Ballantine/31945/$6.95 in USA • $8.95 in Canada

Tim Hartnell

One of the world's best selling authors of computer books teaches you to program in BASIC–through games!

How to Program Your IBM PCjr.®*

*IF YOU'VE NEVER PROGRAMMED A COMPUTER BEFORE

How to Program Your IBM PCjr®*

*If You've Never Programmed a Computer Before

How to Program Your IBM PCjr®*

*If You've Never Programmed a Computer Before

Tim Hartnell

Ballantine Books • New York

This book is available to organizations for special use.
For further information, direct your inquiries to:
Ballantine Books
Special Sales Department
201 East 50th Street
New York, New York 10022

Library of Congress Catalog Card Number: 84-91636

ISBN 0-345-31945-1

Manufactured in the United States of America

First Ballantine Books Edition: September 1984

Designed by Gene Siegel

10 9 8 7 6 5 4 3 2 1

To Evelyn Renold

CONTENTS

Foreword

If you've never programmed a computer before, and you'd like to spend just a few hours learning how to program your new IBM PCjr, then this book was written for you.

A lot of fun and powerful software is already available for the PCjr, but there will come a time when you'll want to be able to create your own programs. Wouldn't it be great to be able to impress your friends and parents by writing your own utility and games programs, or put the stamp of your own personality on programs from other sources by modifying them to suit your own interests and needs? This book will show you how to make your own programs in a matter of hours.

You'll learn the most important rules and techniques for programming and—once you've worked through the book—you'll have a library of interesting programs to keep you, and your IBM PCjr, occupied for weeks to come.

Tim Hartnell
New York, 1984

CHAPTER ONE

Getting Started

Hi there! We're going on a journey of discovery with your new IBM PCjr. You've bought a great computer, and you and your machine are going to have a lot of fun together in the coming weeks and months.

This book was written to help you get your computer up and running—with programs that *you* have created—as quickly as possible. I'll assume this is the first time you've learned to program and that the PCjr is the first computer you've handled. If I start talking about material you already know, just skip over that section.

GETTING IT ROLLING

There's an old adage which says: *If all else fails, read the instructions*. You're probably pretty impatient about getting your machine running as soon as possible. You may want to write some games or a utility routine to handle your personal finances. And you really don't want to wade through zillions of words of documentation before you can do this. As much as I'd like to save you from all technical discussions, the next section of this book is vital. If you don't read and apply it, pretty much all of the rest of the book will be useless. It's not really very difficult, and the section is not very long, but you must understand it before you dip deeper into the book. This introductory material will show you how to start your machine up, get the BASIC language ready for your programming, and how to save programs, so that you won't have to retype a program every time you want to run it.

FIRST STEPS

I assume you've followed the instructions which came with the PCjr to get the computer itself hooked up to your TV, and you've got batteries in your keyboard (unless you've bought the optional cable which links your keyboard to the *system unit*, which is the official name for the oblong box which contains the actual computer).

There are, as you know, two versions of the PCjr. The more expensive version has a *disk drive* in the top right section (when looking from the front) of the system unit. If you have a disk drive, you'll find you can do some things with your computer which are not possible if you have the version that only has the two cartridge slots in the bottom right of the system unit. The differences between the two versions give the *disk system* version slightly greater flexibility with sound and graphics, plus, of course, the ability to load programs into the computer from a disk, and the ability to save your own programs on a disk.

From time to time you'll see the words **Disk System Only** at the start of a paragraph or section. If you don't have a disk drive, skip over that section or paragraph. The parts of this book (and this applies to the vast majority of the book) not marked Disk System Only will apply to both the disk and non-disk versions of the PCjr. If you do have a disk system, but *do not* have the optional **Cartridge BASIC** (IBM product number 1502460) you must proceed following the *non-disk* instructions. If you have Cartridge BASIC, but *no* disk drive, you will be able to run *all* of the programs specified as Disk System Only but will not, of course, be able to record them on a disk. This is not really as confusing as it might sound—but it's important to know what you can and can't do with the system you're using.

First turn on the TV you're using to display your work, then turn on the power to the computer. The letters IBM will appear in white on a blue background along with a band of color at the bottom of the screen, and a number which will change a few times. Then the screen will go blank. If you have a disk drive a red light will come on for a few seconds and there will be some whirring noises as the disk spins. Hold the keyboard where you can see it. On the right hand side, about two inches in from the edge of the keyboard, you'll see a key shaped like a backwards letter 'L.' You'll see the word *Enter* and a strange bent arrow printed on this key. This key is sometimes called the RETURN key (because it is called that on most other microcomputers). The **Enter** key is probably the most important one on the keyboard. It tells your PCjr that you've finished typing some instructions on the screen, and you now want the computer to act on those instructions. It's as if your PCjr was a robot. You give it a command, then yell "Now" at the robot to make it carry out your command. Pressing the **Enter** key is much the same as yelling "Now." It tells the PCjr to get to work instantly, carrying out the instructions you've just typed in. If you have a non-disk system, you can skip the whole next section, and go to p. 7.

