(117,26)
```

This will draw a line from the last position reference to the coordinates specified in parentheses. The new DRAW statement returns the cursor to B, and the abbreviated LINE statement then draws a line from B to A.

Let's move on to another portion of the drawing. Assume you want to connect the left ends of the horizontal solar panel lines. How do we determine the coordinates of the left end of each line? Technically, the information is contained in the DRAW statements used to generate these lines, but you will have to go back through the program and begin adding and subtracting to get the numbers. Another method is shown in Fig. 5-25. We already know the coordinates of B, so let's figure the coordinates at the left end of the top solar panel line, labeled C in Fig. 5-25. Again, key horizontal and

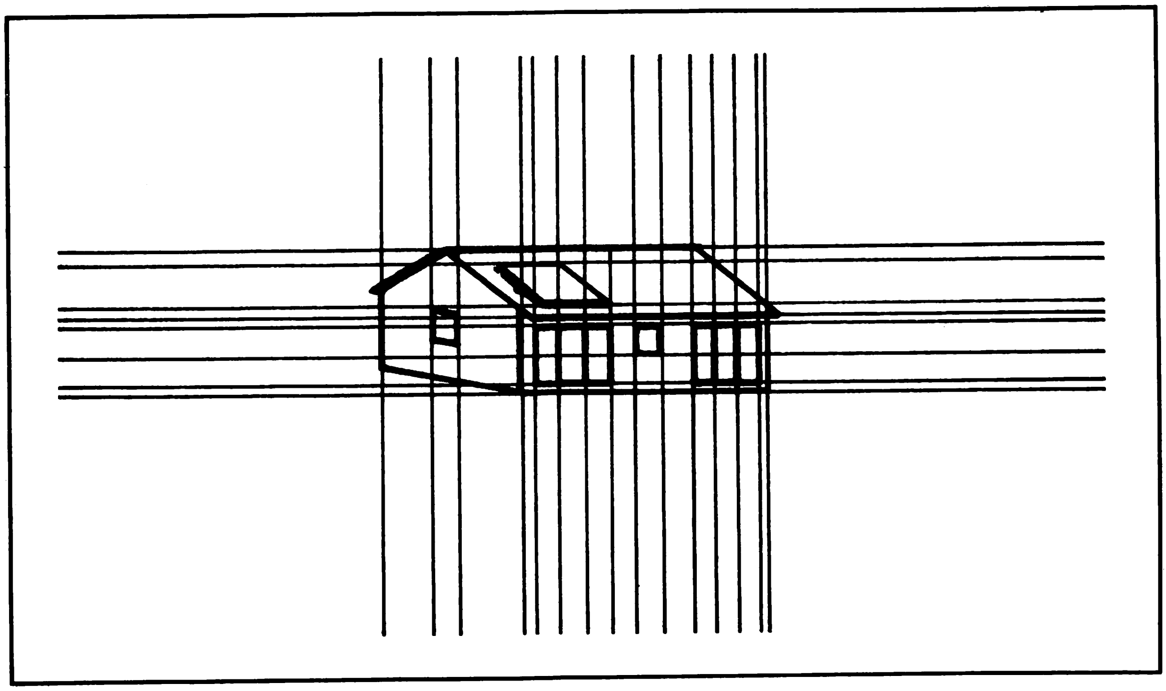


Fig. 5-24. Nearly finished house worksheet showing intersecting lines.

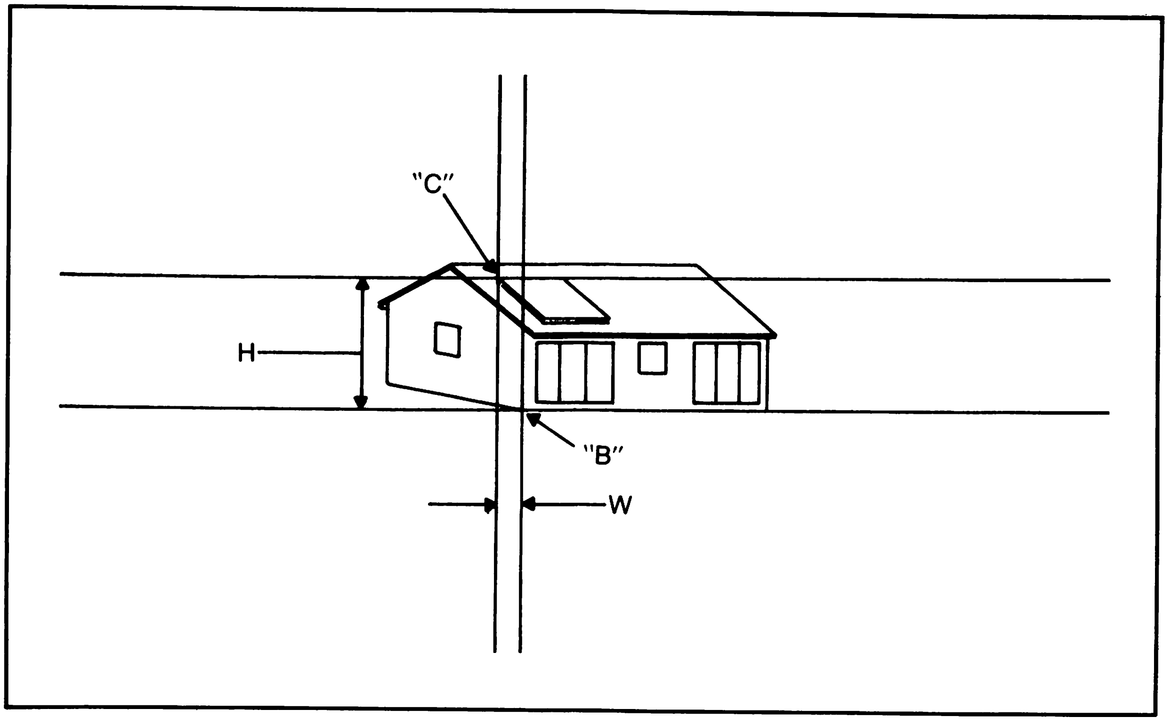


Fig. 5-25. Another method of determining coordinates.

vertical lines are first drawn. Next, we measure the distance from B to the vertical line labeled W, which is about 4 relative units. Now measure the height (H) from the bottom line to C. This will be about 20 relative units. These are then subtracted from the coordinates of B, because the new coordinates lie to the left of and above B. This will give us the coordinates of point C, which are 140 – 4 and 130 – 20, or 136,110.

It is now necessary to determine the coordinates of the left end of the bottom line, which is done by drawing a vertical line through it. In this case, the end point lies to the right of the horizontal coordinate of B, which means the horizontal distance (about 3 units) must be added to coordinate 140. This point also lies above B, so the vertical distance (about 14 units) is subtracted from 130. The new coordinates, then, are 143,116. Since our purpose is to connect the left ends of both lines, the following statement is used:

```
LINE(136,110) - (143,116)
```

To connect the right ends of the two horizontal lines, the process is much simpler. Since the coordinates of the left ends are known, all you have to do is locate the R commands in the DRAW statement that were originally used to produce these horizontal lines. These should be R12 or thereabouts. The right ends will still be at the same vertical coordinates as the left ends. All you do is add 12 relative units to the horizontal coordinates of the left ends. The following command will connect the two lines:

```
LINE(148,110) - (155,116)
```

Drawing every other diagonal line in this picture is handled in the same manner. There is one other problem that occurs when DRAW statements are used in conjunction with LINE statements or, for that matter, with any other graphic commands. In previous programs, the scale command was used to vary the relative size of the object being drawn. However, varying the scale of the DRAW statement will affect the size of only those lines produced by the DRAW statement. The LINE statement coordinates will remain the same. For this reason, the two must be coordinated. Within each DRAW statement there should be an S=X command, with a previous program line specifying the value of X as 4. When 4 is used, the DRAW statement command numerals (and thus the line lengths) correspond exactly with screen points. If you wish to enlarge the object, try doubling or tripling the values used in the DRAW statements. Appropriate changes must then be made to the LINE statements as well—not double the coordinate numbers each time you double the numbers in the DRAW statements, but merely add or subtract the new DRAW command numerals within the LINE statements. For example, it has already been established that the horizontal distance between B and C is 4 relative units. If you want to double the size of the drawing, this distance would be 8; therefore 8 must be subtracted from the horizontal coordinate of B. You can figure the coordinates in a slightly simpler manner by making all LINE statement coordinates relative. For instance, instead of using coordinates 136,110 to draw a line starting at point C, you would use the original coordinates of B, as follows:

```
LINE(140-4,130-20)
```

This line assumes that you used the actual line lengths in the original drawing. Now assume you want to double the size of the drawing. To accomplish this within the DRAW statements, you would include S=X, preceded by a program line establishing the value of X as 8. This would double the values of the numeric designators used in the DRAW statement. You would then go back to the LINE statement and EDIT as follows:

```
LINE(140-(4*2),130-(20*2) )
```

The following lines will modify your program so that all coordinates in LINE statements would automatically correspond to the coordinates at the end points of lines created by the DRAW statements, regardless of the scale factor used in the latter.

```
10  SCREEN 1
20  INPUT X
30  Y=X/4
40  DRAW"BM140,130S=X;U4R4D8R16
      . . . etc.
50  LINE(140-(4*Y),130-(20*Y))
```

Line 20 allows you to input a different value for X each time the program is run. This value is then inserted into the DRAW statement in line 40. Obviously, many DRAW and LINE statements will be required to produce the picture under discussion, but the methodology shown here for illustration purposes applies regardless. The program line shown here multiplies the differences between each of the coordinates for the point of origin (B) and the coordinates for C by the value of Y. If S is equal to 4(X=4), this amount will be divided by 4 in line 30. The value of Y will then be 1, and the line coordinates designated will not change. However, if the value of X changes to 8 (in effect, doubling the command numeral in each DRAW statement), the value of Y will be 2, so the line coordinates will be modified accordingly. Using this system, it is necessary to reference all LINE statements coordinates to the starting point B. This means that in some instances it will be necessary to add to or subtract from the horizontal coordinates only, from vertical coordinates only, or from both. It is quite difficult, or at least time-consuming, to draw a line on the right side of the screen based on the coordinates of point B on the left. You can, however, use partial LINE statements to start a line at the last point drawn on the screen. This is a case of mix and match, where no two computerists will arrive at the same picture in exactly the same way. There are just too many different ways of accomplishing the same objective. After a bit of experimentation, you will find that you are leaning toward one particular method or set of methods; if you feel comfortable with this method, fine. If not, you may wish to continue experimenting until you arrive at a system that is better suited to your own thought processes.

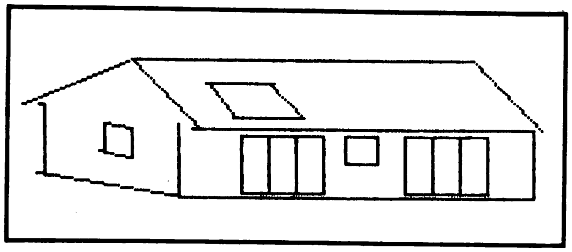


Fig. 5-26. The finished screen write of the three-dimensional house.

Figure 5-26 shows the house as it appears when this program is run.

Section Summary

This section has dealt with the methods used to draw complex pictures on the screen. DRAW is a very simple or basic statement as far as power is concerned. By comparison, the CIRCLE statement is far more powerful because it requires less information to do its job. The simplicity of DRAW, however, can be a plus, because the statement assumes nothing and does exactly what we tell it to do. The DRAW statement is quite often used to perform screen writes that can't be done using any of the other graphics statements, and you will frequently see it used with other graphics statements to write the most efficient program for the on-screen task.

CURSOR CONTROL PROGRAMS

By now, you have probably guessed that a fair amount of estimating is involved in initially drawing an object on the screen. The screen is composed of so many different coordinates that it's hard to visualize exactly where on the screen a certain point lies, judged by its coordinates alone. We do know that coordinates 160,100 identify a point at the exact center of the medium-resolution graphics screen. Therefore, we know that if the X coordinate is less than 160, it falls to the left of the screen center, or if it's more than 160, to the right. Likewise, if the Y coordinate is more than 100, it lies above screen center; if more than 100, it is below screen center. Beyond that it's still hard to know exactly the location of a point.

In text mode there is a text cursor that can be moved across the screen, using the keys designed for this purpose. This lets us know exactly where text will be written as we're typing it in via the keyboard. Through programming, we accomplish much the same thing, here in the form of a graphics cursor. The programs outlined in this section will generate a graphics cursor that can be moved via the keyboard. It will allow you to move a cursor to any point on the medium-resolution graphics screen.

Program 38: Keyboard Cursor Control

This program generates a graphic cursor that is a single white dot, and allows it to be moved over the screen using four of the standard keys on the keyboard. The four keys chosen are U, L, R, D, standing for up, left, right, and down, respectively. PUT and GET statements are used to move the cursor.

Here's how the program works. Line 60 uses PSET to generate a single white dot at screen location 0,0. Line 70 establishes a small array (A) to hold the dot. The GET statement is used in line 80 to retrieve the image of the dot. It's then erased from the screen when it is PUT to itself in line 90.

Multiple statement program line 100 assigns X the value of 160 and Y the value of 100. You will recall that these coordinates represent the center of the medium-resolution screen. Line 110 uses PUT again to display the dot at these center point coordinates. When this program is first run, you will almost immediately see a dot at the center of the screen.

In order to move the dot, line 120 uses the INKEY$ variable to read the keyboard. Line 130 tests for a condition of no keyboard input, simply looping back to line 120 until a keyboard input is received. Lines 140 through 170 test for the keyboard keys mentioned earlier, which will be used to control the on-screen dot. Line 180 branches back to the INKEY$ variable in line 120 when a key other than U, L, R and D is pressed on the keyboard. Notice in lines 140 through 170 that the logical OR operator is used. This means that the

```
10 REM KEYBOARD CURSOR CONTROL
20 REM COPYRIGHT FREDERICK HOLTZ
30 CLS
40 SCREEN 1
50 KEY OFF
60 PSET(0,0),3
70 DIM A(10)
80 GET(0,0)-(0,0),A
90 PUT(0,0),A
100 X=160:Y=100
110 PUT(X,Y),A
120 A$=INKEY$
130 IF A$="" THEN 120
140 IF A$="U" OR A$="u" THEN 190
150 IF A$="L" OR A$="l" THEN 240
160 IF A$="R" OR A$="r" THEN 290
170 IF A$="D" OR A$="d" THEN 340
180 GOTO 120
190 PUT(X,Y),A
200 Y=Y-1
210 IF Y<0 THEN Y=199
220 PUT(X,Y),A
230 GOTO 120
240 PUT(X,Y),A
250 X=X-1
260 IF X<0 THEN X=319
270 PUT(X,Y),A
280 GOTO 120
290 PUT(X,Y),A
300 X=X+1
310 IF X>319 THEN X=0
320 PUT(X,Y),A
330 GOTO 120
340 PUT(X,Y),A
350 Y=Y+1
360 IF Y>199 THEN Y=0
370 PUT(X,Y),A
380 GOTO 120
```

Program 38. Keyboard Cursor Control.

various branches will occur based upon a lowercase or uppercase version of the four control keys.

For discussion purposes, let's assume the first key pressed is U. Line 140 detects the U and branches to line 190. This line uses the PUT statement to erase the dot from the center of the screen. Line 200 then reassigns Y to its former value, less 1. Y is now equal to 99. (Skip line 210 for now.) Line 220 then uses PUT to place the dot at new position 160,99. This lies one pixel above the center of the screen. The U key has indeed caused the cursor to move up. Line 230 branches back the INKEY$ variable in line 120, and the computer waits for another key to be pressed. If you simply hold the U key down, Y is decremented by 1 each time the routine is accessed; the cursor will move rapidly up the screen, erased from its former position by PUT and then placed in a new position, also by PUT.

Now for line 210. Line 210 tests for a position of Y being decremented past the top of the screen to a value of −1, for instance. This would result in an "illegal function call" error, so line 210 reassigns Y a value of 199 (the bottom of the screen) if it passes through 0 into a negative number. In other words, if you continue holding down the U key the cursor will go to the top of the screen, and then reappear at the bottom and keep moving up.

If you press the L key, there is a branch to line 240. Again, the dot is PUT to itself, erasing it from the screen. X is then decreased by 1. Line 260 is a safety line, just like line 210. Line 270 PUTs the cursor at the new coordinates. Assuming a starting point of 160,100, the first press of the L key will cause a dot to be placed at 159,100, just to the left of center. There is then a branch back to line 120 so that another key may be read. In this case, if the cursor is advanced to the far left of the screen at coordinates 0,100, for instance, the next press will cause the cursor to appear at the far right of the screen at coordinates 319,100. The routines starting in lines 290 and 340 move the cursor right and down, respectively. These routines are just like the two previously discussed, except that X and Y are incremented rather than decremented by 1 each time the appropriate key is pressed.

The program is not very practical, but it does effectively demonstrate the principles behind cursor movement. To add a bit of practicality, change line 120 to:

```
120 LOCATE 1,1:PRINT X","Y
```

Then add line 121:

```
121 A$=INKEY$
```

Now when you run the program, the X-Y

```
10 REM MOVING THE CURSOR
20 REM COPYRIGHT
FREDERICK HOLTZ
30 CLS
40 SCREEN 1
50 KEY OFF
60 PSET(0,0),3
70 DIM A(10)
80 GET(0,0)-(0,0),A
90 PUT(0,0),A
100 X=160:Y=100
110 PUT(X,Y),A
120 KEY(11) ON
130 KEY(12) ON
140 KEY(13) ON
150 KEY(14) ON
160 ON KEY(11) GOSUB 210
170 ON KEY(12) GOSUB 260
180 ON KEY(13) GOSUB 310
190 ON KEY(14) GOSUB 360
200 GOTO 160
210 PUT(X,Y),A
220 Y=Y-1
230 IF Y<0 THEN Y=199
240 PUT(X,Y),A
250 RETURN
260 PUT(X,Y),A
270 X=X-1
280 IF X<0 THEN X=319
290 PUT(X,Y),A
300 RETURN
310 PUT(X,Y),A
320 X=X+1
330 IF X>319 THEN X=0
340 PUT(X,Y),A
350 RETURN
360 PUT(X,Y),A
370 Y=Y+1
380 IF Y>199 THEN Y=0
390 PUT(X,Y),A
400 RETURN
```

Program 39. Another Cursor Movement Program.

coordinates of the cursor will be displayed in the upper left-hand corner of the screen. This will at least give you some idea of where the cursor lies in the terms of X and Y.

Program 39: Another Cursor Movement Program

This program is identical to the previous one, except that it uses the standard cursor control keys to manipulate the cursor instead of the four letter keys. Everything is as it was before down to line 120. Lines 120 through 150 activate keys 11 through 14 by using the KEY statement in GWBASIC. Keys 11 through 14 represent the cursor control keys that control up, left, right, and down cursor movement. The KEY ON lines simply turn these keys on. Lines 160 through 190 replace the previous INKEY$ variable. The ON KEY statement causes the computer to branch to another program line as soon as one of the specified keys is pressed. From this point on, the movement routines are exactly as they were in the previous program, although in this case, we have set them up as subroutines using GOSUB statements in lines 160 through 190. The previous GOTO branches in the movement routines have been replaced with RETURN statements.

Program 40: Graphics Drawing Board

This program makes full use of the previous cursor movement program principles. This is a practical program that will allow you to draw simple objects on the screen simply by moving the cursor to a desired point, and then pressing another key or two.

Lines 110 through 190 turn on nine keys that will be used for cursor movement and drawing circles, lines, dots, and boxes on the screen. Keys 11 through 14 are our standard cursor movement keys. Keys 1 through 4 correspond to keys F1 through F4, which are accessed by means of the Fn key on the keyboard. F1 is used to draw squares or rectangles, F2 is used to draw circles, F3 is used to paint these objects, and F4 is used to draw lines or dots. Key F10 is used to move the cursor more rapidly. (The previous cursor movement programs moved the cursor by incrementing X or Y by 1. In this program, when F10 is pressed the cursor is moved in steps of 5. To get back to one-step control, simply press F10 again.)

Now, to draw a box on the screen, advance the cursor to the position on the screen where you want the edge of the box to begin. Press F1. This will produce a blue dot marking one corner of the box. Assuming this is the left upper corner, move the cursor down and to the right to a point where you want the lower right corner to be. Press F1 again and your box appears on the screen. With this routine you can draw different-sized boxes all over the screen, and even draw boxes within boxes with extreme ease. Here's how it works.

When F1 is pressed, line 220 branches to line 320. Here, variable C is incremented by 1. Previously, C was 0. Line 330 tests for the value of C and, since it is equal to 1, there is a branch to line 340. Here, variables A and B are assigned the values of X and Y (the screen location of the cursor) and line 350 sets a dot on the screen. This marks the corner of the box. Line 360 branches back to one line past the GOSUB statement, and the computer again waits to read another key. You can now move the cursor to another position marking the opposite corner of the box. When you press F1 again there is another branch to line 320, and here C is incremented by 1. The former value of C was already 1, so C is now equal to 2. Line 330 branches to line 370, since C is no longer equal to 1. This line assigns AA and BB the values of X and Y. Remember, X and Y have now changed from the previous coordinates because the cursor was moved. Line 370 also returns C to a value of 0. Lines 380 through 450 compare the value of AA to A and, depending on the condition detected, assign the proper coordinates to X and Y to erase the first dot from the screen. Line 450 uses the graphics LINE statement with the B designator to draw the box between the coordinates specified. Line 460 contains a RETURN statement that branches back to the key detection routine.

To draw a circle, you simply press the F2 key and then move the cursor up or down. The first

```
10 REM GRAPHIC DRAWING BOARD
20 REM COPYRIGHT FREDERICK HOLTZ
30 CLS
40 SCREEN 1
50 KEY OFF
60 ST=1
70 DIM A(100)
80 PSET(0,0),3
90 GET(0,0)-(0,0),A
100 PUT(0,0),A
110 KEY(1) ON
120 KEY(2) ON
130 KEY(3) ON
140 KEY(4) ON
150 KEY(10) ON
160 KEY(11) ON
170 KEY(12) ON
180 KEY(13) ON
190 KEY(14) ON
200 X=160:Y=100
210 PUT(X,Y),A
220 ON KEY(1) GOSUB 320
230 ON KEY(2) GOSUB 470
240 ON KEY(3) GOSUB 710
250 ON KEY(4) GOSUB 800
260 ON KEY(10) GOSUB 780
270 ON KEY(11) GOSUB 550
280 ON KEY(12) GOSUB 590
290 ON KEY(13) GOSUB 630
300 ON KEY(14) GOSUB 670
310 GOTO 220
320 C=C+1
330 IF C=1 THEN 340 ELSE 370
340 A=X:B=Y
350 PSET(X,Y),2
360 RETURN
370 AA=X:BB=Y:C=0
380 IF AA>A AND X<319 THEN X=X+1:PUT(X,Y ),A
390 IF AA<A AND X>0 THEN X=X-1:PUT(X,Y), A
400 IF AA>A AND X=319 THEN X=0:PUT(X,Y), A
410 IF AA<A AND X=0 THEN X=319:PUT(X,Y), A
420 IF AA=A AND Y>0 AND Y<199 THEN Y=Y-1 :PUT(X,Y),A
430 IF AA=A AND Y=0 THEN Y=199:PUT(X,Y), A
440 IF AA=A AND Y=199 THEN Y=0:PUT(X,Y), A
450 LINE(A,B)-(AA,BB),3,B
460 RETURN
470 D=D+1
```

```
480 IF D=1 THEN AAA=X:BBB=Y:PSET(X,Y),2:
RETURN
490 CCC=(BBB-Y)-1:D=0
500 IF Y>BBB THEN CCC=CCC*-1
510 IF Y=BBB THEN PSET(X,Y),0:RETURN
520 CIRCLE(AAA,BBB),(6/5)*CCC,3
530 PSET(AAA,BBB),0
540 RETURN
550 PUT(X,Y),A
560 Y=Y-ST:IF Y<0 THEN Y=199
570 PUT(X,Y),A
580 RETURN
590 PUT(X,Y),A
600 X=X-ST:IF X<0 THEN X=319
610 PUT(X,Y),A
620 RETURN
630 PUT(X,Y),A
640 X=X+ST:IF X>319 THEN X=0
650 PUT(X,Y),A
660 RETURN
670 PUT(X,Y),A
680 Y=Y+ST:IF Y>199 THEN Y=0
690 PUT(X,Y),A
700 RETURN
710 PAINT(X,Y+3),COL,3
720 A$=INKEY$
730 IF A$="" THEN 720
740 IF A$<>CHR$(13) THEN COL=COL+1 ELSE 770
750 IF COL>3 THEN COL=0
760 GOTO 710
770 RETURN
780 IF ST=5 THEN ST=1 ELSE ST=5
790 RETURN
800 CT=CT+1
810 IF CT=1 THEN A=X:B=Y:PSET(A,B),2:RETURN
820 AA=X:BB=Y
830 IF AA>A AND X<319 THEN X=X+1:PUT(X,Y ),A
840 IF AA<A AND X>0 THEN X=X-1:PUT(X,Y), A
850 IF AA>A AND X=319 THEN X=0:PUT(X,Y), A
860 IF AA<A AND X=0 THEN X=319:PUT(X,Y), A
870 IF AA=A AND Y>0 AND Y<199 THEN Y=Y-1 :PUT(X,Y),A
880 IF AA=A AND Y=0 THEN Y=199:PUT(X,Y), A
890 IF AA=A AND Y=199 THEN Y=0:PUT(X,Y), A
900 LINE(A,B)-(AA,BB),3
910 CT=0
920 RETURN
```

Program 40. Graphics Drawing Board.

time you press F2, a dot will be placed on the screen that marks the *center* of the circle. When you press F2 a second time, you mark a point one pixel past the circle *perimeter.* The circle is drawn based upon these coordinates. To paint the circle, simply press F3 immediately after the circle is produced and then press any key on the keyboard other than the function or cursor control keys. The first press of the key will produce one color. Pressing it again causes a different color, and pressing it a third time produces the screen background color. When you're satisfied with the color, simply move on.

Pressing F2 brings about a branch to line 470, which increments D by 1. Line 480 tests for the value of D. The first time this routine is accessed D is equal to 1, so AAA and BBB are assigned the values of X and Y. Line 480 also contains the PSET statement, which sets the dot marking the center of the circle on the screen. When F2 is pressed again, D is incremented to 2. Therefore line 490 is executed, which assigns CCC a value that is equal to BBB – Y – 1. D is then reset to 0. CCC now represents the perimeter of the circle minus 1 pixel. The CIRCLE statement in line 520 uses AAA and BBB as the center of the circle, and CCC times 6/5 as the diameter. The 6/5 fraction is used to make the circle perfectly round. (Trust me.) Line 530 then erases the dot at the center of the circle. The RETURN statement puts the computer back in the key detection mode.

Pressing F3 sets up the routine to fill in the circle or other object. To do this you could place the cursor dot in the center of the object to be painted, but this will leave a black spot where the dot was. It is best to fill an object by placing the cursor dot 1 pixel above the object. Upon pressing F3 there is a branch to line 710, which uses the PAINT statement to fill in the object. The variable COL represents the color numeral. Line 720 uses the INKEY$ variable to increment COL by 1 each time a standard key is pressed. When you are satisfied with the color, simply move on to drawing another object.

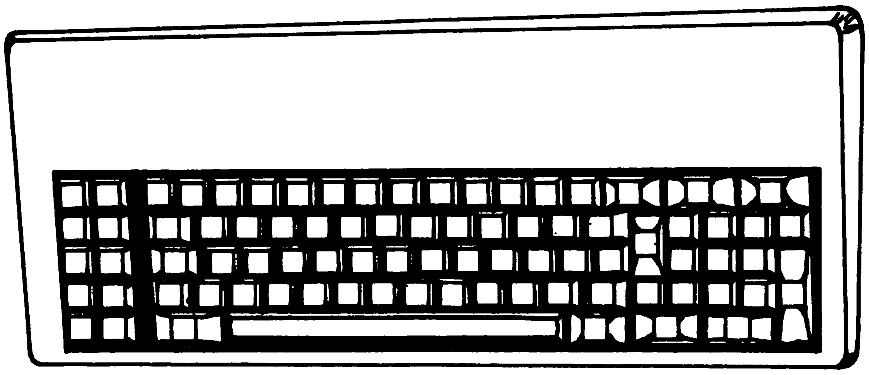
Key F4 is used to draw vertical, horizontal, or diagonal lines. Simply press F4 and the beginning line position dot is fixed. Move the cursor to any other spot on the screen, press F4 again and a line is drawn between the two points. To draw a single dot, press F4 twice. When F4 is pressed there is a branch to line 800, which increments CT by 1, and line 810 then tests for the value of CT. When CT is equal to 1, it assigns A and B the values of X and Y respectively. The PSET statement on this same line produces a dot on the screen that marks the starting point for the line. The second time this routine is accessed, CT is stepped to 2 and line 820 assigns AA and BB the new values of X and Y. Lines 830 through 890 compare the value of AA to A to calculate the position of the original dot, which is then erased. The LINE statement in line 900 then draws a line between the two sets of coordinates. CT is then reset to 0, and the RETURN statement puts the computer back in the key-detect mode.

This program was a bit difficult to write. It took several hours, but most of this involved debugging some simple mistakes and adding some built-in user features. It can be used effectively to draw some very simple objects on the screen, but as these simple objects multiply, so does the complexity of the screen image. You can even use this program to add features to graphics images that have been produced with other programs. To do this, simply remove the CLS statement from line 30. Run your other graphics program and then load this one. You can then draw on the surface which was produced by the first program.

Section Summary

Cursor movement programs make excellent "tools" for the graphics programmer. With them, you can visually pinpoint the screen locations where objects are to be drawn. Such programs can be used almost like a human-controlled stylus to actually draw on the screen, much as you would draw on a piece of paper.

Chapter 6



AT&T PC Game Programs

While game programs are more often associated with home computers than personal computers, it is a fact that many of the best games designed for microcomputers are meant to run on personal computers. The home computer is usually thought of as a low-level microcomputer with limited screen capabilities, while the personal computer is ranked as a machine that might be most comfortable in an office or small business environment. In any event, most people will not pay the price of a personal computer simply to run game programs. For this reason, the PC-6300 will be supported more by business software than game software. From a tutorial point of view, however, game programs offer excellent examples of using various routines and computer functions to accomplish a goal. This chapter includes a number of text and graphics mode programs that fall into the game category. Many are quite fun for programmers of all ages, but more importantly, each will tell you more about programming a microcomputer. This chapter will introduce some new functions and statements that have not been used previously. These should give you ideas as to how they may be used in programming more sophisticated applications.

Program 41: Automatic RANDOMIZE Seed Number Generator

Many computer games depend on random numbers to offer the element of chance. We set this up on the PC-6300 by first using the RANDOMIZE statement followed by RND functions throughout the program. The RANDOMIZE statement, when executed, creates an automatic screen prompt telling you to type in a positive or negative number within a certain range. This one number is used by the computer to "shuffle the deck," so to speak—to mix up the random numbers available to us. However, this can be a real pain. It would be better if the computer could pick a number on its own. This program allows the computer to choose from a range of numbers between 0 and 60, which should be sufficient for most purposes.

We do this using a portion of the TIME$ func-

```
10 REM AUTOMATIC RANDOMIZE
SEED NUMBER
20 CLS
30 SCREEN 0
40 WIDTH 40
50 KEY OFF
60 X=VAL(MID$(TIME$,7,2))
70 RANDOMIZE X
80 FOR A=1 TO 10
90 PRINT INT(RND*20)+1
100 NEXT A
```

Program 41. Automatic RANDOMIZE Seed Number Generator.

tion, specifically the portion that displays seconds. The internal clock of the computer is constantly updating TIME$ second by second, so all we need to do is use the seconds digit(s) as the random number seed. Here's how the program works.

Line 60 uses the MID$ function to pull the seconds digit(s) from TIME$. The VAL function is also used to convert the string value returned into a numeric value that is assigned to X. Line 60 could be replaced with:

```
60 X$=MID$(TIME$,7,2,)
61 X=VAL(X$)
```

Line 60 does all this in a single program line. The MID$ function returns two characters, starting at the seventh position in TIME$. These characters make up the seconds portion of TIME$. Line 70 uses the RANDOMIZE statement with X as its seed number. Variable X will be equal to whatever the clock seconds were when line 60 was executed. After line 70 is executed, the random number generator has been reshuffled and is ready to go.

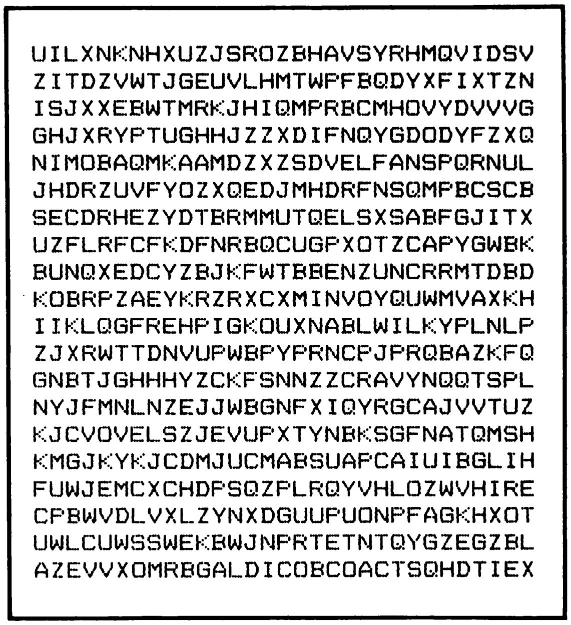
Lines 80 through 100 simply use the RND function to print random values to the screen. Each time the program is run, a different set of random numbers will appear. Lines 80 through 100 are for demonstration purposes only. The preceding lines are the ones that should be used in any program that uses the automatic seed generator.

Program 42: Random Word Maze Generator

Chances are you've played a game where a massive jumble of letters were arranged on a piece of paper. The object of the game is to find the hidden words that may be printed horizontally, vertically, or diagonally within the maze. The words may be printed from bottom to top, top to bottom, left to right, or right to left. This computer program generates such word mazes, but I've cheated a little bit, because no specific words have been programmed. All of the letters generated by the program are chosen purely at random. However, you might be surprised at the number of words that crop up randomly throughout such a display. The object of this game is to see how many words you can find within the maze. Your opponent has the same assignment, and whoever finds the most words is the winner. Figure 6-1 shows the sample maze that was printed at random. If you look closely, you will

```
10 REM RANDOM WORD MAZE
GENERATOR
20 REM COPYRIGHT FREDERICK
HOLTZ
30 CLS
40 INPUT"TYPE ANY NUMBER
FROM 1 TO 30000 ";NUM
50 RANDOMIZE NUM
60 CLS
70 SCREEN 0
80 WIDTH 40
90 DIM A$(26)
100 FOR X=65 TO 90
110 A$(X-64)=CHR$(X)
120 NEXT X
130 FQR Y=1 TO 20
140 FOR X=5 TO 35
150 Z=INT(RND*26)+1
160 LOCATE Y,X
170 PRINT A$(Z)
180 NEXT X
190 NEXT Y
```

Program 42. Random Word Maze Generator.



```
UILXNKNHXUZJSROZBHAVSYRHMQVIDSV
ZITDZVWTJGEUVLHMTWFFBQDYXFIXTZN
ISJXXEBWTMRKJHIQMPRBCMHOVYDVVVG
GHJXRYPTUGHHJZZXDIFNQYGDODYFZXQ
NIMOBAQMKAAMDZXZSDVELFANSPQRNUL
JHDRZUVFYOZXQEDJMHDRFNSQMPBCSCB
SECDRHEZYDTBRMMUTQELSXSABFGJITX
UZFLRFCFKDFNRBQCUGPXOTZCAPYGWBK
BUNQXEDCYZBJKFWTBBENZUNCRRMTDBD
KOBRPZAEYKRZRXCXMINVOYQUWMVAXKH
IIKLQGFREHPIGKOUXNABLWILKYPLNLP
ZJXRWTTDNVUPWBPYPRNCPJPRQBAZKFQ
GNBTJGHHHYZCKFSNNZZCRAVYNQQTSPL
NYJFMNLNZEJJWBGNFXIQYRGCAJVVTUZ
KJCVOVELSZJEVUPXTYNBKSGFNATQMSH
KMGJKYKJCDMJUCMABSUAPCAIUIBGLIH
FUWJEMCXCHDPSQZPLRQYVHLOZWVHIRE
CPBWVDLVXLZYNXDGUUPUONPFAGKHXOT
UWLCUWSSWEKBWJNPRTETNTQYGZEGZBL
AZEVVXOMRBGALDICOBCOACTSQHDTIEX
```

Fig. 6-1. Screen display from the Word Maze Generator program.

find many English words that have been randomly generated by this program.

The INPUT statement in line 40 allows you to type in any number to be used as a random seed. I purposely did not use the automatic random seed program, since it will be a good exercise for you to combine the two programs. Line 50 uses the RANDOMIZE statement with the number you input to reseed the random number generator. Line 90 establishes a string array (A$) which contains 26 characters, corresponding to the 26 letters in the English alphabet. Line 100 starts a FOR-NEXT loop that counts X from 65 to 90. ASCII character 65 is the letter A, 66 is B, and so on, until you reach 90, which is Z. These numbers will be used to generate all 26 letters of the alphabet. Line 110 puts the string value of the ASCII number into the array. The CHR$ function is used here. Array positions 1 through 26 are obtained by subtracting 64 from X in the same line. The CHR$ function converts X to a letter of the alphabet. When this loop times out, array positions 1 through 26 correspond with the 26 letters of the alphabet.

Lines 130 and 140 start two more loops. Here, Y represents the row position on the text screen, whereas X represents the column position. Line 150 uses the RND function to return a number at random. This number will range from 1 to 26. The LOCATE statement is used in line 160 to position the text cursor on the screen at the point determined by the values of X and Y. Line 170 then prints the letter found in the array position determined by random number Z. This program serves to print 20 rows of 31 characters on the screen. Each time you run this program, a different word maze will appear, as long as you use a different random seed number.

Program 43: Math Drill

Math Drill is an educational game that gives the user simple math problems to solve. If the answer is correct, the computer tells you so. If not, you are given the opportunity to try again.

Again it is necessary to input a random seed number. Lines 100 through 120 assign random numbers to numeric variables X, Y, and Z. X and Y may be equal to any number between 1 and 100, while Z may be equal to any number between 1 and 4. All numbers will be whole numbers (integers) because the INT function in these lines. The value of Z is used to determine whether the problem will involve addition, subtraction, division, or multiplication. Lines 130 through 160 read the value of Z and then assign A$ the proper symbol. Numeric variable CNS is assigned the value of the mathematical operation to be performed on X and Y.

Line 170 uses the LOCATE statement to position the text cursor at a point near the center of the screen. Line 180 then prints the value of random number X, followed by the mathematical function symbol (+, –, /, *), followed by the value of Y. The equal symbol is then printed. At this point, the user is expected to input the correct answer. Line 190 assigns this input to ANS. If ANS (the user's answer) is equal to CNS (the correct answer), line 200 causes the monitor speaker to beep. The screen is cleared in line 210, and then the problem and cor-

```
10 REM MATH DRILL
20 REM COPYRIGHT FREDERICK HOLTZ
30 REM
40 CLS
50 SCREEN 0
60 WIDTH 40
70 KEY OFF
80 INPUT"TYPE IN ANY NUMBER";SN
90 RANDOMIZE SN
100 X=INT(RND*100)+1
110 Y=INT(RND*100)+1
120 Z=INT(RND*4)+1
130 IF Z=1 THEN A$="+":CNS=X+Y
140 IF Z=2 THEN A$="-":CNS=X-Y
150 IF Z=3 THEN A$="/":CNS=X/Y
160 IF Z=4 THEN A$="X":CNS=X*Y
170 LOCATE 14,15
180 PRINT X A$ Y"=";
190 INPUT ANS
200 IF ANS=CNS THEN BEEP ELSE 300
210 CLS
220 LOCATE 14,15
230 PRINT X A$ Y"=" CNS
240 LOCATE 18,1
250 PRINT"CORRECT ANSWER!!!"
260 LOCATE 23,1
270 INPUT"DO YOU WISH TO CONTINUE(Y/N)";
C$
280 IF C$="Y" OR C$="y" THEN CLS:GOTO 10
0
290 IF C$="N" OR C$="n" THEN 430
300 CLS
310 BEEP
320 FOR DLAY=1 TO 200:NEXT DLAY
330 BEEP
340 LOCATE 14,10
350 PRINT ANS;"IS AN INCORRECT ANSWER!"
360 LOCATE 23,1
370 PRINT"TYPE 1 TO TRY AGAIN OR 2 FOR A
NSWER.
380 S$=INKEY$
390 IF S$="" THEN 380
400 IF S$="1" THEN CLS:GOTO 170
410 IF S$="2" THEN 210
420 GOTO 380
430 CLS
440 LOCATE 14,10
450 PRINT"THE MATH DRILL IS OVER."
460 END
```
Program 43. Math Drill.

rect answer are reprinted with a prompt telling you that the answer was indeed correct. If the user's answer is not equal to CNS there is a branch to line 300, where two beeps are heard; the two beeps are set up in lines 310 through 330. The computer beeps once in line 310. A slight time delay is set up in line 320 so that the two beeps do not simply run together and sound like one long beep. The computer then displays your answer on the screen, telling you that it is incorrect. You are then given the option of pressing 1 to try again or 2 to ask for the correct answer.

Other prompt lines throughout this program allow you the option of quitting at any time. It's a friendly program that is fun to play as a game, and if you're not careful you might learn something at the same time.

Program 44: Numbers Guessing Game

The Numbers Guessing Game is one that is quite common to computer game programming. It's quite simple but still quite popular, because it's a lot of fun for persons of all ages. At random, the computer chooses a number between 1 and 100. It is the player's duty to guess that number. With each wrong guess, the computer will display prompts telling the player whether the guess is too high or too low. When the number is finally guessed, the computer will prompt with the fact that the correct answer has been guessed, and will also tell the number of guesses it took. In a two-player version, the player with the lowest number of guesses is the winner. When a single player plays this game, the idea is to guess the number with as few guesses as possible.

Here's how the program works. The first five lines initialize the screen, and line 60 prompts you to input any number. (This is to reseed the random number generator.) The screen is again cleared, and the RANDOMIZE statement is used with numeric variable NS, the number you input. Line 100 establishes a value for numeric variable COUNT of 1. This variable will be used to keep track of the number of guesses you take. Line 110 assigns to A a random number from 1 to 100. Lines 120 through 170 print a few simple instructions and then allow you to make a first guess. Your guess is assigned to B in line 170. Lines 190 and 200 check

```
10 REM NUMBERS GUESS GAME
20 REM COPYRIGHT FREDERICK HOLTZ
30 CLS
40 WIDTH 40
50 KEY OFF
60 INPUT"TYPE IN ANY NUMBER.";NS
70 CLS
80 RANDOMIZE NS
90 CLS
100 COUNT=1
110 A=INT(RND*100)+1
120 PRINT"THE COMPUTER HAS CHOSEN A NUMB
ER"
130 PRINT
140 PRINT"BETWEEN 1 AND 100. WHAT DO YOU
 THINK"
150 PRINT
160 PRINT"THIS NUMBER IS";
170 INPUT B
180 CLS
```

```
190 IF B=A THEN 280 ELSE COUNT=COUNT+1
200 IF B<A THEN 260
210 PRINT"THAT'S TOO HIGH!! TRY AGAIN!!
220 FOR T=1 TO 1000
230 NEXT
240 CLS
250 GOTO 120
260 PRINT"THAT'S TOO LOW!! TRY AGAIN!!
270 GOTO 220
280 PRINT"THAT IS THE CORRECT NUMBER !!!
"
290 PRINT
300 BEEP
310 PRINT"IT TOOK YOU ";COUNT;" GUESSES!
!"
320 FOR T=1 TO 2000
330 NEXT T
340 CLS
350 INPUT"WOULD YOU LIKE TO PLAY AGAIN (
YES/NO)   ";A$
360 IF A$="YES" THEN 90
370 IF A$="NO" THEN 390
380 IF A$<>"YES" OR A$<>"NO" THEN 450
390 CLS
400 LOCATE 14,25
410 PRINT"THE GAME IS OFFICIALLY OVER."
420 LOCATE 16,28
430 PRINT"THANK YOU FOR PLAYING"
440 END
450 CLS
460 PRINT"YOU HAVE NOT RESPONDED WITH A
YES OR NO ANSWER!!!!
470 FOR T=1 TO 1000
480 NEXT T
490 CLS
500 GOTO 350
```

Program 44. Numbers Guess Game.

for the relationship of your guess to the actual number the computer has chosen. In line 190, there is a branch to line 380 if your guess (B) is equal to the secret number (A). Lines 280 onward tell you that the correct answer has been given, print the number of guesses to the screen, and then ask if you want to keep playing. However, if B is not equal to A in line 190, COUNT is incremented by one. Line 200 checks to see if your guess (B) is smaller than A. If this is true there is a branch to line 260, which prints the prompt "THAT'S TOO LOW!! TRY AGAIN!!" Lines 220 and 230 form a time delay loop. When this loop times out, the screen is cleared and there is a branch to line 120, where you are prompted to guess again.

Going back to line 200, if B is not less than A by process of elimination it must be more than A, because line 190 has already tested to see if B is equal to A. Assuming your guess is larger than the actual number (B), lines 210 through 250 are executed, telling you that your guess is too high. There is then a branch back to line 120, where you are prompted to guess again.

When the correct answer is finally guessed, there is a branch to line 280, where the correct number prompt is printed. A beep is also heard. Line 310 prints the number of guesses it took. Notice that the COUNT variable is included in line 310. Line 350 prints the prompt on the screen asking you if you'd like to play again. You must respond with a yes or no, which is assigned to A$. Line 360 checks for "yes," which then brings about a branch to line 90, and the program begins again. Line 370 checks for a "no" input and then branches in line 390, where the screen is cleared and the computer tells you the game is over and thanks you for playing. The program then ends. Line 380 checks for an erroneous input of a value other than yes or no. When this happens, the computer admonishes you for not inputting a yes or no answer and then asks you again if you would like to continue playing.

Program 45: Random Partner Matcher

This program is more of a game aid than a game itself. It will allow you to input the names of up to 20 boys and the same number of girls in any order. The computer will then rearrange the names randomly and display them on the the screen side by side. The idea here is that the girl whose name is listed on the left is matched with the boy on the right. This program was originally written to aid teachers in setting up dance partners for grade school "mixers." However, this program can also be used to match any two groups of people, items, or numbers in a random fashion.

In line 70, the random seed number is input, and in line 100 you are prompted to input the number of couples you wish to mix and match. Line 110 checks to make certain that there are no more than 20 couples. If so, there is a branch back to line 100, where you are asked to input this number again. The number of couples is assigned to I. Line 130 establishes two arrays, each of which will hold a maximum of 20 elements.

The name input loop for girls begins in line 140. Here, X is counted from 1 to the value of I (number of couples). Line 150 within the loop prompts you to input the name of a girl. This name is assigned to A$. If you input 20 couples when prompted earlier in the program, this loop will cycle 20 times. After each name is input, the screen is cleared before you are prompted to input the next name.

When the first loop times out a second loop is entered in line 180, which allows you to input the names of the boys in the same fashion as the previous loop. The boys' names are assigned to B$.

When this loop times out, the random shuffling routine is entered. Line 220 again counts X from 1 to the value of I, and line 230 assigns to Z a random number that will range from 1 to I. We'll skip line 240 for now.

As soon as the value of Z has been arrived at, line 250 places the text cursor at the left side of the screen, and line 260 prints the name of the girl held in array position A$(Z). Once the name has been printed on the screen, line 270 blots out the name of that girl from the array, replacing it with zero. (Notice that the zero is assigned to the string array surrounded by quotation marks. Numbers can be inserted into a string array only if enclosed in quotes. This makes them string values.) Line 280 recycles the loop.

We can now discuss line 240. It checks for an array position whose contents have already been read and printed to the screen. For instance, assume on the first pass of the loop that the random number 6 was chosen. The girl whose name was input to array position 6, or A$(6), has already been printed to the screen, and line 270 has reassigned the value of A$(6) to "0." Line 240 is there to prevent a name from being displayed twice, (or, more accurately, to branch back to the random

```
10 REM RANDOM PARTNER MATCHER
20 REM COPYRIGHT FREDERICK HOLTZ
30 CLS
40 SCREEN 0
50 WIDTH 40
60 KEY OFF
70 INPUT"ENTER ANY NUMBER FROM 1 TO 3000
0";NUM
80 RANDOMIZE NUM
90 CLS
100 INPUT"HOW MANY COUPLES";I
110 IF I>20 THEN 100
120 CLS
130 DIM A$(20):DIM B$(20)
140 FOR X=1 TO I
150 INPUT"NAME OF GIRL";A$(X)
160 CLS
170 NEXT X
180 FOR X=1 TO I
190 INPUT"NAME OF BOY";B$(X)
200 CLS
210 NEXT X
220 FOR X=1 TO I
230 Z=INT(RND*I)+1
240 IF A$(Z)="0" THEN 230
250 LOCATE X,1
260 PRINT A$(Z)
270 A$(Z)="0"
280 NEXT X
290 FOR X=1 TO I
300 Z=INT(RND*I)+1
310 IF B$(Z)="0" THEN 300
320 LOCATE X,15
330 PRINT B$(Z)
340 B$(Z)="0"
350 NEXT X
360 LOCATE 23,1
370 END
```

Program 45. Random Partner Matcher.

number routine and assign Z another random number if it returns one already used).

Therefore, the reassignment in line 270 works in conjunction with the test line in line 240. If line 240 detects a value of "0" for A$(Z), there is a branch to line 230, where Z is assigned another random number. This loop will continue until A$(Z) equals something other than "0." When all of the names have been read from A$ the loop times out, and lines 290 through 350 set up the same sequence for reading the names of the boys. The boys' names are printed to the right of the girls' names, beginning near the center of the screen.

When the program terminates, the names of the girls you input will be seen at the left in a random order. The names of the boys to which they are matched will be seen to the right, and will also be in a random order. (Warning: This game should not be used for the selection of spouses.)

Program 46: Computerized Bingo Caller

This program also serves as a game aid, as opposed to a complete game. It is probably one of the most practical and useful programs in this chapter when used by fire departments, civic clubs, and other organizations which have regular bingo games. The computer, using this program, fully takes the place of those mechanical bingo machines with the floating ping pong balls bearing designations of bingo card positions. When this program is run, the computer will randomly select any of 75 possible bingo calls and display it on the screen. All numbers are arrived at randomly, so the game will be absolutely fair.

The RANDOMIZE statement is seeded with numeric variable NUM, which is input via the keyboard. Two arrays are established in lines 100 and 110, each containing 75 elements. Lines 120 through 150 assign the first 75 positions in each array to the number that corresponds to that position. In other words, the first 75 elements of each array are assigned the numbers 1 through 75 sequentially. Line 160 assigns Y a random number between 1 and 75. Line 170 is identical to the operation of lines 240 and 310 in the previous program. It tests for a reassigned value of 0 in array A. This would be an indication that the value contained in this array position had already been called.

Now, while bingo numbers range from 1 to 75, they are also coupled with the letters B, I, N, G, or O. Lines 180 through 220 make the letter assignments based upon the value of the random number. The letter assignment will be found in Q$. For instance, in line 180 a test is made to see if the number at A(Y) is less than 16. Remember, the array has been assigned numbers sequentially from 1 to 75, so A(15) will have a value of 15. In Bingo, numbers from 1 to 15 are preceded by a B. In line 180, if A(Y) is less than 16, then Q$ is assigned a string value of B. Lines 190 through 210 test for a numeric value to assign the letters I, N, or G. In line 220, Q$ is assigned the value of the letter O. If none of the other test conditions find values to which letter can be assigned; the number returned by the random number generator must then be 61 to 75. If one of the previous test lines finds a match, line 220 is branched over.

When line 230 is executed, the letter has