DISK SYSTEM ONLY

Before you turn your PCjr on, make sure the disk marked DOS (which stands for *Disk Operating System*) is in the disk drive, the gate cover is down, and that the Cartridge BASIC is in position in the cartridge slot. Once the large IBM logo disappears, you'll see the following words on the screen:

```
CURRENT DATE IS TUE 1-01-1980
ENTER NEW DATE:
```

Ignore this, and simply press the **Enter** key. A new message will appear:

```
CURRENT TIME IS 0:10:13.35
```

(This number will change)

```
ENTER NEW TIME:
```

Ignore this as well and press the **Enter** key. More words will appear on the screen:

```
THE IBM PERSONAL COMPUTER DOS
VERSION 2.10 © COPYRIGHT
IBM CORP 1981,1982,1983.
```

Underneath this you'll see:

```
A>
```

This means that the PCjr is ready to follow your instructions. Type in the word BASICA (it can be in capital or small letters) and press the **Enter** key. The screen will clear, and a new message will appear on the screen:

```
THE IBM PCJR BASIC
VERSION J1.00
COPYRIGHT IBM CORP. 1981,1982,1983
60130 BYTES FREE
```

A row of information will also be at the bottom of the screen.

You'll have to go through the steps outlined above whenever you want to enter one of the programs. This book will assume you are "in" BASIC from now on. The PCjr's big brother, the IBM PC, can use two disk drives. These are identified as drive A and drive B. Although you have only one drive, the PCjr still refers to it as drive A. Whenever you see "drive A" on the screen, or in this book, it refers to the only disk drive you have. This may seem a little confusing at first, but it is less confusing in the long run if this book and the computer are speaking the same language.

SAVING PROGRAMS

One of the most important things to learn about disk-based systems is how to save programs on disk, and how to get them back. It is very simple. You take the DOS disk out of drive A, and replace it with a blank, formatted disk. We'll be showing you how to format disks in due course, but assume for the moment that you have a formatted disk ready.

Your program is in the computer. You put the disk in Drive A, and then type in:

```
SAVE ''NNNN
```

You need to put the name of the program—CHECKERS, or whatever—in place of the NNNN. Note that there is an opening quote mark after the space following the word SAVE. Now, press **Enter** and the light on the disk drive will glow red for a few seconds, there'll be a little sound as the disk rotates, and then the message OK will appear. Your program is now saved onto the disk.

GETTING IT BACK

When you want to get the program back, you just type in:

```
LOAD ''NNNN
```

Make sure the NNNN is the same as the name under which you saved the program. There will be another little song and dance from the disk drive, and then another message OK will let you know the program is ready to run. Then, all you need to do is type in the word RUN, and press the **Enter** key, and the program will run.

NAMING YOUR PROGRAMS

The IBM PCjr is very tolerant of the procedure you use to assign names to files on disk. Although it expects a name of up to eight letters, followed by a period (.) and then an "extension" (as it is known) of up to three letters, you can actually be pretty haphazard in your naming, and the computer will sort it out.

What is really expects is a file name like:

```
CHECKERS.123
```

In this name, CHECKERS is the eight-letter name, followed by the period, and 123 is the extension.

However, as I said, you can get away with a far less disciplined approach than this.

For example, if you typed in SAVE "ENCYCLOPEDIA the computer would obediently save it, even though that name did not fit the rules. If you then type in FILES and press **Enter**, the IBM PCjr will then print up on the screen the name of every single program it has saved on the disk. You use the command FILES when you've forgotten what name you used to save a program, or you just want to check what is on a disk.

When you typed in FILES to find out what the computer had done with SAVE "ENCYCLOPEDIA you'd discover it had saved it as:

```
ENCYCLOP.EDI
```

In other words, it had taken what it needed to create an eight-letter name and a three-letter extension. To get it back, you need to type in LOAD "ENCYCLOP.EDI

BACK IN THE SYSTEM

When, shortly after you've turned your computer on, you get the A> on the screen, the IBM PCjr is in what is known as "system mode." You need to be in this mode to format disks, and your disks must be formatted before you can save your programs onto them.

To get back into system mode from BASIC, just type in SYSTEM and then press **Enter**. Following the customary whirring and red light, the A> will reappear, and you'll be back in system mode.