```
10 REM COMPUTERIZED BINGO CALLER
20 REM COPYRIGHT FREDERICK HOLTZ
30 CLS
40 SCREEN 0
50 WIDTH 40
60 KEY OFF
70 INPUT"TYPE IN ANY NUMBER FROM 1 TO 30
000";NUM
80 CLS
90 RANDOMIZE NUM
100 DIM A(75)
110 DIM B(75)
120 FOR X=1 TO 75
```

```
130 A(X)=X
140 B(X)=X
150 NEXT X
160 Y=INT(RND*75)+1
170 IF A(Y)=0 THEN 160
180 IF A(Y)<16 THEN Q$="B":GOTO 230
190 IF A(Y)>=16 AND A(Y)<31 THEN Q$="I":
GOTO 230
200 IF A(Y)>=32 AND A(Y)<46 THEN Q$="N":
GOTO 230
210 IF A(Y)>=46 AND A(Y)<61 THEN Q$="G":
GOTO 230
220 Q$="O"
230 LOCATE 1,1
240 PRINT"PRESS ENTER FOR BINGO CALL"
250 INPUT"ENTER 'END' WHEN GAME IS OVER"
;E$
260 IF E$="END" THEN 340
270 LOCATE 12,20
280 BEEP
290 PRINT Q$;A(Y)
300 A(Y)=0
310 COUNT=COUNT+1
320 IF COUNT=75 THEN END
330 GOTO 160
340 CLS
350 T=1
360 FOR X=1 TO 75
370 IF B(X)<16 THEN Q$="B":GOTO 420
380 IF B(X)>=16 AND B(X)<31 THEN Q$="I":
GOTO 420
390 IF B(X)>=32 AND B(X)<46 THEN Q$="N":
GOTO 420
400 IF B(X)>=46 AND B(X)<61 THEN Q$="G":
GOTO 420
410 Q$="O"
420 IF A(X)<>0 THEN 470
430 S=S+1
440 IF S=20 THEN S=1:T=T+9
450 LOCATE S,T
460 PRINT Q$;B(X)
470 NEXT X
480 LOCATE 23,1
490 END
```

Program 46. Computerized Bingo Caller.

already been assigned to Q$. You are prompted to press Enter for a bingo call, or to type "END" if the game is over. Line 260 tests for an input of END and then branches to the closing routine. If you simply press Enter, line 270 positions the text cursor near the center of the screen, a beep is heard, and line 290 prints Q$ (the letter) followed by the value in A(Y). Line 300 reassigns the value of A(Y) to zero, since this number has already been read. Notice that the zero is not enclosed in quotation marks here, because we are dealing with a numeric array instead of the string array in the previous program. Lines 310 increments numeric variable COUNT by one. Line 320 tests for a condition of this variable being equal to 75. This would mark the end of the game, since all numbers have been called, and the program terminates. Since this will theoretically never happen, line 330 branches to line 160, where a new random number is assigned to Y and you are again prompted to press Enter for the bingo call.

Now, when a player scores bingo (using the standard bingo cards which you must supply), simply type END and there is a branch to line 340. Here the screen is cleared, T is assigned to 1, and a FOR-NEXT loop begins in line 360. This counts from 1 to 75, which happens to be the number of different calls in bingo. Lines 370 through 410 perform the same text function as lines 180 through 220. However, these lines use the values contained in the second array named B, which we haven't really used yet. Line 420 is the key here; it tests to see if the value at position A(X) is 0. If it isn't, line 430 increments S by 1. Line 450 locates screen position S,T and prints the value of Q$ (the letter), followed by B(X), the number. What does all this do? It simply prints out on the screen all the calls that have been made from the A array, by referencing them to the B array. This allows you to check the legitimacy of the bingo. Let me digress a bit.

Earlier in the program, any time a call was made the position in array A containing that call was set to zero. (Originally, arrays A and B were assigned the same values.) Array B is there just to serve as a reference; array B has not been changed by any previous program lines, but the values in array A have; the values that have been used are changed to zero. The display routine at the end of the program first checks a position in array A. If it's equal to zero this call has been used. The same position in array B is used to retrieve the number in that position, which corresponds to the original value in array before being reassigned to zero. Any value in array A that is not equal to zero represents a value not used before bingo was reached. Line 420 will then branch to line 470, causing the loop to cycle again without printing anything. Variables S and T are used with the LOCATE statement to neatly display all of the called values.

Program 47: Spin the Bottle

Spin the Bottle is a game that incorporates text characters and animated graphics programming. It will allow you to input the names of two girls and two boys, and then spins a graphic bottle to see who kisses whom. Figure 6-2 shows the screen display. Notice that, in place of the standard bottle, a two-position pointer has been used to indicate the kissing couple. This is a nice game because you can always be certain that a girl will be matched with a boy. This is not true in the mechanical version of this game.

Line 70 prompts you to input a random number seed that is used with the RANDOMIZE statement in line 90. An array is established in line 100, which will be used to hold the screen image that will be used later as part of the animation routine. Lines 120 through 180 prompt you to input the names of the players. Notice that the girls' names are input first, followed by the names of the boys. This pattern must be followed to achieve an appropriate run. Lines 200 through 270 print these names in a box-like pattern on the screen. The names will surround the arrow pointers.

Lines 280 through 340 contain the animation routine. The use of PUT and GET in this program is not standard as far as animation goes. Line 110 uses GET to assign to array A the screen content of a box formed at coordinate 0,0 and 60,60. However, when this happens the screen has already been cleared, and this box will contain nothing but the screen background color. Why GET an image

```
10 REM SPIN THE BOTTLE
20 REM COPYRIGHT FREDERICK HOLTZ
30 SCREEN 1
40 KEY OFF
50 COLOR 1,0
60 CLS
70 INPUT"TYPE IN ANY NUMBER FROM 1 TO 20
000";NUM
80 CLS
90 RANDOMIZE NUM
100 DIM A(500)
110 GET(0,0)-(60,60),A
120 INPUT"NAME OF FIRST GIRL";A$
130 CLS
140 INPUT"NAME OF SECOND GIRL";B$
150 CLS
160 INPUT"NAME OF FIRST BOY";C$
170 CLS
180 INPUT"NAME OF SECOND BOY";D$
190 CLS
200 LOCATE 13,8
210 PRINT A$
220 LOCATE 13,30
230 PRINT B$
240 LOCATE 7,18
250 PRINT C$
260 LOCATE 18,18
270 PRINT D$
280 FOR X=1 TO 101
290 SOUND 1200,.1
300 R=INT(RND*4)
310 DRAW"A=R;BM160,100U20G5BE5F5BM160,10
0R20H5BF5G5"
320 IF X=100 THEN 350
330 PUT(130,60),A,AND
340 NEXT
350 LOCATE 23,1
360 PRINT"PRESS ANY KEY TO SPIN AGAIN"
370 T$=INKEY$
380 IF T$="" THEN 370
390 LOCATE 23,1
400 PRINT"                              "
410 GOTO 280
```

Program 47. Spin the Bottle.

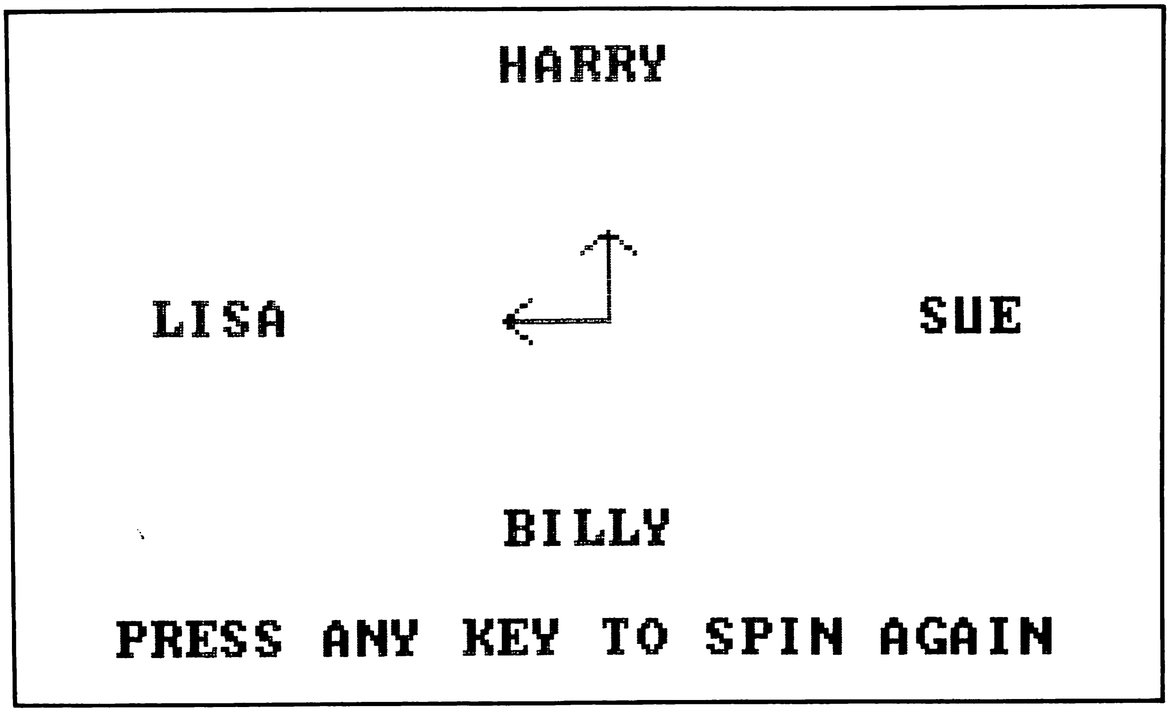


Fig. 6-2. Screen display from Spin the Bottle.

that really isn't there? There is really something there—screen background—and here's how it's used.

Lines 310 draws the arrow figure. This image will consist of one arrow pointing toward the top of the screen and another pointing toward the right. However, notice that the angle command is used in the DRAW statement, assigning it to the angle specified by variable R. In line 300, R is assigned a random number from 0 to 3. You will recall from an earlier discussion that 0 represents no shift of the image produced by the DRAW statement. The numbers 1 through 3 bring about left shifts of 90, 180, and 270 degrees respectively. This is what causes the pointers to rotate. However, line 330 contains a PUT statement, which PUTs the screen background color to the image produced by the DRAW statement, effectively erasing it from the screen. If this were not done, each time the DRAW statement rotated the figure the previous figure would remain on the screen; after a few cycles of the loop, the two pointers would quickly look like several. Every name would have an arrow pointing to it. The way this program is written, the sequence is:

1) Draw the pointers.
2) Erase the pointers.
3) Rotate the pointers.
4) Erase the pointers.
5) Rotate the pointers.
6) Etc.

This action occurs rapidly, but not so fast that you can't see the pointers displayed before they are erased. The result is the appearance of a spinning set of pointers. The pointers spin 100 times. The SOUND statement in line 290 produces some interesting sound effects that sound similar to a glass bottle being spun on the sidewalk.

The loop cycles 100 times, even though it is set up to cycle 101 times by the FOR statement in line 280. However, line 320 exits the loop when X is equal to 100, before the PUT statement in line 330 is executed. Actually, line 280 could assign a top value of 100, since 101 will never be reached because of the exit. The count to 101 and the exit at 100 in line 320 make the exit a little easier to understand. When the loop has been exited, the last position of the pointers remain on the screen. The players are then prompted to press Enter to spin again.

This is a very interesting game from a visual standpoint. The COLOR statement in line 50 sets the background to a dark blue. The spinning pointers, along with the sound effects, are bound to hold attention. Try it. You'll like it.

Program 48: Scrambled Word Game

Here is a game you can modify endlessly by adding different DATA statement lines. The computer will display a word on the screen. It is a legitimate word, but the letters have been mixed up. It is your job to unscramble the letters and type in the correct word. If you guess the word correctly, the computer will prompt you to this fact; if you make an incorrect guess, the computer will tell you about this as well, and will also print the correct answer. At the end of the game, your score is printed. The score gives you the number of wrong answers and the number of guesses.

Line 50 sets up a string array (A$) which holds a maximum of 10 elements. Lines 80 and 90 use a version of a program discussed earlier in this chapter that utilizes the TIME$ function to arrive at a random seed value. Here, E is the random seed number, used with RANDOMIZE in line 100, that is derived from the seconds portion of TIME$. Line 110 contains the READ statement, which retrieves the value from the DATA statement lines beginning at line 490 in the program. Line 120 contains an IF-THEN statement to test for the end of the program. (More on this later.) Line 130 is a count routine using NUM as its variable.

To explain the workings of this program, one must remember that the DATA statements in lines 490 through 510 contain the words to be scrambled. When the program is first run, the READ statement in line 110 will access the first DATA statement element, which is the word "DISKETTE." Line 140 uses the LEN statement to set the maximum value of X in the loop begun in this line. LEN(Q$) is equal to the number of characters in Q$, which is eight. Line 140 is really saying:

```
140  FOR X=1 TO 8
```

However, the top value for X will change with the number of letters in each word accessed.

Line 150 assigns the first eight elements in array A$ to the letters that make up the word DISKETTE. A$(1) equals D, A$(2) equals I, A$(3) equals S, etc. The assignments are made to the array using the MID$ function to read the letters in Q$ one at a time.

Line 170 begins another loop that counts from 1 to LEN(A$), or 8 in this example. Line 180 assigns a random number to Z, in this case a number from 1 to 8, or in other examples from 1 to the length of the mystery word. Line 190 is a test line to see if the value of A$(Z) has been reassigned as "0." The LOCATE statement in line 200 steps the text cursor across the screen near center as each letter is printed. The actual print is made in line 210. Line 220 assigns that position in the array to "0" and the loop goes around for another letter.

Assuming the mystery word is DISKETTE, if the first number assigned to Z is 3, then the letter S is printed to the screen, since this is the letter found at position A$(3). If the next value of Z is 8, the next letter printed will be E, which is found in the eighth element of A$. When the loop times out line 250 prompts you for an answer, which is assigned to ANS$. Line 260 checks to see if your answer is equal to the DATA statement word. If so, there is a branch to line 330, which clears the screen, produces a beep, and tells you that the answer is correct. You are then prompted to press Enter for another word, whereupon the next DATA statement word is accessed via a branch to line 110. The sequence begins again. If, however, your answer is not equal to Q$, line 270 increments C by one. Line 280 clears the screen, and a low-frequency rasp is heard because of the SOUND statement in line 290. You are then told the correct answer. In

```
10 REM SCRAMBLED WORD GAME
20 REM COPYRIGHT FREDERICK HOLTZ
30 SCREEN 0
40 WIDTH 40
50 DIM A$(10)
60 CLS
70 KEY OFF
80 E$=MID$(TIME$,8,2)
90 E=VAL(E$)
100 RANDOMIZE E
110 READ Q$
120 IF Q$="END" THEN 410
130 NUM=NUM+1
140 FOR X=1 TO LEN(Q$)
150 A$(X)=MID$(Q$,X,1)
160 NEXT X
170 FOR X=1 TO LEN(Q$)
180 Z=INT(RND*LEN(Q$))+1
190 IF A$(Z)="0" THEN 180
200 LOCATE 14,16+X
210 PRINT A$(Z)
220 A$(Z)="0"
230 NEXT
240 LOCATE 23,1
250 INPUT"WHAT IS THE WORD";ANS$
260 IF ANS$=Q$ THEN 330
270 C=C+1
280 CLS
290 SOUND 200,5
300 LOCATE 14,5
310 PRINT"THE CORRECT ANSWER IS:";Q$
320 GOTO 370
330 CLS
340 BEEP
350 LOCATE 14,15
360 PRINT"CORRECT ANSWER!!!"
370 LOCATE 23,1
380 INPUT"PRESS <ENTER> FOR A NEW WORD."
;RT$
390 CLS
400 GOTO 110
410 CLS
420 PRINT"THE GAME IS OVER. YOU HAD"C"WR
ONG"
430 PRINT"ANSWERS OUT OF"NUM"GUESSES."
```

```
440 LOCATE 23,1
450 END
460 REM USER MAY PUT ANY WORDS IMAGINABL
E IN THE FOLLOWING DATA STATEMENT LINES.

470 REM REMEMBER TO TERMINATE YOU DATA S
TATEMENT ELEMENTS WITH 'END'.
480 REM KEEP ALL WORDS TO 10 OR LESS LET
TERS
490 DATA DISKETTE,MONITOR,MEMORY,SOFTWAR
E,GRAPHICS,PROGRAM
500 DATA TERMINAL,PROCESSOR,DATA,FLOPPY,
INPUT,CIRCUIT
510 DATA PRINTER,ASCII,END
```

Program 48. Scrambled Word Game.

line 320, there is a branch to line 370, where you are again prompted to press Enter for a new word.

Notice in line 510 that the last DATA element word is END. Line 120 detects this word and branches to line 410, where the screen is cleared. You are then prompted that the game is over; the number of wrong answers and the total number of guesses is displayed before the program is terminated.

To insert your own words into this program to be scrambled and displayed, simply add more DATA statements after line 510 (making sure you remove the END from line 510), or change the words contained in lines 490 through 510 to different words. Maximum word length is 10 characters, although you can use more if you change the size of your array in line 50. Always terminate your last DATA statement line with the word END. This is a game that will never be obsolete, simply because it can be easily updated to include any words you want.

Program 49: Word Guess

Word Guess is another vocabulary game. This one asks you to guess a word based upon another word that looks or sounds like the correct answer, and with the aid of a definition of the correct answer. For instance, one clue might be:

CURABLE = LASTING

The correct answer here is DURABLE, which is a word that sounds like CURABLE and means LASTING. Line 80 through 100 print the welcome to the program and ask if you need instructions. Lines 110 through 130 check for your input. If instructions are needed, there is a branch to line 950, which displays the instructions on the screen. If not, there is a branch to line 140, where the screen is cleared. Line 150 begins a FOR-NEXT loop designed to read the 53 DATA elements which begin at line 430. Line 160 is the exit routine. Line 170 simply assigns variable C the value of zero. C is a count routine that will keep track of the number of guesses made. You are allowed three guesses to come up with the right answer for each clue. If you miss the third time, the correct answer is printed, and the computer chalks up a miss in another count routine, found in line 240. All wrong answers are represented by W. Line 180 reads two values from each DATA statement line. In line 430, the first DATA statement element is

CURABLE = LASTING

This is assigned to string variable L$. The second DATA element in line 430 is DURABLE. This is

```
10 REM WORDGUESS
20 REM COPYRIGHT FREDERICK HOLTZ
30 CLS
40 KEY OFF
50 SCREEN 0
60 WIDTH 40
70 LOCATE 14,11
80 PRINT"WELCOME TO WORDGUESS"
90 LOCATE 23,1
100 INPUT"DO YOU NEED INSTRUCTIONS (Y/N)
";F$
110 IF F$="Y" THEN 950
120 IF F$="N" THEN 140
130 IF F$<>"Y" AND F$<>"N" THEN 100
140 CLS
150 FOR J=1 TO 53
160 IF J=53 THEN CLS:LOCATE 14,5:GOTO 39
0
170 C=0
180 READ L$,M$
190 LOCATE 14,15:PRINT L$
200 INPUT A$
210 IF A$=M$ THEN 370
220 C=C+1
230 IF C<=3 THEN SOUND 100,15:LOCATE 23,
1:PRINT"SORRY, TRY AGAIN":FOR TD=1 TO 15
00:NEXT TD:CLS
240 W=W+1
250 IF C>3 THEN 260 ELSE 190
260 SOUND 100,10
270 FOR TD=1 TO 500
280 NEXT TD
290 SOUND 100,10
300 CLS
310 LOCATE 23,1
320 PRINT"TOO MANY GUESSES. THE WORD WAS
:"M$
330 PRINT"TRY THE NEXT WORD"
340 FOR TD=1 TO 2500:NEXT TD
350 CLS
360 GOTO 380
370 BEEP:LOCATE 23,1:PRINT "CORRECT! TRY
 THE NEXT ONE":FOR TD=1 TO 1500:NEXT TD:
CLS
```

Program 49. Word Guess. (Continued to page 142.)

```
380 NEXT J
390 PRINT"NUMBER OF WRONG ANSWERS IS"W:L
OCATE 15,8
400 AV=INT(((208-W)/208)*100)
410 PRINT"YOUR SCORE IS"AV"%"
420 END
430 DATA  CURABLE=LASTING,DURABLE
440 DATA  WILLOW=A LARGE WAVE,BILLOW
450 DATA SACK=TO CHOP WITH HEAVEY BLOWS,
HACK
460 DATA SLEET=A UNIT OF NAVAL SHIPS,FLE
ET
470 DATA SLICK=TO MAKE A SNAPPING SOUND,
CLICK
480 DATA SLASH=TO COLLIDE NOSILY,CLASH
490 DATA GOWN=THE SOFT FEATHERS OF A GOO
SE,DOWN
500 DATA BOWL=A HOOD OR HOUSING,COWL
510 DATA RABBLE=TO TALK AIMLESSLY,BABBLE

520 DATA BUBBLE=BROKEN PIECES OF STONE,R
UBBLE
530 DATA SILK=TO SWINDLE OR CHEAT,BILK
540 DATA COLDER=TO FOR A JOINT WITH MOLT
EN METAL,SOLDER
550 DATA DOCK=A GATED PORTION OF A CANAL
 OR RIVER,LOCK
560 DATA BUMP=A SMALL MASS OR WAD,LUMP
570 DATA SITE= A SMALL ARACHNID,MITE
580 DATA SIGH=NEAR IN TIME,NIGH
590 DATA ROMP=SPLENDOR,POMP
600 DATA GROW=THE RIDGE OVER THE EYES,BR
OW
610 DATA JOIN=TO INVENT,COIN
620 DATA PIVOT=A PIECE OF SOD,DIVOT
630 DATA BLINK=RINGING SOUND,CLINK
640 DATA CLIP=PERT,FLIP
650 DATA SOUNDER=CAD,BOUNDER
660 DATA DECISION=AN INVALIDATION,RECISI
ON
670 DATA SORRY=A GARMENT OF INDIA,SARI
680 DATA TURN=AN AQUATIC BIRD,TERN
690 DATA TARPON=A TURTLE,TERRAPIN
700 DATA IDOL=A POEM,IDYL
710 DATA CHECKS=TO PUT A CURSE ON,HEX
```

```
720 DATA DIXIE=AN ELF OR SPRITE,PIXIE
730 DATA LAGGARD=WILD LOOKING,HAGGARD
740 DATA FIRST=FORMERLY,ERST
750 DATA SYNICAL=CONCERNING HOSPITALS,CL
INICAL
760 DATA DINNER=AN ORIENTAL COIN,DINAR
770 DATA SCENT=FORCE OR POWER,DINT
780 DATA PHYSICAL=PERTAINING TO PUBLIC R
EVENUES,FISCAL
790 DATA FOOL=DIRVEL,DROOL
800 DATA BIRD=A MAIDEN LADY,BURD
810 DATA OVERLAP=A COARSE FABRIC,BURLAP
820 DATA SENDER=A PIECE OF BURNED COAL,C
INDER
830 DATA SHRAPNEL=A HOOK,GRAPNEL
840 DATA INCITE=TO HEAT INTENSELY,IGNITE

850 DATA LOOP=A CLOTH MASK,LOUP
860 DATA SLUM=A DELICIOUS FRUIT,PLUM
870 DATA SNIFFLE=NONSENSE,PIFFLE
880 DATA PALE=MOLLUSK,SNAIL
890 DATA NOGOODNIK=AN ARTIFICIAL SATELLI
TE,SPUTNIK
900 DATA LUNAR=ROMANTIC MALE SINGER,CROO
NER
910 DATA CACKLE=MANGLE,HACKLE
920 DATA FURROW=A HOLE OR PASSAGE,BURROW

930 DATA CURFEW=A SHORE BIRD,CURLEW
940 DATA CURL=TO ROLL A SAIL,FURL
950 CLS
960 PRINT"WORDGUESS IS A VOCABULARY GAME
 WHICH"
970 PRINT"TESTS YOUR KNOWLEDGE OF WORDS.
 TWO "
980 PRINT"WORDS WILL BE SEEN ON THE SCRE
EN. THE"
990 PRINT"FIRST IS YOUR CLUE WORD. THE S
ECOND IS"
1000 PRINT"DEFINES THE MEANING OF THE WO
RD THE"
1010 PRINT"COMPUTER IS LOOKING FOR. THE
FIRST WORD"
1020 PRINT"MAY RHYME WITH THE WORD WHICH
 CONSTI-
```

```
1030 PRINT"TUTES A CORRECT ANSWER OR IT
MAY"
1040 PRINT"INCLUDE MANY OF THE LETTERS O
F THE"
1050 PRINT"CORRECT ANSWER. YOU WILL BEGI
VEN FOUR"
1060 PRINT"CHANCES TO COME UP WITH A COR
RECT"
1070 PRINT"ANSWER. IF YOU DON'T SUCCEED,
THE ANSWER"
1080 PRINT"WILL BE DISPLAYED. THE COMPUT
ER WILL"
1090 PRINT"KEEP TRACK OF YOUR SCORE. GOO
D LUCK!"
1100 LOCATE 23,1
1110 INPUT"PRESS <ENTER> TO BEGIN";S$
1120 CLS
1130 GOTO 150
```

assigned to M$ in line 180. Line 190 locates a position near the center of the screen and prints L$. You are then allowed to input your guess in line 200. Your guess is assigned to A$. Line 210 compares the correct answer (M$) with your guess (A$). If the two are equal, there is a branch to line 370, which causes the computer to beep and print a prompt indicating that your answer is correct. This is a multiple statement line that actually contains six different statements. After the prompt, there is a time delay loop. When this times out the screen is cleared, and the loop is recycled in line 380. The next clue/answer is read from the next DATA statement line.

If your answer is incorrect (A$ not equal to M$), line 220 increments C by one. Line 230 checks for C being less than or equal to 3. When this is true, the computer tells you the answer is wrong and also tells you to try again. The loop is recycled in the multi-statement line 230. During any of these three guesses, if you get the right answer, line 210 detects this and again branches to line 370. When you've gone past your third guess and still haven't gotten a correct answer, the wrong answer count routine in line 240 is incremented by one, and a wrong answer is chalked up to your score. Line 320 prints the correct answer and you are then prompted to try the next word.

This program is very simple. It is long simply because of the number of DATA statement lines contained in the program. When you have finished guessing all 53 words, line 390 prints the number of wrong answers and then gives you a score based upon percentage of wrong answers to total answers. The formula for this is contained in line 400. Line 410 prints your score on the screen.

Program 50: Numbers Draw Poker

This is a draw poker program which uses numbers to represent cards. One must use imagination here to determine who the winner is, since the computer does not do this for you. In this game, the numbers 2 through 10 are used to represent card values of 2 through 10. The number 11 is used to represent an Ace or any face card. You play this just like you would standard poker, trying for full houses, straights, etc. Line 70 establishes an

```
10 REM NUMBERS DRAW POKER
20 REM COPYRIGHT FREDERICK HOLTZ
30 CLS
40 SCREEN 0
50 WIDTH 40
60 KEY OFF
70 RN=VAL(MID$(TIME$,7,2))
80 RANDOMIZE RN
90 DIM A(5),B(5)
100 INPUT"NAME OF FIRST PLAYER";A$
110 CLS
120 INPUT"NAME OF SECOND PLAYER";B$
130 IF PN=0 THEN PN=1:C$=A$:GOTO 150
140 IF PN=1 THEN PN=0:C$=B$:GOTO 150
150 CLS
160 PRINT"OK ";C$;" IT'S YOUR TURN."
170 INPUT"PRESS <ENTER> TO RECEIVE YOUR
HAND.";ER$
180 CLS
190 CLS
200 LOCATE 12,12
210 FOR X=1 TO 5
220 Z=INT(RND*10)+2
230 IF A(X)=0 THEN A(X)=Z
240 PRINT A(X);
250 NEXT X
260 C=C+1
270 IF C=2 THEN C=0:GOTO 460
280 LOCATE 20,1
290 PRINT"HOW MANY CARDS DO YOU WANT ";C
$
300 INPUT N
310 IF N>3 THEN 290
320 CLS
330 LOCATE 12,12
340 FOR X=1 TO 5
350 PRINT A(X);
360 NEXT X
370 FOR T=1 TO N
380 LOCATE 20,1
390 INPUT"WHICH CARD";B
400 LOCATE 20,1
410 PRINT"                    "
420 A(B)=0
430 NEXT T
```

```
440 CLS
450 GOTO 190
460 LOCATE 23,1
470 PRINT"YOUR TURN IS OVER ";C$
480 LST=LST+1
490 IF LST=2 THEN LST=0:GOTO 570
500 FOR X=1 TO 5
510 B(X)=A(X)
520 A(X)=0
530 NEXT X
540 FOR DLAY=1 TO 1500:NEXT DLAY
550 CLS
560 GOTO 130
570 FOR DLAY=1 TO 1500:NEXT DLAY
580 CLS
590 LOCATE 8,1
600 PRINT A$;"'S HAND =";
610 FOR X=1 TO 5
620 PRINT B(X);
630 NEXT X
640 LOCATE 17,1
650 PRINT B$;"'S HAND =";
660 FOR X=1 TO 5
670 PRINT A(X);
680 NEXT X
690 LOCATE 23,1
700 INPUT"PRESS <ENTER> TO PLAY AGAIN.";
ER$
710 CLS
720 FOR X=1 TO 5
730 A(X)=0
740 NEXT X
750 GOTO 130
```

Program 50. Numbers Draw Poker.

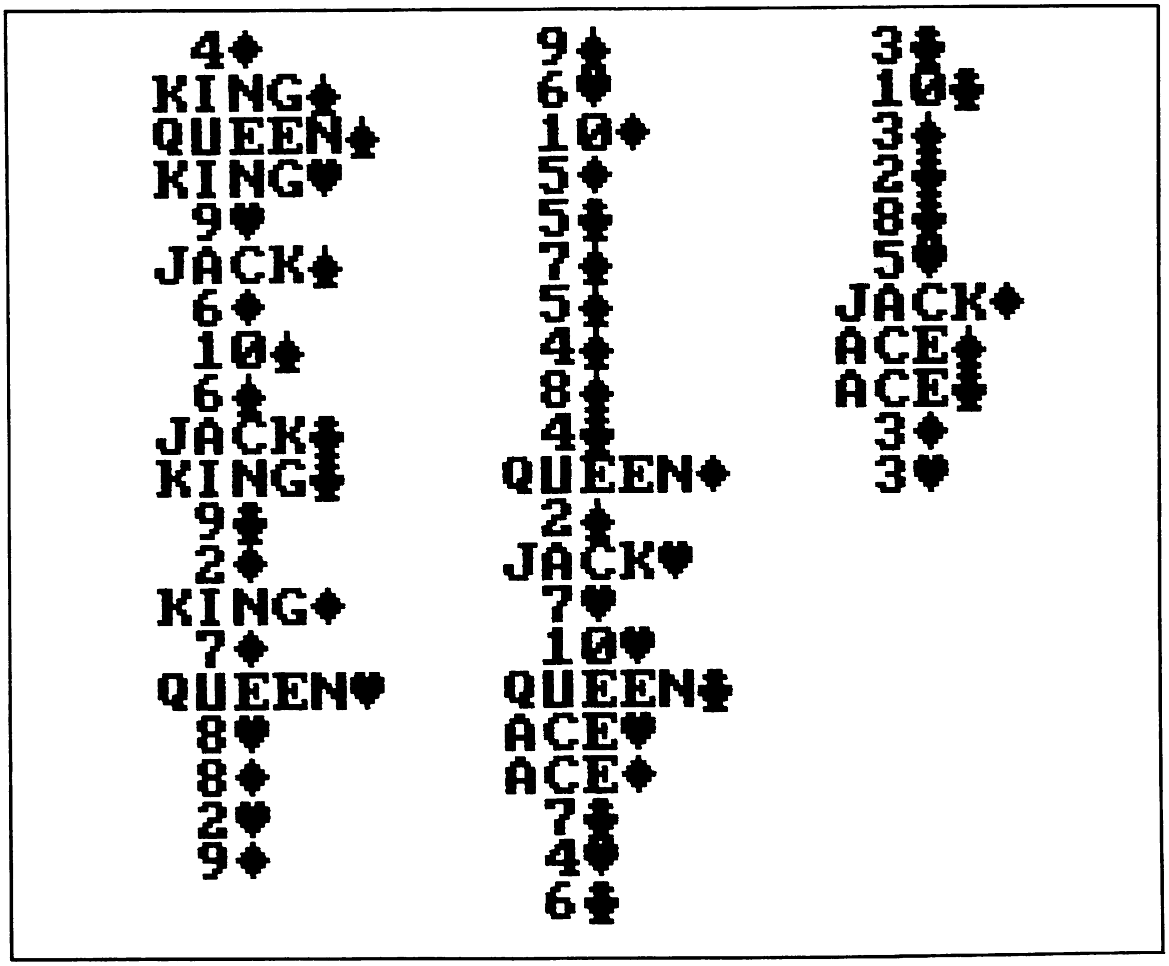
automatic random number seed by pulling it from the seconds portions of TIME$. The RANDOMIZE statement in line 80 uses RN as it random number seed. Line 90 sets up two arrays to contain five elements, the number of cards assigned to each player's hand in this version of draw poker. Line 100 then prompts you to input the name of the first player, which is assigned to A$. Line 120 allows for the input of the second player's name, which is assigned to B$. Lines 130 and 140 detect whose turn it is to play. The name of the current player is assigned to C$.

Line 160 prints the prompt for the player named in C$ to begin. When Enter is pressed,the screen is cleared, the text cursor assumes a position near the middle. A random number routine is set up in lines 210 through 250. In line 220, Z receives a value from 2 to 11. Array position A(X) is assigned the value of Z, and this value is printed to the screen. The player is then asked how many cards he/she wants. This number is assigned to N in line 300. You are allowed to draw a maximum of three cards, and line 310 makes sure that you don't ask for more than this. If you don't want any cards, simply press Enter, since an automatic value for N of 0 will then be assigned.

Again the screen is cleared, and the same position is accessed by the text cursor. The same card numbers as before are displayed on the screen and you are then prompted to pick the first card you want replaced with the draw. Here, you must type in a number from 1 to 5 indicating the position of the card to be replaced. When you press Enter, you are again prompted for another card to be replaced, assuming you wish to draw more than one card. When all of your cards have been replaced, the new arrangement is shown on the screen. This is your final hand. It is now time for the second player to receive a hand and make the same selection. In the end, both sets of cards will be displayed and the two players determine who has won.

Program 51: Card Shuffler

This is not a game in itself, nor is it really a tool with which to play a game. It is a card shuffler routine that can be used to build a true card game



```
    4♦             9♠            3♣
 KING♠             6♥           10♣
QUEEN♠            10♦            3♠
 KING♥             5♦            2♣
    9♥             5♣            8♣
 JACK♠             7♠            5♥
    6♦             5♠         JACK♦
   10♠             4♣          ACE♠
    6♠             8♠          ACE♣
 JACK♣             4♠            3♦
 KING♣         QUEEN♦            3♥
    9♣             2♠
    2♦          JACK♥
 KING♦             7♥
    7♦            10♥
QUEEN♥         QUEEN♣
    8♥           ACE♥
    8♦           ACE♦
    2♥             7♣
    9♦             4♥
                   6♣
```

Fig. 6-3. Printout of Card Shuffler program.

```
10 REM CARD SHUFFLER
20 REM COPYRIGHT FREDERICK HOLTZ
30 CLS
40 SCREEN 0
50 WIDTH 40
60 KEY OFF
70 RN=VAL(MID$(TIME$,7,2))
80 RANDOMIZE RN
90 C=1
100 S=3
110 G=1
120 AC$="ACE"
130 KI$="KING"
140 QU$="QUEEN"
150 JA$="JACK"
160 DIM A$(52),B$(52)
170 S$=CHR$(S)
180 FOR X=C TO C+8
190 G=G+1
200 A$(X)=STR$(G)+S$
210 NEXT X
220 A$(X)=AC$+S$
230 A$(X+1)=KI$+S$
240 A$(X+2)=QU$+S$
250 A$(X+3)=JA$+S$
260 C=X+4
270 S=S+1
280 G=1
290 IF S=7 THEN 310
300 GOTO 170
310 Z=INT(RND*52)+1
320 IF A$(Z)="0" THEN 310
330 CT=CT+1
340 B$(CT)=A$(Z)
350 A$(Z)="0"
360 IF CT=52 THEN 380
370 GOTO 310
380 REM THE FOLLOWING PORTION
390 REM OF THE PROGRAM SIMPLY DISPLAYS
400 REM THE SHUFFLED DECK
410 REM THIS PORTION WILL BE REPLACED
420 REM BY YOUR CARD GAME PROGRAM
430 FOR X=1 TO 20
440 LOCATE X,1
450 PRINT B$(X)
```

```
460 NEXT
470 FOR X=21 TO 41
480 LOCATE X-20,10
490 PRINT B$(X)
500 NEXT
510 FOR X=42 TO 52
520 LOCATE X-41,30
530 PRINT B$(X)
540 NEXT
550 LOCATE 23,1
560 END
```

Program 51. Card Shuffler.

played like any standard card game.

With this program, you can randomly shuffle a deck of 52 cards. In this example, the shuffled deck will be displayed card by card on the screen, as shown in Fig. 6-3. As you can see, the card values are given by name and suit. Fortunately, the character set contains the suit markings for hearts, clubs, diamonds, and spades.

An automatic random number seed routine is set up in line 70. Lines 120 through 150 assign face card names to string variables used to represent them. Line 160 dimensions two string arrays to hold 52 elements each. S$ in line 170 will hold the suit value, which is an ASCII number representing that character in the character set. Line 180 begins a FOR-NEXT loop that steps from a value of C (initially, 1) to a maximum value of C + 8. This is a total of 9 cards, which represent the 9 number cards in any suit. Line 190 is a simple count routine that steps variable G by one during each pass of the loop. Line 200 assigns the elements to A$ using the STR$ function, which converts G (a number) to a string value. Coupled with this number is S$, which is the suit symbol. When the loop times out, the value of X will be one more than C + 8. Therefore, lines 220 through 250 add the face card in that suit to A$. In line 260 C is reassigned a value of X + 4, which represents the next unfilled element in the array. The value of S is stepped by one, which represents the next suit character. You will recall that originally S was equal to 3, because it was assigned that value in line 100. ASCII character 3 is the hearts symbol. In line 270 S is incremented by one, and is therefore equal to 4. ASCII character 4 is the diamonds symbol. Line 280 resets G to a value of one (its original value in line 110). Line 290 checks to see if S is equal to 7, which indicates that the full deck has been loaded into A$. If this is not true there is a branch back to line 170, where all of the same cards are added to the array, but this time with the new suit. When all of the cards in each suit have been added, line 270 branches to line 310. Here, a random number from 1 to 52 is assigned to Z. Line 320 tests for a reassigned value of "0" to A$(Z). Line 330 is a count routine that increments CT by one each time this line is executed. Now, line 340 assigns to the first position of B$ the value of the card found at A$(Z). Line 350 then reassigns A$(Z) to a value of "0," so the card found here can never be accessed again during this run. Line 360 checks to see if all 52 cards have been loaded into array B$. If not, line 370 branches to line 310 for another card. When all 52 cards have been loaded, there is a branch to line 380, which would normally be a card program you have written yourself. In this example, lines 430 through 540 print the contents of the shuffled deck on the screen.

To simplify this, remember that A$ is assigned the value of each card in the deck. This is done sequentially and by suit. Array B$ is assigned the contents of the first array on a random basis. This is where the actual shuffling takes place.

Program 52: Draw Poker

This program is simply the previous two programs combined into one long one and you will notice many similarities to them. This game works just like the Numbers Draw Poker game, except that, instead of displaying numbers that represent

```
10 REM DRAW POKER CARD GAME
20 REM COPYRIGHT FREDERICK HOLTZ
30 CLS
40 SCREEN 0
50 WIDTH 40
60 KEY OFF
70 INPUT"NAME OF FIRST PLAYER";AA$
80 CLS
90 INPUT"NAME OF SECOND PLAYER";BB$
100 CLS
110 DIM A$(52),B$(52),X$(5),Y$(5)
120 RN=VAL(MID$(TIME$,7,2))
130 RANDOMIZE RN
140 LOCATE 14,13
150 PRINT"SHUFFLING DECK"
160 C=1
170 S=3
180 G=1
190 AC$="ACE"
200 KI$="KING"
210 QU$="QUEEN"
220 JA$="JACK"
230 S$=CHR$(S)
240 FOR X=C TO C+8
250 G=G+1
260 A$(X)=STR$(G)+S$
270 NEXT X
280 A$(X)=AC$+S$
290 A$(X+1)=KI$+S$
300 A$(X+2)=QU$+S$
310 A$(X+3)=JA$+S$
320 C=X+4
330 S=S+1
340 G=1
350 IF S=7 THEN 370
360 GOTO 230
370 Z=INT(RND*52)+1
380 IF A$(Z)="0" THEN 370
390 CT=CT+1
400 B$(CT)=A$(Z)
```

Program 52. Draw Poker. (Continued to page 150.)

```
410 A$(Z)="O"
420 IF CT=52 THEN CT=0:GOTO 440
430 GOTO 370
440 IF PN=0 THEN PN=1:C$=AA$:GOTO 460
450 IF PN=1 THEN PN=0:C$=BB$:GOTO 460
460 CLS
470 PRINT"OK ";C$;" IT'S YOUR TURN."
480 INPUT"PRESS <ENTER> TO RECEIVE YOUR
HAND.";ER$
490 CLS
500 CLS
510 LOCATE 12,12
520 FOR X=1 TO 5
530 Z=INT(RND*51)+2
540 IF X$(X)="" THEN X$(X)=B$(Z)
550 PRINT X$(X);" ";
560 NEXT X
570 CC=CC+1
580 IF CC=2 THEN CC=0:GOTO 770
590 LOCATE 20,1
600 PRINT"HOW MANY CARDS DO YOU WANT ";C
$
610 INPUT N
620 IF N>3 THEN 600
630 CLS
640 LOCATE 12,12
650 FOR X=1 TO 5
660 PRINT X$(X);" ";
670 NEXT X
680 FOR T=1 TO N
690 LOCATE 20,1
700 INPUT"WHICH CARD";B
710 LOCATE 20,1
720 PRINT"                    "
730 X$(B)=""
740 NEXT T
750 CLS
760 GOTO 500
770 LOCATE 23,1
780 PRINT"YOUR TURN IS OVER ";C$
790 LST=LST+1
800 IF LST=2 THEN LST=0:GOTO 880
810 FOR X=1 TO 5
820 Y$(X)=X$(X)
830 X$(X)=""
840 NEXT X
```

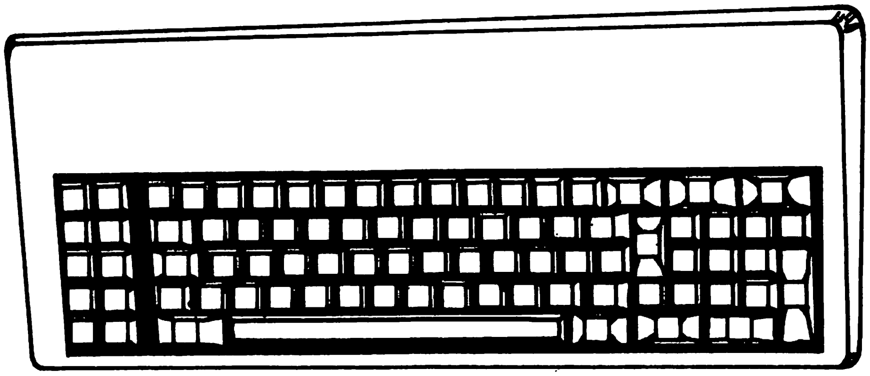
```
850 FOR DLAY=1 TO 1500:NEXT DLAY
860 CLS
870 GOTO 440
880 FOR DLAY=1 TO 1500:NEXT DLAY
890 CLS
900 LOCATE 8,1
910 PRINT AA$;"'S HAND =";
920 FOR X=1 TO 5
930 PRINT Y$(X);" ";
940 NEXT X
950 LOCATE 17,1
960 PRINT BB$;"'S HAND =";
970 FOR X=1 TO 5
980 PRINT X$(X);" ";
990 NEXT X
1000 LOCATE 23,1
1010 INPUT"PRESS <ENTER> TO PLAY AGAIN."
;ER$
1020 CLS
1030 FOR X=1 TO 5
1040 X$(X)=""
1050 NEXT X
1060 FOR X=1 TO 52
1070 A$(X)="0"
1080 B$(X)=""
1090 NEXT X
1100 GOTO 120
```

cards, it displays the card values themselves, which have been taken from B$. I will provide no further explanation of this game, since the explanations provided with the previous two programs do the job.

Summary

The AT&T PC-6300 is quite capable of simulating many common card and parlor games in a very realistic manner. The most interesting games are often the ones which combine graphics and text-mode programming, and even a sound effect or two. To write your own game programs, first pick out a game that you know well, then concentrate on the ways each aspect of the game can be committed to a computer program section. By handling the program assignment on a step-by-step basis, each problem is worked out individually. When the task is complete, all that's necessary to arrive at a working game is to tie all the simple task lines into one overall program. An excellent example of this was illustrated by the Numbers Draw Poker/Card Shuffler combination, which yielded a realistic Draw Poker Game.

Chapter 7



AT&T PC Filekeeping

This chapter will discuss only sequential files, which are accessed starting with the first element and ending at the last.

Program 53: File Reading Program

For this discussion, let's assume that a data file has been written to disk. We will learn later how to set up such a file and to write items to it, but for now, we assume it's already there. In order to access a file, you must first open it. In BASIC this is done using the OPEN statement, the use of which is shown in line 70. Here, the OPEN statement is followed by the name of a file. This name is enclosed in quotation marks, and in this example is called "FILE.FIL." The .FIL designation is not necessary, but is often used to distinguish that file from other files that contain programs written in BASIC. With the OPEN statement, a file may be opened for reading, writing, or adding. INPUT, used with OPEN, opens a file for reading. We must also specify a file number, which in this case is represented by "#1." If you open several different files at the same time, you will use other numbers. Line 70 in this program tells the computer to open "FILE.FIL" as file #1, and that file is to be opened for reading.

Line 80 uses the INPUT# statement to read data from the file. This statement is different from INPUT. The number sign identifies it as the statement to access file information. This statement is followed by a comma and A$. A$ will be used to hold the first data item from the file. Line 90 then prints the value of A$ to the screen; line 100 uses the CLOSE statement to close the file previously opened. This must be done before moving on to other program lines that will be included in a standard filekeeping routine. Therefore, the sequence is:

1) Open the file for input.
2) Get an item from the file using INPUT#.
3) Print the item to the screen.
4) Close the file.

If you assume that FILE.FIL exists and contains

```
10 REM FILE READING PROGRAM
20 REM COPYRIGHT
FREDERICK HOLTZ
30 SCREEN 0
40 WIDTH 40
50 KEY OFF
60 CLS
70 OPEN "FILE.FIL" FOR INPUT
AS #1
80 INPUT#1,A$
90 PRINT A$
100 CLOSE #1
```

Program 53. File Reading Program.

one item—the word let's say—"COMPUTER," when this program is run FILE.FIL file will be opened for input (reading), and the word "COMPUTER" will be assigned to A$ in line 80. Line 90 will then print A$ to the screen, and line 100 will close the file.

Program 54: Advanced File Reading Program

This is the same program discussed previously, but extra lines have been added. Line 110 brings about a branch to line 80 after the value of A$ has been printed to the screen. Line 80 contains the EOF function, which stands for End Of File. The number 1 is used in parentheses to indicate that EOF is to test for an end-of-file condition in the file opened as #1. The EOF function causes a branch to line 120 when there are no more items to be read in the file, i.e., the end-of-file position has been reached. Since line 110 continues branching back to a portion of the program prior to the use of the INPUT# statement, the file is continuously read in sequential order. This continues until line 80 detects the end of the file. The subsequent branch then closes the file. When run, this program will read every item contained in FILE.FIL and print it to the screen. When the last item has been read, the program will terminate. Without the EOF function in line 80, an error message would occur when there were no more items to read.

```
10 REM FILE READING PROGRAM
20 REM COPYRIGHT
FREDERICK HOLTZ
30 SCREEN 0
40 WIDTH 40
50 KEY OFF
60 CLS
70 OPEN "FILE.FIL" FOR INPUT
AS #1
80 IF EOF(1) THEN 120
90 INPUT#1,A$
100 PRINT A$
110 GOTO 80
120 CLOSE #1
```

Program 54. Advanced File Reading Program.

Program 55: File Writing Program

This simple program will allow you to open a file and write an item to it. Line 60 assigns A$ the value of "FILE ITEM." For now, this is just a simple phrase to demonstrate file writing, but later you will be able to assign any value you want to A$. Line 70 clears the screen. Line 80 again uses the OPEN statement to open a file called FILE.FIL. However, this time the file is opened for output—*from* the computer *to* the file. The word OUTPUT means that you are going to write. Again, the file number 1 (#1) is used.

```
10 REM FILE WRITING PROGRAM
20 REM COPYRIGHT
FREDERICK HOLTZ
30 SCREEN 0
40 WIDTH 40
50 KEY OFF
60 A$="FILE ITEM"
70 CLS
80 OPEN "FILE.FIL" FOR
OUTPUT AS #1
90 PRINT#1,A$
100 CLOSE #1
```

Program 55. File Writing Program.

At this point, the computer has opened the file and is ready to write information, in this case the information contained in A$. To accomplish the actual write we use the PRINT# statement, which should not be confused with PRINT. Line 90 tells the computer to write the value contained in A$ to the file "FILE.FIL." Line 100 then closes the file. Congratulations! You have just written to a sequential file. If you run either of the two previous programs at this point, the phrase "FILE ITEM" will be displayed on the screen, since this was the information written to FILE.FIL.

Program 56: Another File Writing Program

This is a slightly advanced version of the previous program. It allows you to insert the name of the file you wish to open for writing, and also to continuously input the items you wish to write to the file. Here you're not limited to a specific file name, nor to a single file item. You can open as many files as you want to (by running the program over and over again), and you can input as many items as you wish.

Line 70 prompts you to enter the name of the file you wish to open. This name is assigned to string variable FI$. The screen is then cleared, and line 90 uses the OPEN statement to open the file named in FI$ for OUTPUT (writing) as #1. Line 100 then prompts you to input the file item. The keyboard input is assigned to A$. Line 110 is an exit line. When you type END, there is a branch to line 150, which closes the file.

```
10 REM FILING PROGRAM-WRITE
20 REM COPYRIGHT
FREDERICK HOLTZ
30 SCREEN 0
40 WIDTH 40
50 KEY OFF
60 CLS
70 INPUT"NAME OF FILE";FI$
80 CLS
90 OPEN FI$ FOROUTPUT AS #1
100 INPUT"FILE ITEM";A$
110 IF A$="END" THEN 150
120 PRINT#1,A$
130 CLS
140 GOTO 100
150 CLOSE #1
160 CLS
170 END
```

Program 56. Another File Writing Program.