INSTANT FORMATTING

The computer actually does most of the work for you at this point. Make sure your DOS disk is in drive A. Type in FORMAT and hit return. The computer will then come up with the following instructions:

```
INSERT NEW DISKETTE FOR DRIVE A:
AND STRIKE ANY KEY WHEN READY
```

Just follow these instructions, and the PCjr will get to work. Eventually, a message like the following will appear on the screen:

```
FORMATTING ... FORMAT COMPLETE

    322560 BYTES TOTAL DISK SPACE
    322560 BYTES AVAILABLE ON DISK

FORMAT ANOTHER (Y/N)?
```

If you do not want to format any more disks, just press the **N** key, and the computer is ready for work again. To return to BASIC, take your formatted disk out of the drive, reinsert the DOS disk, type in BASICA, and hit the **Enter** key. Once you're back in BASIC, take the DOS disk out, and replace it with your newly formatted disk, so you can save your programs on it. Do not format a disk which already has material on it, or you'll lose it all. Formatting wipes a disk clean, programs and all.

OTHER USEFUL WORDS

You now know all you need to in order to be able to get your PCjr up and running. However, there are a few other disk-handling commands which are quite useful to know. I thought I'd put them here, so you'll have a ready source for them. You should find it more convenient to have them spelled out for you here, than having to wade through the DOS manual trying to figure out what to do.

Dir

This stands for 'directory' and is used when you're in system mode to find out what is on a disk. Type in DIR and press **Enter** and the name of every file on the disk will appear on the screen. When you're in BASIC, just type in FILES to get the same result.

Erase

Use this command to get rid of a file you don't want from a disk. When you're in system mode, you just enter ERASE "NAMEOF.FIL with the name of the file following the quote mark, much as if you were loading or saving. The equivalent in BASIC is KILL, as in KILL "NAMEOF-.FIL, which does the same thing.

Copy Nameone Nametwo

In system mode, this copies a program on disk A called NAMEONE onto disk A again under the name NAMETWO. The first program is not wiped out.

Rename Nameone Nametwo

This changes the name of a file called NAMEONE on drive A into a file called NAMETWO. You can no longer get the file by calling up NAMEONE, which has ceased to exist. The equivalent command when in BASIC is NAME.

That's the end of this section for people with disk systems. You should now proceed to the final paragraphs of this chapter (on page 8) to the section headed OTHER IMPORTANT KEYS.

NON-DISK (CASSETTE) SYSTEM

If you are using a PCjr without a disk, first turn on your TV, then turn on the power to the computer. The letters IBM in white on a blue background will appear, along with a band of colors at the bottom of the screen, and a number which will change a few times. Then the screen will go blank, and a message similar to the following will appear on the screen:

```
THE IBM PERSONAL COMPUTER BASIC
VERSION C1.20 COPYRIGHT IBM CORP 1981
62940 BYTES FREE
OK
```

At the bottom of the screen you'll see some numbers and words you can ignore. The 'C' before the 1.20 in the second line on the screen stands for *Cassette*. This tells you that the version of BASIC the computer is using will allow you to save programs onto a cassette and load them in later. Whenever you work with this book, go through the brief routine we've just described (make sure you have no cartridges plugged in the cartridge slots, turn on the TV, then the computer, and wait a few seconds) to get your BASIC ready to use.

SAVING PROGRAMS

Unless you want to go through the hassle of having to type a complete program into your PCjr every time you wish to run it, you'll need to learn how to save programs onto cassettes, and how to load them back into the computer. You'll need the IBM cassette cable to connect a cassette recorder to the PCjr.

It's pretty simple to hook the two together. On the back of the system unit you'll see a socket with the letter "C" (for cassette, of course) near the bottom right hand corner. The single connector end of the cassette cable goes in here. Plug the *red* plug into your cassette recorder where it says AUX or AUXILIARY, and plug the *black* plug into the socket marked EAR or EARPHONE.

You'll have to do a little experimenting to find the best volume level for playing tapes back so that the program loads properly into the computer. I suggest you start by setting the volume at around seven-tenths of full volume (that is, if the volume dial is marked from one to ten, set it at seven). If you have a tone control, turn that to get maximum treble. It is best to use either special computer cassettes available from your computer store (usually in 12, 15, or 20 minute lengths), or the better quality music tapes. Cheap tapes are not reliable, and it's worth spending a few extra dollars to ensure you do not lose your valuable programs forever.

When you have a program in the computer which you'd like to save on tape, proceed as follows:

- Type SAVE "NAME" into your PCjr (NAME can be any name up to eight letters long that you've decided to give your program, such as GUESS or FRIEND).

- Press the RECORD and PLAY buttons on the cassette recorder, so they both stay down and the cassette starts turning. Make a note somewhere of the number on your cassette dial, if it has one, so that later you can locate that part of the tape easily.
- Now press the **Enter** key. The cursor (that flashing little line is the cursor) will vanish.
- Wait until the word 'Ok' appears on the screen. The program is now saved on the tape.
- Stop the cassette player.