```
10 REM FILE APPEND PROGRAM
20 REM COPYRIGHT
FREDERICK HOLTZ
30 SCREEN 0
40 WIDTH 40
50 KEY OFF
60 A$="SECOND FILE ITEM"
70 CLS
80 OPEN "FILE.FIL" FOR
APPEND AS #1
90 PRINT#1,A$
100 CLOSE #1
```

Program 57. File Append Program.

Try running this program and insert 10 or more items. Then run Program No. 2 and read them all out to the screen again. At this point, you have learned the methods by which sequential files are read and written.

Whenever you open a file to be read that doesn't exist, an error message will be displayed on the screen, telling you that the file was not found. There is no real way you can get into trouble here. However, when you open a file for OUTPUT (writing), the computer will create that file on cassette or disk. You must, therefore, be careful, because if you open a file for OUTPUT that already exists, the computer erases all information from that file in the process of recreating it. You *can* open a file that doesn't exist for writing. This creates the file and allows you to input items. However, if you open it one more time for OUTPUT, all those items are erased.

Program 57: File Append Program

Suppose you open a file for OUTPUT and

write several items to it. Suppose then that you wish to go back to the same file and add more items. Obviously, you can't open it for output again, as this will erase the items it currently contains. You simply open the file for APPEND. This is an indication that you wish to open a file that already exists in order to add items to the end of the list. This program will allow you to add one item to the end of a file named FILE.FIL, which already contains items. Line 60 contains the item to be written, which for this example is named "SECOND FILE ITEM." Line 80 opens the file in the APPEND mode, and line 90 uses the PRINT# statement to write the value to the end of the file. If you wish to be able to name your file and add many items to the end of the current list, simply go back to the previous program and change the word OUTPUT in line 90 to APPEND.

To summarize the information discussed to this point, opening a file for input allows it to be read. Opening a file for output creates the file and allows you to write information to it. Opening it for output again erases the contents of the file. Opening a previously written file for append allows you to add information to the end of the current item list contained in that file. Incidentally, if you open a file that doesn't exist for APPEND, the file is created just as if it were opened for output (writing).

Program 58: Complete Filing Program

This is a complete filing program that will allow you to read, write, or append items in a cassette or disk file. Lines 90 through 170 display a program menu on the screen. A menu is a selection of things you can ask this program to do. The four menu selections are:

1) Open file for write
2) Open file for read
3) Open file for append
4) End file program

Line 180 uses the INKEY$ variable to test your input from the keyboard. In lines 200 through 230, you can see the various branches that are designated for the different keyboard inputs.

If you select 1 from the menu, line 200 branches to line 250. Here the screen is cleared, and line 260 prompts you to input the name of the file you wish to open or create for writing. Line 280 prints a warning message stating that if the file you named already exists, it will be erased. You are then given the option to continue or return to the menu and make another selection. Assuming you wish to continue, the screen is cleared in line 330.

Line 340 contains an ON ERROR GOTO statement. This is called an *error trapping* routine. The

```
10 REM COMPLETE FILING PROGRAM
20 REM COPYRIGHT FREDERICK HOLTZ
30 SCREEN 0
40 WIDTH 40
50 COLOR 15,1
60 KEY OFF
70 CLS
80 LOCATE 1,15
90 PRINT"FILE MENU"
100 LOCATE 10,10
110 PRINT"1. OPEN FILE FOR WRITE"
120 LOCATE 12,10
130 PRINT"2. OPEN FILE FOR READ"
140 LOCATE 14,10
150 PRINT"3. OPEN FILE FOR APPEND"
```

Program 58. Complete Filing Program (continued to page 156).

```
160 LOCATE 16,10
170 PRINT"4. END FILE PROGRAM"
180 K$=INKEY$
190 IF K$="" THEN 180
200 IF K$="1" THEN 250
210 IF K$="2" THEN 410
220 IF K$="3" THEN 500
230 IF K$="4" THEN 640
240 GOTO 180
250 CLS
260 INPUT"NAME OF FILE";FI$
270 CLS
280 PRINT"WARNING! IF ";FI$;" ALREADY EX
ISTS"
290 PRINT"IT WILL BE ERASED."
300 PRINT
310 INPUT"DO YOU WISH TO OPEN THIS FILE
(Y/N)";W$
320 IF W$="Y" OR W$="y" THEN 330 ELSE 70

330 CLS
340 ON ERROR GOTO 690
350 OPEN FI$ FOR OUTPUT AS #1
360 INPUT"FILE ITEM";A$
370 IF A$="END" THEN 600
380 CLS
390 PRINT#1,A$
400 GOTO 360
410 CLS
420 INPUT"NAME OF FILE TO BE READ";FI$
430 CLS
440 ON ERROR GOTO 690
450 OPEN FI$ FOR INPUT AS #1
460 IF EOF(1) THEN 600
470 INPUT#1,A$
480 PRINT A$
490 GOTO 460
500 CLS
510 INPUT"NAME OF FILE TO BE APPENDED";F
I$
520 CLS
530 ON ERROR GOTO 690
540 OPEN FI$ FOR APPEND AS #1
550 INPUT"FILE ITEM";A$
560 IF A$="END" THEN 600
570 CLS
```

```
580 PRINT#1, A$
590 GOTO 550
600 CLOSE #1
610 LOCATE 23,1
620 INPUT"PRESS <ENTER> TO FOR MAIN MENU
";ER$
630 GOTO 70
640 CLS`
650 LOCATE 14,13
660 PRINT"PROGRAM TERMINATED"
670 LOCATE 23,1
680 END
690 CLS
700 LOCATE 14,8
710 PRINT"ERROR:FILE CANNOT BE OPENED"
720 FOR DLAY=1 TO 1500:NEXT DLAY
730 CLEAR
740 GOTO 70
```

ON ERROR GOTO statement detects a condition whereby the file could not be written. This might occur when a disk is full, or when there's no cassette file system. If there is an error, there is a branch to line 690, which prints an error message in line 710. Once the message has been printed, there is a slight delay. Line 740 then branches back to the start of the menu print routine. If there is no error, lines 350 through 400 are executed. These lines are nearly identical to those contained in Program No. 4 in this chapter. This is the routine to write information to a file.

If you selected menu option 2, there is a branch to line 410, where you input the name of the file to be read. This name is assigned to FI$. Lines 450 through 490 contain a close copy of the file reading program discussed previously. Selecting menu item 3 brings about a branch to line 500, which accesses the program to add items to a file. When you select menu option 4, the branch is to line 640, where the "PROGRAM TERMINATED" prompt appears and the program ends.

There you have it—a complete file-handling program that can be input to your PC-6300 in about 20 minutes, and can be used over and over again to convert your massive file cabinet to a diskette tape or two.

Program 59: File Item Search Program

Assume that you have set up a file that contains several hundred items and you want to pull one of those items from the file for examination. It's very inefficient to print the entire contents of the file to the screen. It would be better if you have the computer sort through all the items and pull out only the one you're looking for. This program will do just that. It's not terribly useful in its present form, since you have to type in the name of the item you're looking for exactly as it appears in the file. However, this program will be modified later to be much more useful. For now, consider it a practicum in file searching.

Line 70 asks you to input the name of the file to be searched. This is the file name as it appears on the cassette or disk listing. The screen is then cleared, and line 90 asks you to input the item you are searching for. This must be typed exactly as it appeared when first entering it into the file. Again the screen is cleared, and line 110 opens the file you named for input (reading). Line 130 assigns the first item in the file to A$. Line 140 does the actual search. It compares the item you just read from the file with the item you are looking for. If the two match, the item is printed to the screen. Numeric variable COUNT is incremented by 1, and line 150 branches back to line 120 for the next item in the file to be read. Each time the item you are looking for is detected in the file, it is printed to the screen. If the item you are looking for is contained in the file, it is printed to the screen. If the item you are looking for is contained in the file in five different locations, it will be printed to the screen five times.

On the other hand, if the item is never found COUNT will never be incremented, and since it was not assigned previously it will be equal to 0. When line 120 detects the end-of-file condition, the branch is to line 160, which tests the value of COUNT. If COUNT is equal to 0, line 170 is executed. It prints the item you are looking for and tells you that it wasn't found in the file you named. Line 180 is then executed, which closes the file. If the item was found, COUNT will be equal to a number larger than 0, so line 170 is branched over and the file is closed in line 180.

Program 60: Item Portion File Search

This program is an enhanced version of the previous one, but it is far more valuable because it will allow you to search for and display an item contained in a file—without having to input that item in its entirety. For example, let's assume that a file contains (somewhere) the item:

CHARLIE JONES OWES ME $50.00

Wouldn't it be nice to be able to search that file by simply inputting the name Charlie Jones in order to see his record displayed? This program will allow you to do just that. When the program is run, all that is necessary is to type in the name CHARLIE JONES, and the above phrase would be displayed

```
10 REM FILE ITEM SEARCH PROGRAM
20 REM COPYRIGHT FREDERICK HOLTZ
30 SCREEN 0
40 WIDTH 40
50 KEY OFF
60 CLS
70 INPUT"INPUT NAME OF FILE TO BE SEARCH
ED";FI$
80 CLS
90 INPUT"INPUT THE SEARCH ITEM";SI$
100 CLS
110 OPEN FI$ FOR INPUT AS #1
120 IF EOF(1) THEN 160
130 INPUT #1,A$
140 IF A$=SI$ THEN PRINT A$:COUNT=COUNT+
1
150 GOTO 120
160 IF COUNT=0 THEN 170 ELSE 180
170 PRINT SI$;" WAS NOT FOUND IN ";FI$
180 CLOSE #1
```

Program 59. File Item Search Program.

```
10 REM ITEM PORTION SEARCH PROGRAM
20 REM COPYRIGHT FREDERICK HOLTZ
30 SCREEN 0
40 WIDTH 40
50 KEY OFF
60 CLS
70 INPUT"INPUT NAME OF FILE TO BE SEARCH
ED";FI$
80 CLS
90 INPUT"INPUT THE SEARCH ITEM";SI$
100 SI=LEN(SI$)
110 CLS
120 OPEN FI$ FOR INPUT AS #1
130 IF EOF(1) THEN 210
140 INPUT #1,A$
150 L=LEN(A$)
160 FOR X=1 TO L
170 T$=MID$(A$,X,SI)
180 IF T$=SI$ THEN PRINT A$:COUNT=COUNT+
1
190 NEXT X
200 GOTO 130
210 IF COUNT=0 THEN 220 ELSE 230
220 PRINT SI$;" WAS NOT FOUND IN ";FI$
230 CLOSE #1
```

Program 60. Item Portion File Search.

on the screen. You could also type in CHARLIE, JONES, or even CHA, and the file item would still be displayed.

What this program does is take the search item you input and overlay it with items it reads from the file. If you input CHA, the computer will overlay these characters with each part of every file item. When it obtains a match, this file is printed. If you simply input a search item of C, any file that contains a C will be displayed on the screen. Therefore, you will want to put as much information as possible in your search, to prevent other file items having some similarity to the item you're looking for from being displayed.

Line 90 prompts you to input the search item. This is assigned to SI$. SI is assigned the value of the length or the character count in SI$ by the LEN function. Line 120 opens the file you named for reading, and line 140 grabs the first file item. Line 150 assigns to L the length of the file item just read. Line 160 begins a FOR-NEXT loop that counts from 1 to the length of A$ (the value of L). Line 170 assigns to T$ the value of what is found in A$ at position X and for length of SI. (You will remember that SI is the character count of the item you are searching for.) As the loop cycles, T$ simply grabs the required number of letters, starting at the first letter in the item, then starting at the second letter, the third, and so on, until a match is found or the characters in the item are exhausted. Perhaps this can be explained a bit more simply by using the previous example.

CHARLIE JONES OWES ME $50.00

Assume that for a search item you simply typed in

JONES. The number of characters in JONES is five. Therefore, SI equals 5. When the computer comes to the file item that you are looking for, here's how the sequence will go.

1) The word JONES is compared with the first five characters in A$.
2) There is no match here, since the first five characters in our example are CHARL.
3) When the loop recycles, X is equal to 2. Therefore, T$ in line 170 is assigned the value of the five characters in A$, starting with the *second* character.
4) There is still no match, because the search item is JONES and the five characters that are compared with this starting at position 2 are HARLI.
5) The loop continues to advance until it finally gets to the eighth character position of A$. Here, the five characters are JONES. This matches with the input search item of JONES.

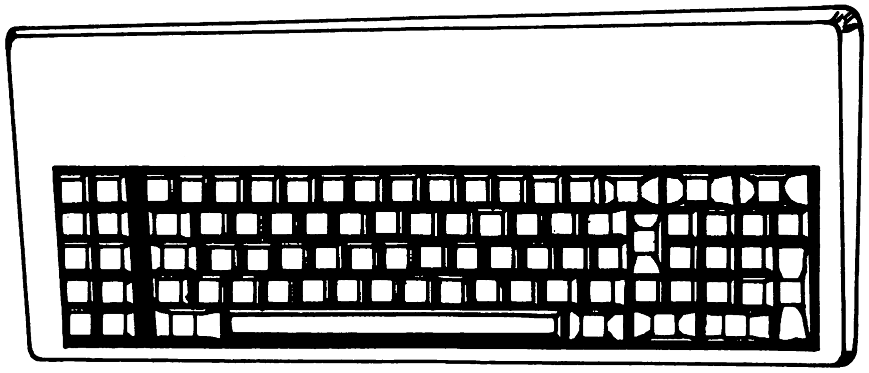
When a match occurs, line 180 determines that T$ is equal to SI$ and the entire contents of A$ (the file item itself) are printed to the screen. The MID$ function is an extremely powerful one in GWBASIC for text manipulation and testing. With it, we have been able to overlay unknown contents with a test pattern of characters to detect the contents of a file item, without having to type in the contents exactly as they appear in the file.

Summary

I think you will find sequential files to be quite useful in nearly any application. Such files are much easier to manipulate than random access files. However, the latter have the advantage of much faster access when disk storage is used. Again, random access files are purposely not discussed, since filekeeping in this mode is far more intricate than it is with sequential files. Random access filekeeping is a subject that is best left to the more experienced computer hobbyist who has already done a fair amount of work with sequential files.

Remember, to read, write, or append a file, it is first necessary to OPEN it. Opening a file for INPUT allows you to access for reading each item it contains. Opening a file for OUTPUT creates a file on cassette or disk and allows you to write information to it. Opening a file for APPEND allows you to write information to the end of a file which already contains items.

Chapter 8



AT&T PC

Music programming is generally quite simple in that it usually requires only a statement or two—followed by numeric or alphabetic designators to specify each note.

Although the SOUND statement does have some application in music programming, time and efficiency can be greatly enhanced by using the PLAY statement, which allows you to use fewer commands and statements. Therefore a music program based upon PLAY statements will generally be far shorter than an equivalent program using SOUND statements.

The SOUND statement requires a frequency command and a duration command. The latter is given in clock cycles. This requires a great deal of interpolation when using this statement solely for the purpose of programming music. For example, if you wanted to use the SOUND statement to produce a middle C, the frequency command would be 523.250. This would be followed by the duration commands, which would be the most difficult to calculate, because you would have to convert clock cycles to the various note hold durations based upon tempo.

The PLAY statement avoids most of these problems. Several command options allow you to program music in different manners. A middle C would be programmed as a quarter note by simply typing "PLAY O3C4." The O3 command sets the octave in which middle C is located. The alphabetic designator C specifies the note C, and the 4 means the note is to be played as a quarter note. Subsequent notes can be added to this statement by typing their alphabetic designators (the note name) followed by their numeric designators, which specify hold times. As long as you stay in the middle octave range, no further O designators are required. Once a specific octave is named, all notes will be played in that octave until another O command causes a shift to occur.

Using the SOUND statement, duration is specified in clock ticks. All duration commands are referenced to the clock cycle time. However, using the PLAY statement, note duration is referenced to every other note. For example, a C8 (an eighth note) will be played for half the time of a C4 (a quarter note). Duration may be as short as 1/64 of a whole note.

Although it is possible to program a complete tune using the SOUND statement and separate frequency and duration commands for each note specified, increasing or decreasing the tempo would necessitate changing every duration command to reflect the new tempo. That would be a great deal of work. With the PLAY statement, however, all notes are entered in standard musical equivalents (whole, half, quarter, eighth, sixteenth and so forth).

The tempo command (T) is separate from these and ranges from 32 to 255. If no tempo command is given the default tempo is 120, *moderato* in musical terms.

Thus, once the note has been entered the tempo may be changed in any way; each note will still be held for the proper duration in relationship to all other notes in the tune.

For example, let's assume a four-note piece consisting of C8, D4, F16, and B1. This is a string consisting of an eighth note, a quarter note, sixteenth note, and a whole note, respectively. If the whole note is held at one tempo for a total of one second, then the quarter note will be held for one-quarter second; the eighth note, for an eighth second; and the sixteenth note, for one-sixteenth second. Let's assume that the tempo is doubled by changing the tempo command. Assume also that the whole note is held for one-half second at this tempo. The eighth note will then be held for one-eight of a half second (one-sixteenth second), and all other notes in this string will have equivalent shortened durations.

Using the SOUND statement to produce the same tune would require four different statements followed by four different sets of frequency and duration commands. To increase tempo, it would be necessary to halve each of the four duration commands (for double time). Using the PLAY statement, however, it is only necessary to change the tempo command. Furthermore, less memory is required for storage, since in the former example this simple four-note song may be easily entered on one program line using a single PLAY statement, whereas a minimum of four program lines would be required using the SOUND statement in the latter.

Pauses or rests are specified in approximately the same manner as musical notes, but the letter P is used followed by a numeric designator to specify the pause length. A P4 is equivalent to a pause for the same duration as the hold time of the quarter note produced under the play tempo. You can see that this makes musical programming easier, since notes are named and their durations specified in exactly the same manner as in musical manuscripts. To play sharps and flats, the note is simply followed by the standard sharp symbol (#). Alternately, you may use a plus sign (+) after the note to specify a sharp and a minus sign (-) to specify a flat.

GWBASIC goes a step further to help you write programs directly from musical manuscripts. In manuscripts one often sees a note followed by a dot or period. Musically, this means that the note is to be held for 1 1/2 times the duration of an identical note without the dot. A half note followed by a dot is held for the equivalent of 1.5 times one half, or 3/4 of a whole note. Using the PLAY statement, the same note (let's call it a C) would be specified as "C2." Many of the programs in this chapter include this type of designation. You should be absolutely certain you do not skip a dot while inputting a program.

Another method of specifying notes involves numeric designators only. These are used in conjunction with the N command. The numeric designators range from 0 to 84. A 0 is a rest; each succeeding number specifies individual notes over a seven-octave range. I don't use this style of programming very often, as I find the direct mode much easier. However, this can be put to good advantage when a PLAY statement using the N command is placed within a FOR-NEXT loop. The loop output (numerically) will then determine the note to be played.

Each of the musical pieces presented in program form was written directly from musical manuscript. It takes about an hour to become accustomed to this form of programming. Some songs are quite simple; others are complex. There are a number of musical programs reproducing the compositions of Johann Sebastian Bach, one of my favorite composers. His complex melody lines are especially suited to the microcomputer, in my opinion. You will also see popular songs, children's songs,and a program or two that combine the best of different types of music. I think you will find all of them pleasing and the programming exercise to be quite educational.

Program 61: Mary Had A Little Lamb

Program 61 will play the basic strain of "Mary Had a Little Lamb." It effectively uses only a single program line, in that the REM and END statements are certainly not necessary for proper execution. This program uses the PLAY statement (in line 20) in conjunction with note commands, a tempo command, and the MF command, which places the music in the foreground mode. I chose a tempo of 170, which is fairly rapid, but you can adjust the tempo by changing the number sequence following the letter T. The alphabetic designators following the command are the actual notes that make up the song. There are three rest commands (P4) that create a quarter-note rest. Actually, the numeric designator is unnecessary, since the other notes are not followed by similar designators. Therefore, each is automatically played as the equivalent of a quarter note. You will note several minus signs occurring within the music command string. These identify flats.

If the program is inserted to complement an already existing program, only line 20 will be needed. This allows you to add music at the expense of a minimum amount of memory. This program is written from a child's music book; each note was read from the staff line and input as the same equivalent note in the PLAY statement commands. If you can read music, musical programming will be easier. If you can't read music you can learn to do so in a few days.

The microcomputer makes an excellent study tool. It may be difficult to decipher a complex musical passage in the mind, but the notes, input in a computer program, allow students to hear how the particular passage should sound. This can then be emulated using a musical instrument. The computer never makes a musical error. It will always play the specified note for the specified duration. The only possibility for error lies with the human being. The computer is totally dependent on him for input.

Program 62: Blue Skies

Program 62 shows another method of musical programming, which will produce the strains of the old standard "Blue Skies." This song has been around for a number of years, having been sung by Danny Kaye in the movie *White Christmas*. More recently, country music star Willie Nelson again made the song popular.

Instead of inserting the musical note designators directly in a PLAY statement, this program commits them to a DATA statement. A subsequent READ statement pulls each note individually from the DATA line. The PLAY statement in line 40 uses the X command to play a string statement. The GOTO command in line 50 branches back to line 20, where the next element is read. In other words each note is pulled from the DATA statement by a machine operation which is constantly repeated until the data runs out.

The variable A$ is composed of only one note, but the string is constantly refreshed by the GOTO

```
    10 REM "MARY HAD A LITTLE LAMB"
    20 PLAY"MFT170GFE-FGGGP4FFFP4GB-B-P4GFE-FGGGGFFGFE-"
    30 END
```

Program 61. "Mary Had a Little Lamb."

```
10 REM "BLUE SKIES"
20 READ A$
30 IF A$="END" THEN 70
40 PLAY "XA$;"
50 GOTO 20
60 DATA C,G2,F8,E-16,F16,P8,G2,F8,E-16,F
16,P8,G3,C4,P6,C8,E-16,P8,C4,END
70 END
```

Program 62. "Blue Skies."

statement in line 50. Line 30 tests for the appearance of the last item in the DATA statement.

I used the word "END", but anything will work as long as it is not one of the musical notes already read. Without building this feature into the program, the READ statement would still be trying to pull data elements from the DATA statement after the line has been exhausted. That would result in an error message on the screen: "OUT OF DATA IN 20". This has no effect on the musical run of the program; however I always try to avoid error messages as these can confuse the beginning programmer.

Using this program all of the notes are pulled from the DATA statement line. When the word "END" is encountered, line 30 branches to line 70, which ends the program. To replay the song, run the program again.

One might ask why such a method was chosen to play this simple song rather than the simpler method used previously. It is true that the latter method is more complex. It requires more program lines, and each note must be separated by a comma within the DATA statement. It also requires several different branches. The reason this method is used is to demonstrate a different way of accomplishing the same result using the AT&T Personal computer. In certain situations, this method is to be preferred over the former.

Program 63: Jingle Bells

Program 63 is identical to the "Blue Skies" program, except the data elements have been changed to produce the song "Jingle Bells." The same lines have been added as before to detect the end of the data string in order to stop the program run. One of the advantages of a program using READ/DATA statements is that an original song can be expanded or more tunes added by simply adding further DATA lines. Of course one must remove the word "END" from the first data line to prevent execution from being halted before the new lines can be read.

Program 64: Go Tell Aunt Rhodie

Program 64 is another example of this form of

```
10 REM "JINGLE BELLS"
20 READ A$
30 IF A$="END" THEN 70
40 PLAY"XA$;"
50 GOTO 20
60 DATA E,E,E,P8,E,E,E,P8,E,G,C,D,E2,F,F
,F3,F8,F,E,E,E8,E8,G,G,F,D,C2,END
70 END
```

Program 63. "Jingle Bells."

```
10 REM "GO TELL AUNT RHODIE"
20 READ A$
30 IF A$="END" THEN 70
40 PLAY "XA$;"
50 GOTO 20
60 DATA A2,A4,G4,F2,F2,G2,G4,B-4,A2,F2,A
2,A4,G4,F2,F2,G4,F4,G4,A4,F3,END
70 END
```

Program 64. "Go Tell Aunt Rhodie."

musical programming. The song has been changed to "Go Tell Aunt Rhodie" by substituting different data elements. Program 65 is the first program with the data lines from the next two added. This program will play all three songs, one after the other, before execution is halted. While a program that uses READ/DATA statements to play one song may be more complex than one using the direct mode, if one program is used to play several songs, READ/DATA statements often speed programming and certainly lessen program complexity. Using this method, you could write a single program to produce thirty or more different songs. Each song would be committed to a separate DATA line. Therefore, a thirty-song program would include thirty DATA statements. The remainder of the program could remain the same, regardless of the number of DATA statements used. The last data line would end with the word "END", which would bring about a branch to line 1000, where the program halts. You can add more selections by merging other musical program with the notes specified in DATA statements.

Notice that the information in some of these programs includes individual letters and also letters followed by numeric designators. Without the

```
10  REM "MULTISONG PROGRAM"

20  READ A$

30  IF A$="END" THEN 1000

40  PLAY "XA$;"

50  GOTO 20

60  DATA C,G2,F8,E-16,F16,P8,G2,F8,E-16,F16,P8,G3,C4,P6,C8,E-16,P8,C4

70  DATA E,E,E,P8,E,E,E,P8,E,G,C,D,E2,F,F,F3,F8,F,E,E,E8,E8,G,G,F,D,C2

80  DATA A2,A4,G4,F2,F2,G2,G4,B-4,A2,F2,A2,A4,G4,F2,F2,G4,F4,G4,A4,F3,END

1000  END
```

Program 65. Multisong program.

```
10 REM "MUSICAL PROGRESSION"
20 FOR X=1 TO 84
30 PLAY"N=X;"
40 NEXT
50 END
```

Program 66. Musical Progression.

designator the note is played as a quarter note. Therefore to speed programming time, I have often omitted the "4" (quarter note designation), since this is the default state. In other words an E4 within a Play statement will be output the same if the 4 is omitted.

MUSICAL FOR-NEXT LOOPS

Program 66 shows a musical progression using the End command with the PLAY statement. Each note in a seven-octave scale is represented by a number. Line 20 begins the FOR-NEXT loop, which begins with 1 and ends with 84. This means that the entire range of notes will be played automatically. The END command is inserted in the PLAY statement in line 30, which specifies that N is equal to the value of X. I elected not to start with a 0 (FOR X = TO 84) because this value is always a pause or rest. The NEXT statement in line 40 allows the loop to recycle and step up by one place each time. The first note that is played is one, the next is two, then three and so on.

This program will allow for an output equivalent to starting at the bottom of the piano keyboard and playing every note in order to the very top . . . and then some. The last few notes will be so high as to be practically inaudible.

This type of designation and the use of FOR-NEXT loops are not very conducive to accurate musical programming. for creating certain types of musical sound effects, however, this mode may be ideal. Because of the recycling of the loop, an entire range of notes may be played with a minimum of programming time.

Program 67 may be thought of as the previous program in reverse. The only change is in line 20, where the loop counts from 84 to 1 in steps of –1. The first program starts at the bottom of the computer's musical range and proceeds to the top. The latter program starts at the top and counts to the bottom, playing one note during each loop cycle. If you combine these two programs, the entire chromatic scale will be reproduced by the computer. This will start at the bottom of the range, go to the top, and then proceed to the starting point.

The fact that loop counts may be stepped in different increments allows for a specific series of notes to be selected. This can be put to advantage musically in some instances by committing a certain portion of a musical piece to a FOR-NEXT loop.

Program 68 is identical to the first one, except the loop is not stepped in increments of 1. The STEP command here causes the count to skip every other number; thus every other note is played. The first note will be a one, the second a three, the third a five and so on. With the proper step increments, many standardized musical scales can be produced using the same basic program under discussion in this section. You can randomize the STEP command to produce some unusual musical sound effects, none of which may be

```
10 REM "MUSICAL PROGRESSION - REVERSE OR
DER"
20 FOR X=84 TO 1 STEP -1
30 PLAY"N=X;"
40 NEXT
50 END
```

Program 67. Musical Progression, Reverse Order.