GETTING THE PROGRAM BACK

To get a previously-saved program back into your computer is just as simple. Here are the steps you follow:

- Rewind the tape to the start of the program, going back a little further than the start to make sure you don't miss any of it.
- Type in LOAD "NAME" (using the same NAME you SAVED the program with).
- Press the PLAY key on your cassette recorder. Wait 10 or so seconds, until the message "NAME.B Found" appears on the screen. This means the PCjr has found your program on the tape.
- Wait until the cursor reappears, and the message "OK" is printed on the screen. Your program now is ready to run.

OTHER IMPORTANT KEYS

Finally, in this introductory chapter, I'd like you to find two other keys on the keyboard which will be of great help to you.

Press a few keys at random. You'll see that the letters you press will appear on the keyboard. Now press the **Enter** (remember, the L-shaped key) key. The words SYNTAX ERROR will appear beneath the letters you've typed. SYNTAX ERROR is PCjr talk for "I don't understand that." You can't hurt your computer by typing in something it doesn't understand, so don't worry when you get a SYNTAX ERROR message. It usually happens because you've spelled an instruction incorrectly. When you see the words SYNTAX ERROR, check the last thing you typed in.

You'll notice that the letters you've typed on the screen are *lower-cased*—not capital letters. You'll find it best to type in capital letters when programming your PCjr. Most of the program examples in this book are in capitals. Go to the bottom right hand corner of the keyboard, and count across the bottom row to the fourth key, which has the words **Caps Lock** printed on it. Just press this. From now on, everything you type will turn out in capital letters on the screen. Press **Caps Lock** again, and you'll get small letters. Make sure you press **Caps Lock** before programming, so that everything you type appears on the screen in capital letters.

Now, look back at your keyboard. Go to the top right-hand corner. To the left of the key marked **Fn,** you'll see a key which is wider than many of the others, with the words **Back space** and an arrow printed on it. If you make an error typing, just press this key—it will move backwards along the line, wiping out letters as it goes.

Putting Things on the Screen

We start learning to program using the most commonly used command in BASIC, the word PRINT.

Type the following on your computer:

```
PRINT 2
```

Until you press the **Enter** key, the computer will do nothing. Specifically, at this point, it will ignore the command PRINT 2. Press **Enter** now, and you should see the number 2 appear underneath the words PRINT 2. This is the way PRINT works. It takes the information which follows the command PRINT, with a few exceptions which we'll learn about shortly, and PRINTs this on the screen which is, after all, exactly what you'd expect it to do.

But your computer is not completely stupid. That is, it can do more than just blindly print what you tell it to. If the word PRINT is followed by a sum, it will work it out before printing, and give you the result. Try it now. Enter the following line, then press **Enter**:

```
PRINT 5 + 3
```

You should see the figure 8 appear. The computer added 5 and 3 together, as instructed by the plus (+) sign, then printed the result on the screen. It can do subtraction as well (clever inventions, these computers). Type in this, and press **Enter** to see subtraction (and PRINT) at work:

```
PRINT 7 - 2
```

Now the computer can—of course—carry out a wide range of mathematical tasks, many of them far more sophisticated than simple addition and subtraction. But there is a slight hitch. When it comes to multiplication, the computer does not use the × symbol you probably used at school. Instead, it uses an asterisk (*) and for division the computer uses a slash (/).

DOING MORE THAN ONE THING AT ONCE

The computer is not limited to a single operation in a PRINT statement. You can combine as many as you like. Try the next one, which combines a multiplication and a division. Type it in, then press **Enter** to see the computer evaluate it:

```
PRINT 5*3/2
```

This seems pretty simple. Just type in the word PRINT, follow it with the information you want the computer to print, and that's all there is to it. But it is not quite as simple as that! Try the next one and see what happens:

```
PRINT TESTING
```

That doesn't look too good. Instead of the word TESTING we've got a zero. The computer thought we wanted a *variable*, rather than the word TESTING. We won't try to explain the meaning of the word *variable* at this point (it's not in the curriculum for this chapter), but it means simply that the computer thought you wanted to print a number which had the name *testing*. Foolish machine. Computers may be very, very clever machines, but they need to be led by the hand, like a very stupid child, and told exactly what to do. Give them the right instructions and they will carry them out tirelessly and perfectly, without an error. But give them incorrect instructions, or—even worse—confuse them, and they give up in despair, or do something quite alien to your intentions.

STRINGS

If you want the computer to print the word TESTING, you must put quotation marks around the words, like this:

```
PRINT ''TESTING''
```

This time when you press the **Enter** key, the word testing will appear at the top of the screen. This is worth remembering. When you want the computer to print out some words, or a combination of words, symbols, spaces and numbers, you need to put quotation marks around the material you want to print. Information held in this way between quotation marks is called a most peculiar name in computer circles. The jargon for the information enclosed in quotation marks is *string*. So, in our example above, the word *testing,* when enclosed in quotation marks, is a string. You can, in fact, get away with just the first quotation mark so the line reads PRINT "TESTING, but this is not good practice.

OUR FIRST PROGRAM

Type the following into your computer. Notice that each line starts with a number. Type this into the computer, and follow this with the other material.

```
10  PRINT ''JACK AND JILL''
```

Type in the next line, the one starting with 20, and press **Enter** once you have it in place. Do the same with the rest of the lines:

```
20  PRINT ''WENT UP THE HILL''
30  PRINT ''TO FETCH A PAIL''
40  PRINT ''OF WATER''
```

When you run this you'll see the following (if all is well):

```
JACK AND JILL
WENT UP THE HILL
TO FETCH A PAIL
OF WATER
```

ADDING NEW LINES

The computer, clever beast that it is, allows you to enter your lines in any order you choose. It will then sort them into order for you. Although our first program, and most of the other ones in this book, are numbered in 10's, starting at 10, there is no particular reason why you should follow this convention if you do not want to. However, there is a reason for leaving "gaps" in the counting. Although our first program could easily be numbered in 1's, it would leave no room to add later lines, if we decided there was a need to do so.

To see the computer sorting lines into order, add the following:

```
25  REM A LINE IN THE MIDDLE
```

Now type in LIST, to get the computer to list out the current program it is holding, and you'll see line 25 neatly in its proper numerical place. Now, run the program again. You should find that line 25 made no difference at all to it.

MAKING REMARKS

Why did the computer decide to ignore line 25? The word REM stands for *remark,* and is used within programs when we want to include information for a human being reading the program listing. You'll find REM statements scattered throughout the programs in this book. In each and every case, the computer ignores the REM statements. They are only there for your convenience, for the convenience of the programmer, or of someone else reading the program.

Often you'll use REM statements at the beginning of the program, like this one:

```
5  REM JACK AND JILL POEM
```

You may wonder why this would be necessary. After all, it is pretty obvious that the computer is holding a "Jack and Jill poem," even without the line 5 REM statement. You are right. In this case, there is little point in adding a title REM statement to this program. But have a look at some of the more complicated programs a little further on in this book. Without REM statements you'd have a pretty difficult time trying to work out what the program was supposed to do.

REM statements are often scattered throughout programs. They serve to remind programmers what each section of the program is supposed to do. Once you've been programming a while, you'll be amazed at how many programs you'll collect in listing form which—when you go back to them in a month or two—will seem totally obscure. You won't have a clue how the program works, or even more important, what on earth it is, or what it is supposed to do. This is where you'll find REM statements invaluable.

It is worth getting into good habits early as a programmer. So I suggest you start right now adding REM statements to programs. If you come across programs, or program fragments, in this book which you want to keep, and which do not have REM statements, get into the habit of using them by adding REM statements to these programs. And make sure you use them in your original programs. Note that if you add a remark to the end of another program line, you do not need to spell the statement REM out, but can substitute the apostrophe (') for it.

BACK TO PRINT

Let's return to the subject of the PRINT command. Empty your computer's memory by entering NEW and then pressing **Enter**. Type the following program into your computer and run it:

```
10  PRINT 1,2
15  PRINT
20  PRINT 1;2;3
25  PRINT
```

```
30 PRINT ''COMPUTER''
35 PRINT
40 PRINT ''23 + 34 = '';23 + 34
45 PRINT
50 PRINT 2*3
55 PRINT
60 PRINT 3^5
65 PRINT
70 PRINT ''THE ANSWER IS '';23 + 5 - 7/6
```

There is a lot we can learn from this program.

First, as in the Jack and Jill program, the computer executes a program line by line, starting at the lowest numbered one and proceeding through the line numbers in order until it runs out of numbers, when it stops. (You'll discover that this orderly progression of line numbers does not always apply, as there are ways of making the computer execute parts of a program out of strict numerical order, but for the time being it is best to assume that the program will be executed in order.)

Look first to line 10 of your program. You can see that there is a comma between the 1 and the 2. This has the effect of making the computer print the numbers with a wide space between them. The comma can be used in this way to space out numbers neatly for a table of results or a similar purpose. (Try PRINT 1,,2 and see what effect this has.) When you use a comma in this way, to separate the things which follow a PRINT statement (but not when the comma is inside a string, that is, between quotation marks) you'll find it divides the screen up into neat little columns. Try PRINT 1,2,3 and see the result of the commas. Then you can try the effect of PRINT 1,,,2,,,3,,,4,,,5,,,6,,,7,,,8,,,9,,,0 to make it perfectly clear what is going on.

The second line of the program, 15, is just the word PRINT with nothing following it. This has the effect, as you can see in the display on your screen, of putting a blank line between those lines which include material after the word PRINT. The same comment, of course, applies to lines 25, 35, 45, 55, and 65.