```
10 REM "MUSICAL STEPS"
20 FOR X=1 TO 84 STEP 2
30 PLAY"N=X;"
40 NEXT
50 END
```

Program 68. Musical Steps.

predicted in advance. The beginning and ending loop numbers may also be randomized for further effects.

The numeric designation system outlined here may also be used in a similar manner in programs discussed previously. READ/DATA statements could be incorporated in a program and the numeric information included in the data lines. That would be identical to a previous programming method, but numeric designators would replace alphabetic ones.

Although most of us are accustomed to working with numbers on a microcomputer rather than letters, one must also remember that musical manuscript contains alphabetic and numeric designators. Therefore the ability to input various notes using an alphabetic and numeric format is much more convenient, especially for those who are more familiar with music than computers. Far less human interpolation and extrapolation are required in making the transition from manuscript to computer form, so this mode is to be preferred in most instances. However, a programmer who is more accustomed to working with computers than music may find that numeric-only designators are more to his or her liking.

Both can be used to accomplish the same thing, so choose the method you think most appropriate. The numeric method is limited in that it is inconvenient to specify different note lengths, something which is greatly simplified when alphabetic and numeric designators are combined. The numeric-only system is a shortcut for producing very simple songs in which each note is held for the same length as all others. It is possible to combine alphanumeric and numeric-only designations, but for the beginning programmer, this may become a bit confusing. Alphabetic-numeric designation will be used most of the time, but if one runs into a long series of quarter notes, one can immediately revert to the numeric-only style. The purpose of this discussion is to point out the available options. I think most programmers will choose the alphabetic-numeric mode of programming, as I have done throughout most of this chapter.

BACK TO BACH (J.S.)

Johann Sebastian Bach, a German composer of great renown, had the inborn ability to take simple melodies and combine them in such a manner as to produce a pleasingly complex interweaving of sounds. His pieces are often performed today on a single piano with little or no musical accompaniment. The harpsichord is often the instrument which communicates his delicate phrasings. Being a fan of Bach, it is only natural that I include some of his musical pieces in this chapter. I think of the audio output of the AT&T PC as being a delicate and quite precise musical form. I feel similarly about most of Bach's music. Therefore, the mating of Bach with the AT&T PC is only natural. If you're not a fan of the classics, please bear with me and try some of the programs presented in this chapter. You may not like them as much as some of the more popular pieces, but you may find them eminently more interesting from a technical point of view.

Program 69: How Gentle Is the Rain

You may think you've missed something or the publisher has omitted 40 pages from this chapter, but this popular hit song from the sixties is based upon a piece that Johann Sebastian Bach wrote before the word "Beatles" meant anything more than a misspelling of the hard-shelled insect. I specifically chose this song to gently acquaint some of you to Bach, although I had to work out the song on a piano, since I didn't have the musical manuscript.

The song (Program 69) is produced by two PLAY statements in lines 20 and 30. I chose the music foreground mode (MF) because this song contains more than 32 notes, the buffer limit for the background mode. When songs are played in

```
10 REM "HOW GENTLE IS THE RAIN"
20 PLAY "MF T130;O2;G2;L4;C8;D8;E8;F8;L2
;G;C;P16;L2;A;L4;F8;G8;A8;B8;O3;L2;C;O2;
C;P16;F;L4;G8;F8;E8;D8;E2;L4;F8;E8;D8;C8
;L2;O1;B;O2;L4;C8;D8;E8;C8;D2;O2;A;G;"
30 PLAY "O2;G2;L4;C8;D8;E8;F8;L2;G;C;P16
;L2;A;L4;F8;G8;A8;B8;O3;L2;C;O2;C;P16;F;
L4;G8;F8;E8;D8;E2;L4;F8;E8;D8;C8;L2;D;O2
;L4;E8;D8;C8;O1;B8;O2;C2;"
40 END
```

Program 69. "How Gentle Is the Rain."

background all of the musical information is held in the buffer, which outputs it to the speaker on a programmed time basis. This effectively removes the processing from the main part of the computer, permitting it to tackle other problems. The buffer receives the musical information on a real-time basis (occurring in a fraction of a second) and then outputs it in a time which is based upon the tempo and the individual note durations, all of which are specified in the program information. This allows the computer to continue program execution past the music point; there is thus the capability of playing limited music while, for instance, information is being printed on the screen.

The notes in this program are too numerous to be fully committed to the background mode, so the foreground mode was chosen. The computer will continue processing after the song has been entered or until it has finished. The second command in line 20 specifies the tempo for the rest of the musical piece. Tempo may be changed by adding other T commands, but this is not necessary for this particular number. The rest of the commands specify notes, durations, and octaves. I will not discuss each command, but we will examine the first few.

Following the tempo command is an 02, meaning that all subsequent notes are to be played in octave 2, which begins with the note C (one octave below middle C) and ends with B. The next command specifies the note G as a half note. Then we come to C, D, and F, all of which are eighth-notes. Throughout the PLAY line you will see several octave changes. These are used to access higher or lower notes which fall outside of the second octave. In some pieces octave changes will be quite numerous, which confused me when I first began programming. If you remember that each octave begins with a C and ends with a B, you should have little difficulty.

Let's assume that you're in octave 2 and have just programmed a B. The next note is a C (above the B). This indicates that it will be necessary to go to octave 3 in order to accurately reproduce the C which begins a new octave. However, the reverse is true. If you've just input a C in one octave and then want to program the B which lies just below it (on the musical staff), it will be necessary to switch to the next lower octave (in this case, 01). After a bit of practice, the insertion of octave commands becomes almost automatic and will give you little trouble, especially if you're familiar with reading musical manuscripts.

Program 70: Polonaise

Program 70 is treated in approximately the same manner as the previous one, and for that manner, many of the programs that follow. "Polonaise" is a gentle piece composed by J.S. Bach.

This program commits the musical information to six program lines, all containing PLAY statements. The computer limits you to 64 characters per line, which may be one reason several PLAY statements are needed for a single

```
10 REM "POLONAISE" BY J.S. BACH
20 PLAY"MFO3G8.A16B-4O4C4O3A8A16B-16O4C2
O3B-8B-16O4C16D8G8C8G8O3B-8A16B-16G2"
30 PLAY"B-8.O4C16D4F4D8C16O3B-16A16B-16O
4C16O3A16F4"
40 PLAY"O4F8D8O3B8O4F8G16F16E-16D16E-8C8
O3A8O4C8F16E-16D16C16D8C16D16E-8"
50 PLAY"D8C8O3B-8A16B-16O4C16O3A16B-4O2B
-4O4D4E-4O3G4"
60 PLAY"F#8F#16G16A8D8F8A8"
70 PLAY"O4D4E-4O3G4F#8F#16G16A8D8F8A8O4D
8D16E-16D8D16E-16D8G8O3G8A16B-16G4O1G4"
80 GOTO 20
```

Program 70. "Polonaise."

song. However, many musical pieces are written in different connective segments, and I will often use a separate PLAY statement when a new section in a piece takes effect.

Looking at line 20, you can see that this number is also played in the foreground mode, and the first note is in octave 3(O 3). The first note is a G, a dotted eighth note. This means it will be held for 1½ times its normal duration. You can see that the command designation is "G8." This mathematically simplifies things, in that anytime you see a dotted note on a musical staff, you simply input the dot next to the commands that play that note. You will notice several octave changes in each program line. Part of Bach's magic is his use of closely spaced notes in different octaves. This gives the impression of constant and rapid movement, a trait with which Bach's music has long been associated.

I enjoy this piece so much that I built the program as an endless loop. As soon as the first playing has finished, the GOTO statement in line 80 returns the program to its beginning and the song is heard again.

This program was written directly from the musical manuscript for piano and is completely accurate. Admittedly the score I used was a revised interpretation of Bach's piece, but it closely simulates the original work. The program took a total input time of about 15 minutes, including debugging. If you do much music programming, you will from time to time leave out a sharp or a flat, or even program an incorrect note. The same is true if you play a musical instrument. Sooner or later you're going to hit a "clam." If you are not familiar with Bach's music, you should not have much difficulty identifying a wrong note just by listening to the computer output. There is nothing especially dissonant about any of his pieces; a wrong note should stand out like a sore thumb. For those of you who are not very familiar with music and its many terms, dissonant means discordant. Some of the modern musical pieces may include a fair amount of dissonance, which is intentional (or so the composers say), and identification of improper notes is more difficult. Bach's pieces, however, are quite logical. They are somewhat easier to debug and are more interesting than simple children's songs, which are also logical but are often quite boring.

Program 71: Intrata

Program 71 is another composition by J.S. Bach, "Intrata." Each of the commands is separated by a semicolon, for note-by-note clarification. The semicolon has no effect on the program run. This is another intricate piece beautifully output by the AT&T Personal computer. Placing

```
10 REM "INTRATA" BY J.S. BACH
20 PLAY"MF O3 D4 G8;F#8;G8;D8;E8;F#8"
30 PLAY"G8;F#8;G8;D8;E8;F#8;"
40 PLAY"G8;F#8;G8;A8;B8;O4;C8;D2"
50 PLAY"O1;D4;G8;F#8;G8;D8;E8;F#8;G8;F#8
;G8;D8;E8;F#8;"
60 PLAY"G8;F#8;G8;A8;B8;O2;C8;D2;"
70 PLAY"O4;C8;D8;D8;C8;C8;O3;B8;O4;C8;O3
;B8;O4;C2;O4;D8;C8;C8;O3;B8;B8;A8;B8;A8;
B2;"
80 PLAY"O3;D4;G8;F#8;G8;D8;E8;F#8;G8;F#8
;G8;D8;E8;F#8;G8;F#8;G8;A8;B8;O4;C8;D2;O
3;G8;B8;A4;G4;O4;G8;B8;A4;G4;G8;F#8;E8;D
8;C8;O3;B8;A8;O4;D8;O3;B2;O1;G8;F#8;E8;D
8;C8;O0;B8;A8;O1;D8;O0;B2;O4;D8;C8;C8;O3
;B8;A8;G8;F#8;A8;G1;"
90 END
```

Program 71. "Intrata."

semicolons between each note designation consumes more programming time, but makes debugging easier.

Debugging is best accomplished by inserting a very slow tempo command (T20). You can then follow each note on the musical score until the wrong one is identified, and then replay the piece while reading the PLAY command for each note. When all the glitches are found and corrected, the tempo can be increased to its original speed.

No tempo command is used in this program because the default tempo is 120, which is about right for this piece. If you want to have some real fun add "T255", the maximum tempo, and Bach is spewed forth at almost real-time speeds.

Program 72: Minuet

By definition a *minuet* is a slow, graceful dance in three-quarter time. In purely musical terms a minuet is the music by which the dance is enacted. Bach's minuets cannot be classified as slow, but they do follow the three-quarter time format and are graceful, if a bit fast.

```
10 REM "MINUET" BY J.S. BACH
20 PLAY"MS O4C4C4D4E-4E-4F4G4G4A-4G16F#1
6G16F#16G16F#16E16F#16G4G8A-8F8E8F4F8G8E
-8D8E-4C8O3B8O4C4D4O3G2."
30 PLAY"O3G4G4A-4B-8A-8G8A-8B-4C4E4G12A-
12B-12A-8G8F2F4F4G4A-8G8F8G8A-4O2B-4O3D4
F12G12A-12"
40 PLAY"G8F8E-2G4A-4A4B-4B4O4C4O3B4O4C4D
4E-4E4F4D4E-4E4F4F#4G4O3G4O4F8E-8D8E-8C2
."
```

Program 72. "Minuet."

Program 72 plays the Bach piece, "Minuet" at a tempo of T120 (the default tempo). You will see a number of sharps and flats and a profusion of octave changes, as there is a great deal of switching back and forth between octaves 3 and 4. This is one of Bach's more pleasing features, and is highly enjoyable when produced by the computer.

Program 73 produces Bach's "*tempo di minuetto*." This is a more complex piece, characterized by sudden octave changes. You will see a number of sharps and flats specified in the PLAY statement lines.

ADDITIONAL NOTES

Many classical pieces include a number of what are commonly called *grace notes*. The best way to describe grace notes is to say that they are more or less dispersed into a number with little or no specified mathematical patterns. The human performer is expected to play them very rapidly and make adjustments (in time) to the rest of the measure. For example, a whole note may be specified in one bar immediately preceded by a miniature series of grace notes. The whole note will normally consume the entire measure, but in this case the grace notes will consume a small amount of time. The performer will normally play the grace notes as quickly as possible, then hold the whole note for whatever time remains in that measure. Two different performers will probably play the grace notes a bit differently.

When programming grace notes on the AT&T PC, I usually specify them as 64th notes or 32nd notes. Many musical manuscripts will provide a hint for playing these notes; however, it is still necessary to instruct the machine as to the duration the main note or notes in that measure are to be held. When you start the musical programming on the AT&T PC, you may wish to eliminate the grace notes at first, then add them later when your musical programming proficiency has improved. You will have to calculate the total amount of time consumed by the grace note passage and then subtract that from the remaining note or notes in the single measure. If you are working with a fast tempo, you can probably drop a 32nd note or two and not be able to tell the difference in timing; but if you are a perfectionist, you will want all measures to be mathematically equivalent.

Some musical pieces require *trills*, which are usually two notes played together very rapidly to produce a birdlike warble. These are often played as individual 64th notes and occasionally as 32nd notes. At rapid machine tempos, trills and complex grace note passages often seem to run together. For trills, this is desirable; but for grace notes, this may not be so. You may try slowing the tempo for bet-

```
10 REM "TEMPO DI MINUETTO" BY J.S. BACH
20 PLAY"MFO3A4O4F8E8D8C#8D4O3A4B-4C#8E8G
8B-8A8G8F4E8F8D4F4B-8A8O4D8C8F4E8D8C8O3B
-8A8B-16O4C16O3F4E4F2.A4O4F8E8D8C#8D4O3A
4B-4C#8E8G8B-8A8G8F4E8F8D4F4B-8A8O4D8C8F
4E8D8C8O3B-8A8B-16O4C16O3F4E4F2.O4A4O3F8
O4A8G8F8E16F16G8C2F4O3F8O4F8E8D8C#16D16E
8O3A2A8B8O4C#8
30 PLAY"D8E8F8G8E8C#8B-8A8G8F16E16D8E4C#
4D2."
40 PLAY"A4O3F8O4A8G8F8E16F16G8C2F4O3F8O4
F8E8D8C#16D16E8O3A2A8B8O4C#8D8E8F8G8E8C#
8B-8A8G8F16E16D8E4C#4D2."
```

Program 73. "tempo di minuetto."

```
10 REM "THE TWELVE DAYS OF CHRISTMAS"
20 PLAY"MF T125 O2C8C8C4F8F8F4E8F8G8A8B-
8G8A4.
30 PLAY"MF B-8O3C4D8O2B-8A8F8G4F2."
40 FOR X=1 TO 4
50 PLAY"MF T125 O2C8C8C4F8F8F4E8F8G8A8B-
8G8A4.
60 GOSUB 80
70 NEXT X
80 FOR A=1 TO X
90 IF X=4 THEN 140
100 PLAY"MF O3C4O2G8A8B-4"
110 NEXT A
120 PLAY"MF A8B-8O3C4D8O2B-8A8F8G4F2."
130 RETURN
140 PLAY"MF O3C2D2O2B..O3C1"
150 PLAY"MF C8O2B-8A8G8F4B-4D4F4G8F8E8D8
C4A8B-8O3C4D8O2B-8A8F8G4F2."
160 FOR X=1 TO 7
170 PLAY"MFO2C8C8C8C8F8F8F4E8F8G8A8B-8G8
A2"
180 FOR A=1 TO X
190 PLAY"MF O3C8C8O2G8A8B-8G8"
200 NEXT A
210 PLAY"MF O3C2D2O2B..O3C1"
220 PLAY"MF C8O2B-8A8G8F4B-4D4F4G8F8E8D8
C4A8B-8O3C4D8O2B-8A8F8G4F2."
230 NEXT X
240 PLAY"MF T75F4O2A8B-8O3C4D8O2B-8A8F8G
4F2."
250 END
```

Program 74. "The Twelve Days of Christmas."

ter definition of each grace note. Alternately, you can specify each grace note with a slightly longer duration designator and make appropriate time adjustments later in the musical measure. Grace notes and the way they are played is pretty much left up to the musician, who will develop a feel for their presentation. The same applies to the musician who uses the computer as his instrument. The computer, however, cannot "feel" the phrasing of a certain passage. The programmer will have to perform this function for the computer mathematically.

A MORE COMPLEX MUSIC PROGRAM

Program 74 will produce all the versus of "The Twelve Days of Christmas." This is a song nearly everyone is familiar with. Although there is much repetition, there are subtle differences in certain sections of verses. It is possible to use straight programming to produce this song, meaning that each verse will be given its own program line. It is not necessary to establish FOR-NEXT loops to repeat those bars that are constantly repeated in the song. Straight-line programming will take less forethought and planning, but will result in a very long program.

The program takes advantage of FOR-NEXT loops and results in a relatively short program. Those verses which are repeated are committed to FOR-NEXT loops, and proper branches are inserted to pick up those verses which do not conform to the loop material.

Line 20 and line 30 play the first verse. A FOR-NEXT loop is then entered (line 40), and the first portion of that same verse is repeated. There is a GOSUB in line 60 which causes the program to enter a nested loop. Line 100 plays the "Two Calling Birds, Three French Hens, and Four Turtle Doves" song lines. Line 90 tests for the condition X = 4. When this occurs, there is a branch to line 140, which plays the "Five Golden Rings" portion of the song. Line 150 is the countdown verse to "A Partridge in a Pear Tree."

We are now at "Five Golden Rings." The next seven verses begin with the FOR-NEXT loop in line 160, which cycles seven times. Line 170 enters the "On The Sixth Day . . ." portion of the song, and another nested loop is then encountered in line 180 which plays the song line associated with "Six Geese A Laying" and the remaining verses. Notice that the loop starting in line 180 counts from 1 to X. When the loop times out, "Five Golden Rings" is played again in line 210. Line 220 is a repeat of line 150 and plays the verses which count down to the partridge again. The NEXT statement in line 230 branches back to line 160 for the next cycle of the loop. During this next cycle, X is equal to 2. Therefore, the nested loop will play the stanza in line 190 two times before reaching the "Five Golden Rings" sequence again in line 210. When this last loop has timed out completely, the finale is played in line 240. Notice that the tempo here has been slowed from 125 to 75. This gives us the needed *ritard* for the final strains. This last portion is a repeat of the last few bars of the song, producing "And a Partridge in a Pear Tree."

The FOR-NEXT loops allow you to repeat repetitive stanzas without having to put in a separate PLAY statement for each one. The loops step the number of times corresponding to the number of times the verse is repeated in the song. There are two major internal verse repeats. One begins with the second day of Christmas and ends at the fourth day of Christmas; the other begins at the sixth day of Christmas and ends at the twelfth. The "Five Golden Rings" portion is a separate entity, but it must be inserted at the proper time, which is different as each day is encountered.

Admittedly, it took me longer to write this program in the form shown than it would have, had I used straight-line programming throughout. The program shows a more efficient method of producing this song and requires much less input time. There is less room for note errors as well, since there are fewer note commands and Play statements. If you do enter an incorrect note somewhere, it will probably crop up several times as it is accessed by the loops. Once you have identified and corrected this note, however, the correct version will also be repeated. I think this program will be much easier to debug than a straight-line version, and much less memory is required.

Program 75: Sing-Along Christmas Song

Program 75 combines music and text, similar to the bouncing ball sing-alongs you have probably seen on television. While it would be possible to duplicate the bouncing ball on the screen using graphics, this program does not go quite so far. It simply prints the words at the center of the screen as a particular musical verse is played. LOCATE statements are used to place the words at the center of the screen. As a new verse appears, it writes over the old one. Line 40 prints the first phrase line, "Joy to the World," and line 50 plays the music. Immediately after this PLAY line is completed, lines 60 and 70 print the next few words, followed by the music in line 80. This continues throughout the remainder of the program. You will notice in line 160 that the final word (sing) is followed by a number of spaces. This is because the phrase in line 160 is shorter than the one in line 130, so the spaces simply write over the previous line with screen blanks. In other words, the spaces cover up for the lack of length in the new line. The same method is used in line 250, which prints "Merry Christmas"

```
10 REM "JOY TO THE WORLD" SINGALONG
20 CLS
30 LOCATE 14,30
40 PRINT "JOY TO THE WORLD"
50 PLAY "MF O3D4C#8.O2B16A4.
60 LOCATE 14,30
70 PRINT"THE LORD IS COME"
80 PLAY "G8F#4E4D4.
90 LOCATE 14,30
100 PRINT"LET EARTH RECEIVE HER KING
110 PLAY "A8B4.B8O3C#4.C#8D4.
120 LOCATE 14,30
130 PRINT"LET EV'RY HEART PREPARE HIM ROOM"
140 PLAY "D8D8C#8O2B8A8A8.G16F#8O3D8D8C#8O2B8A8A8.G16F#8
150 LOCATE 14,30
160 PRINT"AND HEAVEN AND NATURE SING                    "
170 PLAY " F#8F#8F#8F#8F#16G16A4.
180 LOCATE 14,30
190 PRINT"AND HEAVEN AND NATURE SING"
200 PLAY"G16F#16E8E8E8E16F#16G4."
210 LOCATE 14,30
220 PRINT "AND HEAVEN AND HEAVEN AND NATURE SING"
230 PLAY" F#16E16D8O3D4O2B8A8.G16F#8G8F#4E4D2"
240 LOCATE 14,35
250 PRINT"MERRY CHRISTMAS                               "
```

Program 75. "Joy to the World."

```
10  REM "WHATEVER WILL BE, WILL BE" (QUE
 SERA SERA)
20  PLAY "MF ML T150 O2C4D4E4G2E4G2E4G2.
"
30  PLAY "E4G4E4A4G2A4G4E4F2.F2.B4O3C4D4
C4O2B2"
40  PLAY "O2A4B4O3C4O2B2.F4G4A4G2F4E2."
50  PLAY "F4O3C4.O2B-8A2F4A2.A2B4O3D4C4O
2A4"
60  PLAY "O2G2C4E2.E2F#4A4G4E4G2D4G2.G4D
4E4F2O1B4O2C2.C2.C4"
70  PLAY "O2D4E4F2O1B4O2C2.C2.C2"
80  END
```

Program 76. "Whatever Will Be, Will Be."

at the end of the song.

This is a very simple program to write, since you first program the music line and then return to add the text. You will input this program as shown on a line-by-line basis.

I had originally intended to add an MB with the PLAY statements instead of an MF. The former places the music information in a buffer and will play it while other machine operations are taking place. My intentions were to have the text displayed while the music was being played. However, the write was so fast that the program was no more effective than the one shown here using PRINT statements ahead of the PLAY statements, the latter being in the foreground mode.

Problem 76: Whatever Will Be Will Be

Bridging the gap from Christmas carols that are hundreds of years old to the semi-modern Program 76 is known by two names, "Whatever Will Be, Will Be" and "Que Sera Sera." Rosemary Clooney had a hit with this song in the middle 1950s.

The program is run in *legato* mode at a moderate tempo of T150. This was an ideal tune to input in the *legato* mode, since it was not necessary to insert a string of P64 separation rests because of the note sequence. I prefer the mode of

```
10   REM "VAYA CON DIOS"
20   FOR X = 1 TO 2
30   PLAY "MF ML T100 O2C8D8E8D#8E8D#8E4E
8D#8E4.F8E4D2"
40   PLAY "D2P64D8E8F8E8F8E8F4F8E8F4.D8D#
4E2E2."
50   PLAY "O3C4P64C4.P64C8P64C4O2B4.A8P64
A4G2G2."
60   PLAY "O2P64G4P64G4.P64G8P64"
70   IF X = 2 THEN 100
80   PLAY "G4F4G4E2.E2"
90   NEXT X
100   PLAY "O2G4F4O1B4O2C2.C2P64C4"
110   PLAY "O2B-8O3C8O2B-8A8B-4B-8A8B-4.O
3C8"
120   PLAY "O2B-4A2A2C4G8A8G8F#8G8A8G2A4F
2.F2D4"
130   PLAY "O3C8D8C8O2B8O3C4C8O2B8O3C4.D8
C4O2B2B2"
140   PLAY "O2D4A8B8A8G#8A8G#8A2B4G4.F#8F
8E8D2"
150   PLAY "O2C8D8E8D#8E8D#8E4E8D#8E4.F8E
4D2"
160   PLAY "D2P64D8E8F8E8F8E8F4F8E8F4.D8D
#4E2E2."
170   PLAY "O3C4P64C4.P64C8P64C4O2B4.A8P6
4A4G2G2."
180   PLAY "O2P64G4P64G4.P64G8P64"
190   PLAY "O2G4F4O1B4O2C2.C2"
200   END
```

Program 77. "Vaya con Dios."

```
10   REM "WINDY"
20   FOR X = 1 TO 2
30   PLAY "MF MS T170 O2A4O3C8O2A8G4F4O3D
8D8C4O2A4O3C4O2A4"
40   PLAY "O3C8O2A8G4F4O3D8D8C4O2G2A4O3C8
O2A8G4F4"
50   PLAY "O3D8D8C4O2A4O3C4O2A4O3C8O2A8G4
F4G4F2.O1F4F4F2"
60   PLAY "ML O2F4F4A8O3C8F4D2O2B-4O3C4C4
O2F4A8O3C8F4"
70   PLAY "O3D2O2B-4G4G4E4F8A8O3F4D2O2B-4
O3C4C4O2A4"
80   PLAY "O2F64A8F64A4."
90   IF X = 2 THEN 130
100   PLAY "O2G4.O3F8F64F8F64F4.C4.D8F64D
8F64D4.C4.F64C8"
110   PLAY "D8F4.G1"
120   NEXT X
130   PLAY "O2G4.O3D8F64D8F64D4.F64D4.F64
D8F64D8F64D4."
140   PLAY "C4.D8F64D8F64D4.F2."
150   END
```

Program 78. "Windy."

legato programming for most songs because of the smooth output. The music normal mode often brings about an unwanted choppiness and does not differ in highly noticeable form from *staccato* mode programming.

This song has been performed at extremely fast and extremely slow tempos, so I chose a point somewhat in between. The PLAY statement information is included in lines 20 through 70. It is possible to arrange all of this information in two lines; my method was used for easy debugging purposes. The total range of notes spans only two octaves, so 02 and 03 statements are used throughout. The manuscript from which this program was written came from a student piano book. I incorporated a few subtle changes to make the piece sound better. Those persons who are heavily involved in composition and arranging may spend hours on a simple song to custom-tailer it. Admittedly, I did not take a lot of time, but I think you will find this program quite enjoyable and capable of bringing back memories associated with this song when it was popular in the 1950s.

Program 77: Vaya con Dios

Program 77 causes the AT&T personal computer to output what can certainly be classified as an old standard, "Vaya con Dios." This song precedes my birth, so I can't way when it reached its height of popularity. You can still hear this song occasionally on records and radio stations.

The song plays through the main theme twice and then enters the musical bridge to its conclusion. Therefore the main melody lines are committed to a FOR-NEXT loop in lines 20 and 90, respectively. The intro to the bridge is a modification of the last few bars of the initial verse, so it was necessary to include the detection line found in line 70 of the program. During the second cycle of the loop, execution will branch to line 100 after line 60 is completed. Line 100 contains the musical intro to the bridge, while lines 110 through 190 include the bridge notes themselves.

This program is run in *legato* mode. Therefore, you will notice an occasional P64 rest in several portions of the program. The outro is actually a repeat of the opening verse, so lines 30 through 60 are repeated in lines 130 through 180, until the final outro in line 190. It would have been possible to re-enter the loop at the desired point, but for the sake of understanding (both from the reader's and the author's standpoint), straight-line programming was used for the final portions of the song. This piece is normally performed at a medium-slow tempo, and the T100 tempo command produced the desired speed.

Program 78: Windy

Program 78 brings us up to the sixties and a big hit for the musical group known as The Association. The song is called "Windy" and is output by the computer in *staccato* mode programming, which closely simulates the original song.

The first portion of the song, the main verses, is committed to a FOR-NEXT loop which begins in line 20. The detection line at 90 exits the loop at the proper point to access the outro lines at 130 and 140. Since the *staccato* mode of programming was used, it is unnecessary to use P64 dividers in the first few lines. However, beginning at line 60 there is a switch to *legato* mode. This occurs during the middle of the first two verses. During the first verse play, line 120 recycles the loop and the *staccato* mode is again entered in line 30. When the final loop exit is made from line 90 to 130, the *legato* mode is maintained until the end of the piece. Therefore you will see several P64 dividers beginning below line 60.

The switch from *staccato* to *legato* programming mode was not mandatory, but allowed me to reproduce more accurately the song in its original style. If you are familiar with the tune, you already know what I am talking about. If not, you will find out as soon as you run the program. Musical switches from *staccato* to *legato* and back are quite common in many modern pieces, though not with older songs. The duration mode capabilities of the AT&T computer will greatly aid you in accurately reproducing many of these modern pieces. To really be able to hear the difference, input the program and run it as shown. When you have finished debugging, display line 60 and temporarily change the ML command to MS. When the modified program is run, you will be able to see the difference made by the mode change. You certainly will want to switch back to the ML command in line 60 after completing the test run.

Program 79: Everything Is Beautiful

A popular entertainer of the sixties and seventies, Ray Stevens, wrote and sang "Everything Is Beautiful," which was a big hit around 1970. This song reminds me of a small town in Virginia, and of an even smaller AM radio station where I was employed when this song was popular.

This is an unusual song in that it begins with children singing "Jesus Loves the Little Children." When that has been completed, the main melody line of Steven's composition begins. The melody line is then repeated twice before entering the musical bridge. Line 20 begins the intro part and contains the tempo and mode designations for the remainder of the song. The intro is completed in line 50. At this point the main melody line begins. Since this is to be played twice, a FOR-NEXT loop is established in line 60, with the main melody note information found in lines 80 through 120. During the second cycle of the loop, line 110 creates a branch to line 140 where the second melody strain begins. When this portion is exited at the conclusion of line 210, the original melody line is repeated in lines 220 through 260. It would have been possible to re-enter the original loop, but it was much easier to renumber original lines 70 through 100 with 220 through 250. Line 260 plays the final outro to this number.

This program is longer than most of those presented thus far, because it includes three separate melody sections. By efficiently utilizing an additional FOR-NEXT loop the program could be shortened to about twenty lines; this would result in little savings of time for programming and for input and would probably add to the debugging.

The basic program is executed in *legato* mode. However, I have often switched to music normal mode rather than include a larger number of P64 separation rests between identical notes. The first switch occurs toward the end of line 20, where a series of G notes must be produced. The switch back to *legato* mode occurs near the center of line 30. You will see instances where I elected to use P64 separators rather than switch modes. In these cases the identical note strings were short, and I felt this form of separation was more advantageous than switching to the music normal mode. You may elect to do otherwise, but the program will run quite satisfactorily as shown here.

Program 80: Snowbird

Program 80 outputs another song which was popular around 1970. As I recall, Anne Murray had a big hit with "Snowbird." As with most popular songs, a main verse repeats at least twice, with an exit near the end of the second repetition playing to an appropriate outro. The FOR-NEXT loop established at lines 20 and 100 contains the main verse data. The detection portion in line 80 exits the loop at a specific point in the second run by branching to lines 110 and 120, where the outro data is located.

```
10  REM "EVERYTHING IS BEAUTIFUL" (BY RA
Y STEVENS)
20  PLAY "MF ML T120 O2A4P64A4P64A8.G16F
8.D16C2F2MNG8.G16G8.G16"
30  PLAY "O2A8.G16F8.A16MLG2.C8.O1B-16A8
.O2C16F8.G16F4P64"
40  PLAY "O2F8.E16D8.E16F8.D16C4A8.G16F8
.E16F8.D16E8.C16E8.G16"
50  PLAY "F1F4P4T160A8O3C8P64C8O2A8O3"
60  FOR X = 1 TO 2
70  PLAY "O3D4.C8P64C2C2"
80  PLAY "O2A8B-8A4P64A8G8G2.G2"
90  PLAY "O2D8E8F8G8A4.G8P64G4O3C4D4C8P6
4C8"
100  PLAY "O2A8G8F8G8"
110  IF X = 2 THEN 140
120  PLAY "O2A1P2A8O3C8P64C8O2A8"
130  NEXT X
140  PLAY "O2D8C8D8F2.F2P2P4A8P64A8B-4A4
"
150  PLAY "O2P64A4G2C4D4F4G8A4.G8F8F2.P4
"
160  PLAY "O3C8P64C8P64C4P64C8P64C8P64C8
O2A8G2C8P64C8D4F4G4.A8G8F8F2."
170  PLAY "O3P8C8F8P64F8P64F2P8C8E8P64E8
D4C4"
180  PLAY "O2P8F8O3D8C8D8C8O2A8MNC8C8C8"

190  PLAY "MLO2A8G8F2.P8F8O3C8O2A8O3C8P6
4C8D4"
200  PLAY "O2P8A8O3C8O2A8P64A4MNC8C8MLD8
F4.P8F8G8A8"
210  PLAY "O2G8F4.F2P2A8O3C8P64C8O2A8"
220  PLAY "O3D4.C8P64C2C2"
230  PLAY "O2A8B-8A4P64A8G8G2.G2"
240  PLAY "O2D8E8F8G8A4.G8P64G4O3C4D4C8P
64C8"
250  PLAY "O2A8G8F8G8"
260  PLAY "O2D8C8D8F2.F2"
270  END
```

Program 79. "Everything Is Beautiful."

```
10  REM "SNOWBIRD"
20  FOR X = 1 TO 2
30  PLAY "MF ML T150 O3F4P64F4E4P64E4D4P
64D4C4P64C4O2F4G1"
40  PLAY "O2G2.O3C8P64C8P64C4O2B-4P64B-4
A4P64A4G4P64G4A4B-4"
50  PLAY "O3C4O2A4G4C1C2.P8"
60  PLAY "O3F8P64F4E4P64E4D4P64D4C4P64C4
O2F4G1G2."
70  PLAY "O3C8P64C8P64C4O2B-4P64B-4A4P64
A4G4P64G4A4B-4P64B-4A4G4"
80  IF X = 2 THEN 110
90  PLAY "O2F1F2."
100  NEXT X
110  PLAY "O2F1F2O3E8D8C4MNE8E4.E4D4C4C4
O2B-4O3C4MLD1D1"
120  PLAY "P4MNF4F4F4MLF1F1F2."
130  END
```

Program 80. "Snowbird."

The original version of this song was treated very delicately by Ms. Murray and included smooth transitions from note to note at a fairly rapid tempo. A T150 command was an appropriate tempo; it was necessary to use the *legato* play mode to bring about smooth transitions. As before, I used a combination of P64 rest separators between identical notes and switched to music normal mode for the longer, identical note lines. The last notes of the outro are all run in *legato* mode. In this program I relied heavily on P64 separators and not much on mode transitions.

Program 81: Wipeout

In the middle sixties I surprised my family and especially my band director by becoming very involved in the rock and roll scene, which was given new life by the Beatles. Their popularity generated hundreds of other rock groups, who enjoyed varying degrees of success. Most groups relied heavily on vocal ability and not much on instrumental backup. An exception was the Surfaris, who were famous for their guitar instrumentals rather than their vocals. Their most popular hit was called "Wipeout" and is still performed by bands today.

"Wipeout" was built mostly from *staccato* eighth notes at a very fast tempo. Program 81 presents the AT&T PC computer version performed mainly in *staccato* mode at a rapid tempo of T200. In musical terms this tempo would be described as *presto* or *prestissimo*.

This program is straight line; lines 90 through 150 mostly repeat lines 20 through 80. Programming time was not greatly lengthened by using this method, because after I completed line 80 I returned to line 20 and renumbered it 90. Line 30 became line 100, and so on. I did this after the first eight lines had been debugged. Therefore, I was certain that the "copy lines" would be accurate. Line 150 is the outro and differs greatly from line 80, which would be its match if the copied line sequence were maintained throughout.

I have made a few transitions from *staccato* mode to *legato*, as shown in lines 80 and 150. It was only at these two points that sustained *legato* mode was necessary. The switch from *staccato* to *legato* in line 80 is reversed at the beginning of line 90, which repeats line 20. The ending note in line 150 is sustained in *legato* mode.

```
10 REM "WIPEOUT" BY THE SURFARIS
20 PLAY"MF MS T200 O2G8B-8B8O3C8C8C8O2B-
8G8G8B-8B8"
30 PLAY"O3 C8C8C8O2B-8G8G8B-8B8O3C8C8C8O
2B-8G8G8B-8B8"
40 PLAY"O3C8C8C8O2B-8G8O3C8E-8E8F8F8F8E-
8C8C8E-8E8"
50 PLAY"O3F8F8F8E-8C8O2G8B-8B8O3C8C8C8O2
B-8G8G8B-8B8"
60 PLAY"O3C8C8C8O2B-8G8O3C8E8F8G8G8G8F8D
8D8G8G8"
70 PLAY"O3F8F8F8E-8C8O2G8B-8B8O3C8C8C8O2
B-8G8G8O3C8O2B-8"
80 PLAY"O3C8C8C8MLC8C8"
90 PLAY"MF MS T200 O2G8B-8B8O3C8C8C8O2B-
8G8G8B-8B8"
100 PLAY"O3 C8C8C8O2B-8G8G8B-8B8O3C8C8C8
O2B-8G8G8B-8B8"
110 PLAY"O3C8C8C8O2B-8G8O3C8E-8E8F8F8F8E
-8C8C8E-8E8"
120 PLAY"O3F8F8F8E-8C8O2G8B-8B8O3C8C8C8O
2B-8G8G8B-8B8"
130 PLAY"O3C8C8C8O2B-8G8O3C8E8F8G8G8G8F8
D8D8G8G8"
140 PLAY"O3F8F8F8E-8C8O2G8B-8B8O3C8C8C8O
2B-8G8G8O3C8O2B-8"
150 PLAY"O3C16P16O2B-8G8MLO3C1"
160 END
```

Program 81. "Wipeout."

This type of program is ideally suited for the AT&T PC. Each note is distinct, although the tempo is quite rapid. This is the type of program ideal to run during a programming pause, after long hours behind the computer have made you sleepy. The output from this program should pep you up and allow a few more hours of programming.

Program 82: Blues in F

Program 82 is entitled "Blues in F" and will slow things down after the panic strains of the previous program. This piece is fairly old and was designed to be played at a slower tempo than that keyed by the T100 command in line 20. I originally tried T50, but it was too slow to be enjoyable on the computer.

The program is run entirely in *legato* mode and is input without the need for P64 dividers because of the note transitions. In some cases I lengthened or shortened a note and made up for the difference with a longer duration rest, but each of the bars should come out mathematically correct. In some instances I had difficulty with the math, so you will see a P64 or two, but these are used very rarely in this program.

```
10 REM "BLUES IN F"
20 PLAY"MF ML T100 O3 C1C8O2F8A8B-8O3C4O
2B-8O3C8O2A2."
30 PLAY"O2F8E8E-2.F8O3F8P64F1F8O2B-8O3D8
E8F4D4C2.D8C8"
40 PLAY"O2A2F8F8A8O3C8E8C4.C8O2E8G8O3C8D
8O2B-4.B-8"
50 PLAY"O2D8F8B-8O3C8O2F4.F2P1"
60 PLAY"O2A2O3C2C2.O2B-8O3C8O2A2.F8E8E-2
F8O3F8"
70 PLAY"O3D2F2F2.D4O2A2O3C2C2P8O2F8A8O3C
8E8C4.C8"
80 PLAY"O2E8G8O3C8D8O2B-4.B-8D8F8B-8O3C8
O2F4."
90 PLAY"O3C8O2F8G8F8A2"
100 END
```

Program 82. "Blues in F."

For programming this blues number the *legato* mode was mandatory to cause the "lazy" sliding effect common to these types of pieces. Because of the single melody line and the note transition, there was no point within this piece where a FOR-NEXT loop would have been practical. Therefore straight-line programming was used throughout. Since the program plays through only once, this might be an ideal situation to introduce an endless loop by replacing the END statement in line 100 with a GOTO 20 statement.

Program 83: Somethin' Stupid

During the last half of the sixties, Frank Sinatra's daughter Nancy became an exceedingly popular entertainer in her own right. Many of her songs did not receive favorable reviews, but a few are still heard in smooth instrumental form today. One of these, "Somethin' Stupid," is presented in Program 83. I never liked this song, but I'm probably in the minority.

You will see many transitions from music normal to music *legato* mode. The note structure is exceedingly simple (from musical standpoint) and involves the sequential playing of a series of identical quarter and/or eighth notes. I switched to music *legato* whenever possible, but many sections required the normal mode to avoid destroying measure timing. A good example of this is shown in line 50, where four Fs played as eighth notes are followed by four Gs played as eighth notes and four eighth note As. If I had inserted P64 between each one, I would eventually have an extra measure at the end. Running this sequence in music normal mode causes the same effect at the audio output end, greatly shortens programming time, and keeps me mathematically correct with the original manuscript. The music *legato* mode was used mainly to play certain identical notes over several sustained measures.

For Nancy Sinatra fans, please don't become angry over my remarks about this particular song. I liked her version of "You Only Live Twice," the theme song for the James Bond movie of the same name. Unfortunately I couldn't find a manuscript for this song, so it's not included here.

Program 84: King of the Road

The biggest hit for a country music entertainer named Roger Miller, who made the transition to the pop charts, was "King of the Road." Program 84 offers an especially nice run on the AT&T PC.

This program combines straight-line programming with a FOR-NEXT loop at the beginning. Some of the main melody lines are repeated in the latter portion of the program.

Line 110 allows the FOR-NEXT loop to be exited at the proper point. The bridge lines are contained in program lines 140 through 180. The main melody appears in straight-line programming form again beginning at line 190 and executing through line 270, which contains the outro.

You will see a lot of transitions from music normal to music *legato* modes and then back. This allows the song to be reproduced in much the same manner as the original, which included some choppy note sections, along with those which offered smoother transitions. I had originally programmed for a faster tempo, but later decided to slow it to a moderate speed with a T150 command. I also tried varying the tempo at different sections, but the final version shown here produces the best effect (in my opinion).

Whenever possible I have tried to program the various songs presented in this chapter close to their original style. There is much to be said for

```
10 REM "SOMETHIN' STUPID"
20 PLAY"MF MN T160 O1 A8O2C8C8C8C8D8D8D8
D8E8E8E8E8"
30 PLAY"O2F8F8E8C8D8D8F4MLE2E2.MNF8C8C8C
8C8C8D8D8D8D8"
40 PLAY"O2 E8E8E8E8F8F8E8C8D8D8E4MLD2D2.
MNF8C8"
50 PLAY"O2 F8F8F8F8G8G8G8G8A8A8A8A8O3C8O
2B-8A8B-8"
60 PLAY"O2A4G8MLG8G2G2.MNF8D8F8F8F8D8E8E
8E8D8"
70 PLAY"O2F8F8F8D8E8E8E8A8G2MLF2MNF2.F8F
8"
80 PLAY"O2F8F8F8F8G8G8G8G8A8A8A8A8O3C8O2
B-8A8B-8"
90 PLAY"O2A8G8MLG2.G2MNF8F8G8G8G8G8A8A8A
8A8B8B8B8B8"
100 PLAY"O3D8D8D8E8D8C8MLC2.C2.MN"
110 PLAY"MF MN T160 O1 A8O2C8C8C8C8D8D8D
8D8E8E8E8E8"
120 PLAY"O2F8F8E8C8D8D8F4MLE2E2.MNF8C8C8
C8C8C8D8D8D8D8"
130 PLAY"O2 E8E8E8E8F8F8E8C8D8D8E4MLD2D2
.MNF8C8"
140 PLAY"O2 F8F8F8F8G8G8G8G8A8A8A8A8O3C8
O2B-8A8B-8"
150 PLAY"O2A4G8MLG8G2G2.MNF8D8F8F8F8D8E8
E8E8D8"
160 PLAY"O2F8F8F8D8E8E8E8A8G2MLF2MNF2."
170 END
```

Program 83. "Somethin' Stupid."

```
10   REM "KING OF THE ROAD"
20   FOR X = 1 TO 2
30   PLAY "MF MN T150 O3P4C4O2G4E4D4E8MLF
8F2P4B4MNB8G8A8G8C4"
40   PLAY "O2C8MLC8C2MNP4O3C4O2G4E4"
50   PLAY "O2D4E8MLF8F2MNP8G8G4B4O3C4"
60   PLAY "O3D4C8MLO2B8B8MNG8A4P4O3C4O2G4
E4"
70   PLAY "O2D4E8MLF8MNF4F8D8P4MLB4B8MN"
80   PLAY "O2G8A8G8C4C8MLC8C8MNE8G4"
90   PLAY "O3P4E2D4MNC4C4C8O2A4.P2"
100   PLAY "O3D8C8O2B4"
110   IF X = 2 THEN 140
120   PLAY "O3C1"
130   NEXT X
140   PLAY "O3C2MNP8C8C4C8C8C4C8C8C8O2B8"

150   PLAY "O2A8O3C8O2A4F4P4B8B8B4A8G4G8"

160   PLAY "O2A8O3C8O2A4G4P8G8O3G8G8E4D8C
4"
170   PLAY "O2A8O3C8O2A8O3C4O2A4P8A8G8G8G
8G8"
180   PLAY "O2B8B4B8O3D8D8D8MLF8MNF8E8D4"

190   PLAY "O3P4C4O2G4E4D4E8MLF8F2P4B4MNB
8G8A8G8C4"
200   PLAY "O2C8MLC8C2MNP4O3C4O2G4E4"
210   PLAY "O2D4E8MLF8F2MNP8G8G4B4O3C4"
220   PLAY "O3D4C8MLO2B8B8MNG8A4P4O3C4O2G
4E4"
230   PLAY "O2D4E8MLF8MNF4F8D8P4MLB4B8MN"

240   PLAY "O2G8A8G8C4C8MLC8C8MNE8G4"
250   PLAY "O3P4E2D4MNC4C4C8O2A4.P2"
260   PLAY "O3D8C8O2B4"
270   PLAY "O3C2."
280   END
```

Program 84. "King of the Road."

taking a popular song and creating a special arrangement designed to take advantage of all the computer has to offer. The computer offers much less than a full orchestra, so certain changes are warranted to create the best run possible using the limited musical abilities of the microcomputer. It may sound like I am contradicting a previous statement alluding to the excellent musical capabilities of the AT&T PC. However, the statement was in relation to the musical capabilities of other computers, which are sometimes nonexistent. The musical output of the AT&T PC is produced by a flexible, single-tone generator. The single tone output is a major disadvantage. The computer may be interfaced with an electronic organ or a synthesizer to control hundreds of different tone generators (offering simultaneous output), to form a highly versatile and efficient musical instrument. This book concentrates on the abilities of the computer alone and does not take into account the unlimited possibilities of the musical peripherals that are available.

Program 85: Jean

"Jean," written by Rod McKuen, was another bit hit at the beginning of the seventies and is shown as Program 85. This piece was performed very delicately and at a moderate tempo; hence, the *legato* programming mode and the T120 tempo. The T120 command is unnecessary, since this is the default tempo. If the tempo command is not included, the speed will still be the same. The main melody lines are committed to a FOR-NEXT loop in lines 20 and 80, respectively. By now you have probably noticed a "rhythm" (pardon the pun) to my musical programs, in that there is an exit from the loop during the second cycle, which enters the bridge lines found in program lines 90 through 120. The outro is a repeat of the main melody lines and begins at line 130. Lines 130 through 150 are copies of lines 30 through 50 at the beginning of the program. Line 160 gives the final outro notes.

```
10   REM "JEAN" (BY ROD MCKUEN)
20   FOR X = 1 TO 2
30   PLAY "MF ML T120 O2F2.O3C2.D4.C8O2B-
4O3C2.P2MNO2F8F8"
40   PLAY "MLO2F4.D8F4D4C2C2A8O3C8D2C8O2A
8O3C2O2A8F8"
50   PLAY "O2G4F4.D8F2.P4P8G8A8F8P64F8P64
"
60   IF X = 2 THEN 90
70   PLAY "MLF4P64F2P64F2.P1"
80   NEXT X
90   PLAY "MLO2G2F8G8F2.F4P4O3MNF8F8ML"
100   PLAY "MLO3D2P64D8E8F4E4.D8E2C4P64C2
MNF8F8MLD2P64D8E8F4E4D8"
110   PLAY "MLO3E2C8O2A8O3C2E8F8D2P64D8E8
"
120   PLAY "MLO3F4E2P4P8C8D8C8O2A-4G2F4P6
4"
130   PLAY "O2F2.O3C2.D4.C8O2B-4O3C2.P2MN
O2F8F8"
140   PLAY "MLO2F4.D8F4D4C2C2A8O3C8D2C8O2
A8O3C2O2A8F8"
150   PLAY "O2G4F4.D8F2.P4P8G8A8F8P64F8P6
4"
160   PLAY "MLO2G2F8G8F2.F2"
170   END
```

Program 85. "Jean."

You will also notice that I have switched between *legato* and normal modes of programming, and have combined these transitions with P64 separation commands when it was not convenient to exit the *legato* programming mode. The P64 separators are used only at program positions where the short, extra bar duration cannot be practically detected.

Program 86: Five Hundred Miles

"Five Hundred Miles" was a big hit for the folk group, Peter, Paul and Mary, in the late sixties. Construction-wise, this program is very similar to many others which repeat the opening verse and then move into a bridge and an outro. The same pattern is used here as with similar programs and includes an opening FOR-NEXT loop. You will notice a separate PLAY line in program line 15. This is the intro to the opening melody line and is not repeated during the second repetition of the loop. The opening is placed outside of the loop and plays the two quarter notes that form the intro before entering the loop. There is an intro to the main melody line which is different from the first; it begins the second playing, which is composed of two quarter note Gs, found at the end of line 70. When the loop repeats, line 60 branches to line 90, which enters the bridge strains. In modern songs you will often find a one-time intro to the main melody. In these cases it is desirable to keep these few notes outside of your major melody loop.

Program 87: Supercalifragilisticexpialidocious

One of the major musical hits of the sixties was the movie *Mary Poppins*. One of the many musical numbers performed in this movie was "Supercalifragilisticexpialidocious." This was termed a comical, nonsense song and was quite popular among children as well as adults.

Program 87 reproduces this song using *staccato* mode programming throughout. The major verse

```
10   REM "FIVE HUNDRED MILES"
15   PLAY "MF ML T100 O2D4G4"
20   FOR X = 1 TO 2
30   PLAY "O2B4.P64B8A8G4.B2A4G4"
40   PLAY "O2A4.B8A4G4E2.P64E8G8A4.B8A8G4
."
50   PLAY "E8D4.D8P64D8E8G8P64"
60    IF X = 2 THEN 90
70   PLAY "G1MNG4:G8G4G4ML"
80   NEXT X
90   PLAY "MLO2G1MNG2G4G4.ML"
100   PLAY "MLO2B2A8G4.B2A4G4A4.B8A4G4E2.
P64E8G8A8B4.A8G4."
110   PLAY "O2E8D4.D8P64D8E8G8P64G1MNG4.G
8G4G4"
120   PLAY "MLO2B4.P64B8A4G4B4.P64B8A4G4A
4.B8A8G4."
130   PLAY "O2E2.P64E8G8A8B4.A8G4.E8D4.D8
P64D8E8G8P64G1G2"
140   PLAY "P64MNO2G4G4O3MLE4.D8C4O2B4A4.
G8E4G4G1G2."
150   END
```

Program 86. "Five Hundred Miles."