Line 20 has three numbers (1, 2, and 3) separated not by commas (as in line 10) but by semicolons (;). Instead of separating the output of the numbers as the comma did, you'll see that it causes them to be printed with a single space on either side of them. When printed, numbers are always followed by a space. Positive numbers are also preceded by a space. You use the semicolon when you want printed material to follow other printed material without a break.

Line 30 is a word, and this is a . . . . If you mentally said "string" when you came to those dots, then you're learning well. This word is a string, in computer terms, because it is enclosed within quotation marks.

Line 40 is rather interesting. For the first time we have included numbers and a symbol (=) within a string. As you can see the computer prints exactly what is within the quotation marks, but works out the result of the calculation for the material outside the quotation marks, giving—in this case—the result of adding 23 to 24. Try to remember that the computer considers everything within the quotation marks as words, even if it is made of numbers, symbols, or even just spaces, or any combination of them, while it counts everything that is not within quotes in a PRINT statement as a number. This is why it got so upset earlier when we told it to print the word *testing* without putting the word in quotation marks. It looked for a number which was called testing, and because it could not find one (as we had not told the computer to let testing equal some numerical value), it refused to cooperate.

So line 40 treats the first part, within quotation marks, as a string, and the second part, outside quotation marks, as numerical information, which the computer processed.

In line 50 we see the asterisk (*) used to represent multiplication and the computer quite reasonably works out what 2 times 3 is and prints the answer 6. In line 60 we come across a new and strange sign, ^. This means "raise to the power," so line 60 means print the result of 3 raised to the fifth power. In ordinary arithmetic, we indicate this by putting the 5 up in the air beside the three. However, it is pretty difficult for a computer to print a number halfway up the mast of another number, so we use the ^ symbol to remind us (by pointing upward) that it really means "print the second number up in the air."

The final line of this program combines a string ("The answer is") with numerical information (23 + 5 − 7/6). You can see that, as expected, the computer works out the sum before printing the answer, and prints the string exactly as it is. Look closely at the end of the string. You'll see there is a space there. After the closing quote there is a semicolon which, as we learned in line 20, joins various elements of a PRINT statement together. This semicolon means that the result of the calculation is printed up next to the end of the string.

This brings us to the end of the second chapter of the book. I'm sure you'll be pleased at how much you've learned so far and are looking forward to continuing your learning. But now you've earned a break. So take that break and then come back to the book to tackle chapter three.

CHAPTER THREE

Ringing the Changes

It's all very well getting things onto the computer's screen as we learned to do in the last chapter, but from time to time you'll discover we need to be able to get printed material off the screen during a program, to make way for more PRINT statements. We do this with a command called CLS, for Clear the Screen.

CLEAR THAT SCREEN

Enter the following program into your computer and run it:

```
10 PRINT ''TESTING''
20 INPUT A$
30 CLS
```

When you run the program, you'll see the word TESTING appear at the top of the screen, more or less as you'd expect. However, below it you'll see a question mark. Where did that come from? The question mark is known as an input prompt. An input prompt, which appears in a program when the computer comes to the word "INPUT," means the computer is waiting for you to enter something else into the machine, or just to press **Enter**. You'll recall that we spoke earlier about strings, and about how they were anything which was enclosed within quotation marks. In line 20 above the computer is waiting for a string input (because the A which follows the word INPUT is, in turn, followed by a dollar sign). You can enter a word, a number, or any combination of words, numbers, and/or symbols in response to a string input. (But you can only type in a number in response to a numerical input. If you just press the **Enter** key when the computer wants a number, the computer will assume you want zero).

Anyway, when you respond to the input prompt by pressing **Enter**, you'll see the screen clears and the word TESTING disappears. Where did it go? We pointed out that the computer works through a program in line order. First the program printed TESTING on the screen with line 10 and then progressed to line 20, where it waited for an input (or for you to press **Enter**). Once you've done this in line 20, the computer moved along to line 30 where it found CLS and obeyed that instruction. The instruction was to clear the screen, so the computer did just that and the screen cleared.

Run the program a few times, until you've got a pretty good idea of what is happening, and you've followed through—in your mind—the sequence of steps the computer is executing.

DOING IT AUTOMATICALLY

Instead of waiting for you to press the **Enter** key, you can write a program which clears the screen automatically, as our next example demonstrates. Enter this next program into your computer, type in RUN, press **Enter** and sit back for the Amazing Flashing Word demonstration. Note that you must have spaces on either side of words like FOR and TO.

```
10  PRINT "AUTOTESTING"
20  FOR A = 1 TO 500
30  NEXT A
40  CLS
50  FOR A = 1 TO 500
60  NEXT A
70  GOTO 10
```

Run this program, and you'll see the word AUTOTESTING flashing off and on at the top of the screen. What is happening here? Let's look at the program and go through it line by line. First, as you know, line 10 prints the word AUTOTESTING at the top of the screen. Next, the computer comes to line 20, where it meets the word FOR. We'll be learning about FOR/NEXT loops (as they are called) in detail in a later chapter, but all you need to know here is that the computer uses FOR/NEXT loops for counting. In this program, lines 20 and 30 (the FOR is in line 20, the NEXT in line 30) tell the computer to count from one to 500 before moving on. To stop the program, hold down the key in the top right hand corner of the keyboard marked **Fn** (for *function*) and press the **B** key (which also has the word **Break** printed on it).