```
10  REM "SUPERCALIFRAGILISTICEXPIALIDOCI
OUS"
20  FOR X = 1 TO 4
30  PLAY "MF MS T255 O2L4EGGGAGGEGGAGG2F
2"
40  PLAY "O2L4GGGGAGGDGGAGG2E2L4GGGGAGGG
"
50  PLAY "O3L4CCDCC2O2A2L4AO3CO2BAL4O3CO
2GGE"
60  IF X = 4 THEN 150
70  PLAY "O2L4GG#ABO3C2C4P4O2G4L8GGGGGGG
4G8G8G4P4"
80  PLAY "O1G4L8GGGGGGG4G8G8G4"
90  PLAY "O2L4GEGGGAGGEGGAGF2.G4"
100  PLAY "O2L4GGGGAGGDGGAGE2.L4GGGGG"
110  PLAY "O2L4AGGGO2CCDCA2.A4"
120  PLAY "O2L4AAAABAAAO3DCO2BA"
130  PLAY "O2G4P4F2"
140  NEXT X
150  PLAY "O2L4GG#ABO3C2C4":END
```

Program 87. "Supercalifragilisticexpialidocious."

is repeated four times before entering the outro section. Therefore the FOR-NEXT loop established around the melody line segment cycles four times before an exit occurs in line 60, branching to line 150.

The *staccato* programming mode was ideal for this song, as each note was to be short and choppy. You will also notice that maximum machine tempo (T255) is used to play the notes at the fastest possible pace. Because of the *staccato* mode programming, each note is clearly defined. *Legato* mode programming would be very difficult here, since there are many sequences which repeat the same note up to seven times. At these speeds a P64 separator would throw off the timing to a very noticeable degree.

You may wish to slow the tempo; this will probably be mandatory during the debugging procedure. This piece should be executed very rapidly, so don't slow the final version too much. The notes in this song span three octaves. You will see designations in octave commands of O1 to O3. The broad range of note frequencies makes this song especially interesting when played at a rapid tempo on the AT&T PC.

Program 88: Theme from *Love Story*

Americans fell in love with tragedy, based partially upon the hit movie *Love Story*, starring Ryan O'Neil and Ali McGraw. The haunting theme from this movie still endures and is still performed by many orchestras.

Program 88 reproduces this song using fairly slow tempo and a *legato* format. P64 separators are used in conjunction with mode changes in music normal for long sequences of identical notes. At the slower tempo, the P64 separators and the measure imbalance they cause are not only audibly detectable.

After the main melody line is played through twice, there is a branch to the bridge lines beginning at line 100. When the bridge is complete (line 160), the main melody line is again entered, and this carries through to the outro. Lines 170

```
10  REM "THEME FROM LOVE STORY"
20  FOR X = 1 TO 2
30  PLAY "MF ML T100 O3C8O2E8P64E8O3C8P6
4C2C8O2E8"
40  PLAY "O2P64E8O3C8P64C8O2E8F8E8MND8D8
D8B8MLB2B8"
50  PLAY "O2D8P64D8B8P64B8D8E8D8MNC8C8C8
A8A2P8C8C8A8"
60  PLAY "MLA8C8D8C8MNO1B8B8B8O2G#8MLG#2
G#4"
70   IF X = 2 THEN 100
80  PLAY "O2A4B4F4E1E2.P4"
90  NEXT X
100  PLAY "O2A4B4G#4O3C#1P4D4E4O2A4"
110  PLAY "MLO3F2F8O2A8O3F8O2A8P64"
120  PLAY "O2A8B8P64B4B8O3D8F8D8E2E8O2G8
O3E8O2G8P64G8A8"
130  PLAY "O2P64A4A8O3C8E8C8D2D8O2B8O3D8
O2B8O3C4.D8"
140  PLAY "O3E8O2A8O3C8E8F2F8O2G8A8O3C8"

150  PLAY "O3P64C4O2B4B8O3C8D8O2F8E2E8P6
4E8F8G8"
160  PLAY "O2B4A4A8P64A8G12F12E12D#2D#8F
#8A8F#8G#1G#1"
170  PLAY "MF ML T100 O3C8O2E8P64E8O3C8P
64C2C8O2E8"
180  PLAY "O2P64E8O3C8P64C8O2E8F8E8MND8D
8D8B8MLB2B8"
190  PLAY "O2D8P64D8B8P64B8D8E8D8MNC8C8C
8A8A2P8C8C8A8"
200  PLAY "MLA8C8D8C8MNO1B8B8B8O2G#8MLG#
2G#4"
210  PLAY "O2A4B4G#4A1A1"
220  END
```

Program 88. "Theme from *Love Story.*"

through 200 are repeats of lines 30 through 60. Line 210 plays the notes which form the final outro.

Program 89: Tijuana Taxi

Herb Alpert and the Tijuana Brass were another hit instrumental group in the sixties and early seventies when vocal groups were usually considered king. Alpert's group combined the sounds of a mariachi band with the style of jazz ensembles. They had many hit songs, one of which was "Tijuana Taxi," produced by Program 89. This is a very fast number and is played at T225 tempo. Most notes are played in *staccato* mode, although there is an occasional change to *legato* mode to create the needed slurs indicated in the musical manuscript. This is straight-line programming all the way. The music runs so rapidly that

```
10  REM "TIJUANA TAXI"
20  PLAY "MF MS T225 O2A8O3C4F4E4D8C4.O2
B-2P8G8B-4O3E4D4C8O2B-4.A2"
30  PLAY "O2P8F8A4O3D4C4O2B-8A4.G2P8G8A4
O3C8C8C4P2"
40  PLAY "O2P4B-4A4G4MLF1F4MSP4P8A8O3C4"

50  PLAY "O3F4E4D8C4.O2B-2P8G8B-4O3E4D4C
8O2B-4."
60  PLAY "O2A2P8F8A4O3D4C4O2B-8A4.G2P8G8
A4O3C8C8C4P2"
70  PLAY "O2B-8B-8B-4P2A8A8A4P2G8G8G4P2F
8P8O1C8MLD8D8MSC8O0A4F4P4O3F8"
80  END
```

Program 89. "Tijuana Taxi."

you may wish to insert a FOR-NEXT loop beginning at line 15, with the NEXT statement at line 75. If the loop counts from 1 to 2, then the entire song will be played twice, but you can increase the upper end of the loop to play it through any number of times. Alternately, you can leave the program as is, but replace the END statement in line 80 with a "GOTO 20." This will form an endless loop, and the song will play continuously until manually halted at the keyboard.

Program 90: Spanish Flea

Another of Alpert's big hits (possibly his biggest) was "Spanish Flea," and is run on the AT&T PC using Program 90. This runs in music normal mode; there is an intro line (line 20) which is placed before the beginning of the FOR-NEXT loop, which contains the main melody information. The intro notes are played and the FOR-NEXT loop is then entered. During the second go-round there is an exit at line 80, which branches to line 1000.

```
10  REM "SPANISH FLEA"
20  PLAY  "MF MN T150 O2D8E-8E8F4O3D4D8C
4."
30  FOR X = 1 TO 2
40  PLAY "O2B2P8G8G-8F8E4O3C4C8O2B-4."
50  PLAY "O2A2P8F8E8E-8D8F8B-8MLG8MNG8B-
8O3C4"
60  PLAY "O2F8A-8O3D-8O2MLB-8MNB-8O3D-8E
-4MLF1MNF4"
70  PLAY "O3MNF8F8G8F8D-8C8"
80  IF X = 2 THEN 1000
90  PLAY "O2MNB-4O3D4D8C4."
100  NEXT X
1000  PLAY "O2MNB-4F8F8P4F8F8P4F4P4F4P4F
8F8P4F8F8P4F4B-8"
1010  END
```

Program 90. "Spanish Flea."

This line contains the final outro information.
line contains the final outro information.

I slowed this piece quite a bit. The T150 tempo should accurately reflect the playing of the original version of the song. Other versions done by different orchestras have been played at different tempos, and depending on which version you like, you may or may not wish to modify the tempo command found in line 20.

Program 91: Cotton Candy

Another trumpet player who enjoyed popularity to the mid-sixties was "Mr. Mardi Gras," Al Hirt. Whereas Herb Alpert brought a mariachi/jazz blend to the American public, Al Hirt gave us modified Basin Street sounds. One of his hits was "Cotton Candy," produced by Program 91. The song is run in *staccato* mode at maximum machine tempo (T255).

Musically the tempo may be classified as *prestissimo*. This arrangement involves a number of sharps; be careful you make note of each when inputting the program to your machine. I have used the standard musical sharp symbol throughout these programs (#); you may also use a plus sign (+) to indicate a sharp.

When debugging this program it might be best to use a test tempo of 155, since there are so many notes. Once you have debugged each line, you can revert to the faster speed.

Program 92: Alley Cat

Most readers will remember the popular song "Alley Cat," which was originally played as a solo piano instrumental. The version in Program 92 is a solo computer instrumental. This song moves rapidly with a T220 tempo command. Music normal mode is used throughout; the intro line (line 20) lies outside of the main melody line. The latter is contained in the FOR-NEXT loop between lines 30 and 80. Lines 100 and 110 form the musical bridge between the intro melody and the outro melody, which begins in line 120. Lines 120 and 130 are identical to lines 40 and 50. Line 140 is the final

```
10   REM "COTTON CANDY"
20   FOR X = 1 TO 2
30   PLAY "MF MS T255 O2G4P8G8G8E8G4A4P8F
8A4O3C4"
40   PLAY "O2G4P8G8G8E8G4A4P8F8A8A8O3C4"
50   PLAY "O2G4G8E8G8E8G4A4P8A8B8A8B4A4P8
A8B8A8B4"
60   PLAY "O2O2A4G2.
70   NEXT X
80   PLAY "O2A4P8A8L8G#G#G#G#A4P8A8G#2A4P
8A8L8G#G#G#G#F#1"
90   PLAY "O2A4P8A8L8G#G#G#G#A4P8A8G#2"
100   PLAY "O2A4P8A8L8G#G#G#G#F#2.F#4G#1"

110   PLAY "O2B4G#4G#4G#4"
120   PLAY "O3C1C4O2F8F8A8A8O3C4O2B4E8E8G
8G8B4"
130   PLAY "O2A4D8E8F8G8A8B8O3C2."
140   END
```

Program 91. "Cotton Candy."

```
10   REM "ALLEY CAT"
20   PLAY "MF MN T220 O2D#8E12G12A12"
30   FOR X = 1 TO 2
40   PLAY "O3L4CO2BAA-GG#AP4G8G8G#4A4A#4B
2.P4"
50   PLAY "O3C4O2B4A4A-4G4G#4A4P4G8G8G#4A
4B4"
60   IF X = 2 THEN 90
70   PLAY "O3C2P8O2D#8E12G12A12"
80   NEXT X
90   PLAY "O3C2P4C4D4D2O2C#4D2.O3C4"
100   PLAY "O3D4D2O2C#4D2.O3C4D4D2O2C#4D2
."
110   PLAY "O3C4D4C4O2B4A4G4G-4F8D#8E12G1
2A12"
120   PLAY "O3L4CO2BAA-GG#AP4G8G8G#4A4A#4
B2.P4"
130   PLAY "O3C4O2B4A4A-4G4G#4A4P4G8G8G#4
A4B4"
140   PLAY "O3C2P4O2E8C8D8D8P4D#8D#8P4E4G
4B4.A8O3E4C8O2A8O3E-4D4C2"
150   END
```

Program 92. "Alley Cat."

outro. You may also wish to run this program in staccato mode. It will be completely different from the original version, but at this speed it becomes a very novel arrangement. You can also convert to *legato* mode programming, but it will be necessary to insert some P64 separators or switch to the music normal mode for identical note progressions.

I originally used a much lower speed with this program. The original song was played *moderato*; this was not too pleasing to the ear in the computer version, so I increased the speed by 50 percent.

Program 93: It Was a Very Good Year

Frank Sinatra fans, take heart! Program 93 will play "It Was A Very Good Year," a big hit in the late sixties. This song followed Frank Sinatra's hit, "That's Life," by one year.

The program runs in *legato* mode, although there are numerous changes to music normal mode to accommodate the note spacing. The tempo chosen is faster than the version Sinatra sang, but this produces the most pleasing output from the computer.

The main melody line is in two nearly identical parts. The second portion is nearly a repeat of the first, but there are subtle differences. I originally thought I could use a FOR-NEXT loop that would count from 1 to 4; because of the differences in the second portion of the first melody line, I had to straight-line program the entire sequence. When these two similar melodies are coupled together, they are contained within a FOR-NEXT loop which counts from 1 to 2. This takes you from age 17 to the "Autumn of My Life." The exit line (IF-THEN statements) is at 190. This causes a branch to the final outro line in 220. Some readers may wish to slow the tempo marginally to 90 or 100. I feel the default tempo of T120 produces the best musical effect.

```
10  REM "IT WAS A VERY GOOD YEAR"
20  PLAY "MF ML T120 O1B8O2C#8D#8"
30  FOR X = 1 TO 2
40  PLAY "O2E4B4P64B2B2P8G8A8B8"
50  PLAY "O2MNA8A8F4MLF2F2P8F8G8F8"
60  PLAY "O2MNE8E8B4MLB4.O3C#8D4O2B4"
70  PLAY "O2P64B4.O3D8O2B4MNA8A8MLA2A2"
80  PLAY "O3P4C4O2B4MNG#8G#8MLG#2G#2.MNB
8B8"
90  PLAY "O2MLA4F#4P64F#2F#2P8F#8G#8A8G#
4E4P64E2E2"
100  PLAY "O2MLF#4E8F#8O1B4P4O2F#4E8F#8G
8F#8E4F#4"
110  PLAY "O2MLE8F#8D4P64D4D#2P8O1B8O2C#
8D#8"
120  PLAY "O2E4B4P64B2B2P8G8A8B8"
130  PLAY "O2MNA8A8F4MLF2F2P8F8G8F8"
140  PLAY "O2MNE8E8B4MLB4.O3C#8D4O2B4"
150  PLAY "O2P64B4.O3D8O2B4MNA8A8MLA2A2"

160  PLAY "O3P4C4O2B4MNG#8G#8MLG#2G#2.MN
B8B8"
170  PLAY "O2MLA4F#4P64F#2F#2P8F#8G#8A8G
#4E4P64E2E2"
180  PLAY "O2MLF#4E8F#8O1B4P4O2F#4E8F#8G
8F#8E4F#4"
190  IF X = 2 THEN 220
200  PLAY "O2MLE8F#8D4P64D4D#2P8O1B8O2C#
8D#8"
210  NEXT X
220  PLAY "O2MLE8F#8D4P64D4D#1"
230  END
```

Program 93. "It Was A Very Good Year."

Program 94: I Left My Heart in San Francisco

While Frank Sinatra may rate first as the most popular male vocalist of all time, Tony Bennett has to rate a close second—and to many might even end up tied for first.

Program 94 produces one of his biggest hits, "I Left My Heart in San Francisco."

The program is a shortened version of the song, since it only plays through one complete chorus. This is an unusually simple program and uses straight-line techniques throughout, with no FOR-NEXT loops. I was able to slow the tempo to T100, which closely matches the tempo in the original song. I was not delighted with the musical manuscript used to program this song, so the better musicians among my readers may want to "dot a few eighths" to match the original version more closely.

You may perhaps use music normal mode, but the *legato* programming mode used is the most pleasing. At no point was it necessary to switch to music normal mode, as P64 statements were used

```
10  REM "I LEFT MY HEART IN SAN FRANCISC
O"
20  PLAY "MF ML T100 O2D4E-4G4F1F4G4A4B-
4"
30  PLAY "O2G4C2.C4P64C4O1B4O2C4G1G4B-4A
4.F8D1D8P8D4E-4E4"
40  PLAY "O2F8E-8D8E-8F2F2.G4"
50  PLAY "O2A8G8F8G8A2A4P64A4G#4.A8B-1B-
4"
60  PLAY "O3C4O2A4.C8F2.G8F8E-4D4E-4G4F1
F4G4A4B-4"
70  PLAY "O2G4C2.C4P64C4O1B4O2C4A1A4P64A
4B-4O3C4"
80  PLAY "O3D1D8P64D4C#4D4E-2.D4O2B2O3C4
D4C4O2G2.G8"
90  PLAY "O2P8G4F#4G4O3E-2.O2G4O3D2P64D2
O2B-1B-2."
100  END
```

Program 94. "I Left My Heart in San Francisco."

only sparingly as identical note separators.

Program 95: America the Beautiful

I came across a pleasing arrangement of "America the Beautiful" in a music manuscript book written in 1937.

Program 95 reflects the most pleasing parts of this manuscript.

Legato programming is used throughout to accurately reproduce the solemn strains. I elected to go with the default tempo (T120), which is automatically activated because there is no tempo command included in any of the eight program lines. The moderate tempo allows me to use the P64 commands without any noticeable effects detrimental to the timing of this piece.

```
10  REM "AMERICA THE BEAUTIFUL"
20  PLAY "ML O2G4P64G4.E8P64E4G4P64G4.D8
P64D4E4"
30  PLAY "F4G4A4B4G2.P64G4P64G4.E8P64E4G
4P64G4."
40  PLAY "D8P64D4O3D4C#4D4E4O2A4O3D2.O2G
4O3E4.P64"
50  PLAY "E8D4C4P64C4.O2B8P64B4O3C4D4O2B
4A4G4O3C2."
60  PLAY "P64C4P64C4.O2A8P64A4O3C4P64C4.
O2G8P64G4"
70  PLAY "P64G4A4O3C4O2G4O3D4C2."
80  END
```

Program 95. "America the Beautiful."

Like the previous program, this is a one-verse version. You can play all the verses by using FOR-NEXT statements in new lines 15 and 75, if desired. This song is often performed at a faster tempo; however the default tempo in this program closely matches the speed of the 1937 arrangement. This was a very basic manuscript, and you may wish to use this program as a starting point for a more modern version. This gives much leeway to the musicians among you who might like to add grace notes and other refinements.

Program 96: Me and Bobby McGee

Modern composer and singer Kris Kristofferson enjoyed one of his biggest hits with "Me and Bobby McGee," shown in Program 96. The most popular version was probably performed by the late Janis Joplin. She sang it blues rock-style, but the version shown here is more like the original as sung by Kris Kristofferson. Unfortunately, songs by this writer do not play very well on the AT&T PC, mainly because they entail many sequentially-run identical notes. His songs depend on the phrasing of the artist rather than the manuscript notes themselves. To overcome some of the difficulties in programming this song, I speeded the tempo considerably using a T200 command. Because there are long series of identical quarter notes, I was able to use large numbers of P64 rest separators, evenly spaced, which has the effect of stretching the entire song fairly evenly. In most instances these separators would completely destroy a song at this tempo. It would probably be simpler to use the music normal mode and switch to *legato* during different portions of the song that required this mode.

```
10   REM "ME AND BOBBY MCGEE"
20   PLAY "MFLT200O2G4P64G4P64G4P64G4P64G
4F4E2G4P64G4P64"
30   PLAY "G4F4E2.P4E4G4P64G4E4F4E4D4C4D1
D1P4F8E8F4E4F4E4D4D4"
40   PLAY "P4F8E8F4E4D1P4F8E8F4P64F4E4D4C
4D4E1E2.P4G4P64G4"
50   PLAY "P64G4P64G4P64G4F4E4F4G4P64G4P6
4G4P64G4F4E4P64E4"
60   PLAY "F4G4P64G4P64G4A4B-4P64B-4G4E4F
1F4P4A4B4O3C4P64"
70   PLAY "C4P64C4P64C4P64C4O2B4A4F4G4F4E
4F4G4E4F4E4D4P64"
80   PLAY "D4P64D4E4F4D4C4O1B4O2C1P2.O3C8
P64C8P64C4P64C4P64"
90   PLAY "C4O2B4A4P64A4G4F4E4F4G1P4F8P64
F8P64F4P64F4E4D4C4D4E1E4"
100   PLAY "P2.O3C8P64C8P64C4P64C4P64C4O2
B4A4F4P4G8F8E4F4G1"
110   PLAY "P8D8P64D8P64D8P64D4E4F4D4P64D
4P64D4O1B1B2P2P4"
120   PLAY "O2F8P64F8P64F4E4D4.P64D8P64D8
C4O1B8O2C1C4"
130   END
```

Program 96. "Me and Bobby McGee."

```
10 REM "NIGHT TRAIN"
20 PLAY"MF ML T170 O3E1P64E2L32E-FEDCO2B
B-AP8O3C8E8D8C8O2B-8G8F8E-8D8D1"
30 PLAY"O3E-1P64E-2L32FEDCO2BB-AGG-P8O3C
8E8D8C8O2B-8G8F8E-8D8D1"
40 PLAY"O3E-1P64E-2L32FEDCO2BB-AGG-P8O3C
8E8D8C8O2B-8G8F8E-8D8D1"
50 PLAY"P8O2G8O3MNC8.C16E-8.E-16C8.C16O2
A8.G16"
60 PLAY"O3C8.C16E-8.E16C8.C16O2A8.G16"
70 PLAY"O3C8MLC4.C2P64C4.P64C8C4P8"
80 PLAY"O2G8O3MN C8.C16E-8.E-16C8.C16O2A
8.G16"
90 PLAY"O3C8.C16E-8.E-16C8.C16O2A8.G16O3
C8MLC4.C2P64C4."
100 PLAY"P64C8C4P8O2G8O3MNC8.C16E-8.E-16
C8.C16O2A8.G16"
110 PLAY"O3C8.C16E-8.E-16C8.C16O2A8.G16O
3MNC8MLC4.C2"
120 PLAY"O3E1P64E2L32E-FEDCO2BB-AP8O3C8E
8D8C8O2B-8G8F8E-8D8D1"
130 PLAY"O3E-1P64E-2L32FEDCO2BB-AGG-P8O3
C8E8D8C8O2B-8G8F8E-8D8D1"
140 PLAY"O3E-1P64E-2L32FEDCO2BB-AGG-P8O3
C8E8D8C8O2B-8G8B-8B8O3MLC8C1."
```

Program 97. "Night Train."

```
10 REM "CAST YOUR FATE TO THE WIND"
20 PLAY"ML O2 E8O3E4C8O2G8G4P8O1A8O2A4F8
C8C4P8E8"
30 PLAY"O3E4C8O2G8P64G4B-8F8D2."
40 PLAY"P8E8O3E4C8O2G8G4O1A8P64A8O2A4F8C
8"
50 PLAY"C8O1A8O2C8F8D4F8D8D4E8G8F2D8P64D
4."
60 PLAY"G2D8P64D4.F2D8P64D4.G2D8P64D4E8O
3E4C8O2G8G4P8O1A8O2A4"
70 PLAY"F8C8C4P8E8O3E4C8O2G8P64G4B-8F8D2
.P8E8O3E4C8O2G8G4P8O1A8O2A4F8C8C8"
80 PLAY"O1A8O2C8F8D4F8D8E8P64E8G8E8F2."
90 END
```

Program 98. "Cast Your Fate To The Wind."

```
10 REM "AVE MARIA" BY FRANZ SCHUBERT
20 PLAY"MF ML T70 O2 F2.F4.E4F8A2.A4.G4.
F2.F2.G4.G8A8G8F4E8"
30 PLAY"O2D4E8F2.F4.A4.P64A4.A8G8F8E4D8A
4B8"
40 PLAY"A2.G#4.E4.G4.G4F8E8G8A8B-8G8E8F2
.F4."
50 PLAY"A4G8G4.G4E8D8F#8A8O3C8O2A8F#8"
60 PLAY"O2G2.P8D8E8F8A32F32E8D8C2.C4.P64
C4.G4.G4P64G8P64G4F#8"
70 PLAY"G4A8G4A8F4.F4.P64F4.G4.G4P64G8P6
4G8F#8G8B-8A8G8F2."
80 PLAY"F4.P64F4.G4.G4P64G8A4P64A8P64A8G
8A8O3C4.O2B-4.B-4.D4.A4.G4.F8E8"
90 PLAY"O2F8A-8G8F8G2.G2.F2.F4.E4F8A2.A4
.G4.F2.F2.P64F4.P64F8A8O3C8F2."
100 END
```

Program 99. "Ave Maria."

Music programs are probably among the easiest to write in GWBASIC after you get the hang of it. So far in this chapter, I have presented many different music programs, some of which use FOR-NEXT loops to repeat musical phrases and others which are just straight note-by-note programming. For those readers who are interested in music programs simply because of how they sound, I have included three music program listings (Programs 97, 98, and 99) to complete this chapter. These are just rehashes of the previous programs, although each represents a new computer tune.

SUMMARY

While the music programs presented in this chapter are discrete programs, these examples will often be incorporated into other programs to add a bit of flair. Just as graphics can enhance almost all text mode programs, sound can do likewise. When a program incorporates text, graphics, and sound, perhaps this is an example of using all of the potential of the PC-6300 to best advantage.

Index

Index