So, the computer waits for a moment while counting from one to 500. Then it comes to line 40, which is the command CLS, which tells the computer to clear the screen. The computer then encounters, in lines 50 and 60, another FOR/NEXT loop, so waits a while as it counts from one to 500 again. Continuing on in sequence, it comes to 70 where it finds the instruction GOTO 10. This, as is immediately obvious, tells the computer to go to line number 10. When the computer gets to line 70, it obeys the GOTO instruction, and starts over from line 10, going through the autotesting printing, counting to 500, clearing the screen, counting to 500 again, and then coming to GOTO 10 so that it starts all over again.

CHANGING PROGRAM LINES EASILY

Your computer is provided with an EDIT function which makes it very simple to change the contents of lines within programs. Get rid of the current program by typing the word NEW and then pressing the **Enter** key. Then enter the following program into your computer. *Do not run the program.* I want to explain something about it before you do.

```
10  REM AN EDIT TEST
20  PRINT''TEST AGAIN''
30  PRINT ''AND AGAIN''
```

If you want to alter lines which are on the screen, all you have to do is use the four keys in the bottom right hand corner of the keyboard. You'll see these have arrows printed on them. Press the top key of the four (the one which also has the word **Home** printed on it). You'll see the cursor (the flashing little line) move up the screen. Once you have the cursor in the required position you can make changes you want easily.

It is a little different when you want to change lines which are not currently on the screen. If you wanted to change, say, line 10, and it was not currently on the screen, all you'd need to do would be to type in EDIT 10, followed by the **Enter** key, and line 10 would be reprinted below the rest of the listing. Once you have the line in position on the screen, you can use the middle two keys (with the arrows pointing left and right) to move the cursor to where you wanted it on the line. Next you press the **Ins** (for *insert*) key which is the third from the right on the bottom row of keys. Try it now, adding the word EXCITING before the word EDIT so your line looks like this:

```
10  REM AN EXCITING EDIT TEST
```

Then, press the **Enter** key again, type in LIST and press **Enter**. This time, when the program is listed, you'll see that the new version of line 10 is included within the program.

That was simple, wasn't it? It is just as easy to delete a word, or letter, as it is to add one. Type in EDIT 20, and once the line appears on the screen, use the arrow key with **Pg Dn** printed on it to position the cursor where you want it. Now press the **Del** (for *delete*) key, which is the second key from the right on the bottom row, to erase the letters you wish to remove. If you have only one or two wrong letters within a line, you can move the cursor to the error and type in the correct letter or letters. These will automatically replace the incorrect material.

These instructions may seem a little complex, and they certainly do not need to be mastered before you can continue your learning. If you're not sure how a particular line should be edited, and you can't be bothered looking it up in your BASIC manual, or in this book, just type the whole line again. When you press **Enter** the new line will automatically take the place of the old one within the listing.

GETTING THE PROGRAM BACK

If you want to see a complete listing after it has vanished once a program has been run, all you need to do (as we mentioned briefly a little earlier) is to type in the word LIST, then press the **Enter** key. The whole program listing will appear on the screen.

Another way of doing this is to hold down the **Fn** key in the top right hand corner, and press the **1** key (which also has **F1** for *function one* written on it). Now we can explain the meaning of the words across the bottom of the screen. You'll see the first one says 1 LIST, the second 2 RUN and so on. Instead of typing in these commands in full, you can just hold down the **Fn** key, then press the indicated numbered key. The first five preprogrammed functions are the most useful. These are:

```
1 LIST
2 RUN
3 LOAD''
4 SAVE''
5 CONT
```

It is pretty obvious what they stand for (CONT is short for *Continue*). Using the function keys can save time when you're programming on your PCjr. I suggest you copy out the names of these five functions (because the words won't always be at the bottom of the screen) with their corresponding numbers on a little strip of paper, and stick this on the flat area above the top row of keys. Then you'll know without hesitation what the keys stand for. You'll probably find—as I have—that a list placed on the computer is much simpler to read than the inverse words on the bottom of the screen.

There is no reason, when using LIST, why you must list from the top of the program. When you have longer programs, you may well want to list only part of them. You do this by use of the hyphen (-) as follows:

LIST -100	This lists up to and including line 100
LIST 50-90	This lists lines 50 to 90
LIST 150-	This lists the program from line 150 to the end
LIST 270	This lists just line 270

USING THE PRINTER

Full instructions for using the printer come with the printer, of course, but if you'd prefer not to bother with them at the moment, and you just want your printer to work, there are only two commands you need to know.

To print out a copy of whatever is on the screen, hold down the **Fn** key, and at the same time press the **P** key (which also has **Prt Sc**, for *print screen*, printed on it). To print out a listing of a program, use the

LLIST command. This will print out the program in memory onto your printer, instead of on the screen. LLIST can be used in the same way you LIST to get just parts of the listing (so LLIST 40 - 70 is valid, and will print lines 40 through 70 of your program on your printer).

CHAPTER FOUR

Descent into Chaos

It's time now to start developing some real programs. You'll notice that from this point on in the book there are some rather lengthy programs. Many of them will contain words from the BASIC programming language which have not been explained. This is because, as programs become more complex (and far more satisfying to run) it becomes more and more difficult to keep words which have not been explained out of the programs. However, this is not a major problem, and you'll probably be able to work out what many of them mean, just from seeing them in the context of a program line.

We are working methodically through the commands available on the computer, and in due course, all of the important ones will be covered. When you come across a word in a program which seems unfamiliar, just type it in. You'll find that you'll soon start picking up the meaning of words which have not been explained, just by seeing how they are used within the program. So if you find a new word, don't worry. The program will work perfectly without you knowing what the word is, and investigating the listing after you've seen the program running is likely to lead you to work out what it means.

RANDOM EVENTS

In the world of nature, as opposed to the manufactured world of men, randomness appears to be at the heart of many events. The number of birds visible in the sky at any one time, the fact that it rained yesterday and may rain again today, the number of trees growing on one side of a particular mountain, all appear to be somewhat random. Of course, we can predict with some degree of certainty whether or not it will rain, but the success of our predictions appears to be somewhat random as well.

When you toss a coin in the air, whether it lands heads or tails depends on chance. The same holds true when you throw a six-sided die down onto a table. Whether it lands with the one, the three, or the six showing depends on random factors.

Your computer's ability to generate random numbers is very useful in order to get the computer to imitate the random events of the real world. The BASIC word RND lies at the heart of using this means of generating random numbers.

GENERATING RANDOM NUMBERS

We'll start by using RND just as it is to create some random numbers. Enter the following program, and run it for a while:

```
10 PRINT RND;
20 GOTO 10
```

When you do, you'll see a list of numbers like these appear on the screen:

```
.6291626 .1948297 .6305799 .8625749 .736353
.9076439 .1619371 .134104  .699964  .4928016
.5907651 .8864992 .1386402 .6516876 .2416347
```

As you can see, RND generates numbers randomly between zero and one. If you leave it running, it will go on and on, apparently forever, writing up new random numbers on the screen.

Now random numbers between zero and one are of limited interest if we want to generate the numbers and get them to stand for something else. For example, if we could generate 1's and 2's randomly, we could call the 1's heads and the 2's tails and use the computer as a kind of "electronic coin." If we could get it to produce whole numbers between one and six, we could use the computer as an imitation six-sided die.

Fortunately, there is a way to do this. Enter the next program and run it:

```
10 PRINT INT(RND*6) + 1;
20 GOTO 10
```

When you run this program, you'll get a series of numbers chosen at random between 1 and 6, like these:

```
4  2  4  6  5  6  1  6  1  1  5  3  5  1  1  4  6
2  2  1  6  1  6  6  2  3  3  2  5  4  1  2  2  5
3  2  4  5  5  3  1  6  4  1  5  1  3  6  6  1  6
4  1  1  5  2  6  5  1  4  1  3  5  5  3  6  5  2
```

Even though we could create vast series of numbers between 1 and 6 with a program like this, it is not particulary interesting. And, if you ran the program over and over again, you'd find that the sequence of numbers was starting to look very familiar. The random numbers, as you'd discover if you ran the program a number of times, are not really random at all.

This is because the computer does not really generate random numbers, but only looks as if it is doing so. Inside its electronic head, your PCjr holds a long, long list of numbers, which it prints in order when asked for random numbers. The list is so long, that it is impossible to see a pattern in it, once it is running. However, the series always appears to start at the same place, and in most programs, this is not good enough.

SEEDING THE RANDOM NUMBER GENERATOR

Fortunately for us, there is a way to get the computer to choose a different spot within the list of numbers each time you run the program, so that the numbers it generates are more nearly random.

The key word for this is RANDOMIZE. It chooses a random starting spot for the list of numbers. We can use RANDOMIZE in two ways:

1. By asking the program user to enter a "seed" for RANDOMIZE (the seed is a number which is used within the program in the form RANDOMIZE, *n;* the same seed always produces the same sequence of "random" numbers).

2. By getting the computer to do the seeding itself. This is the most satisfactory way to do it. I prefer to do this by taking the value of the second which is being recorded by the PCjr's internal clock. This is easy to do, although the line you use for this looks very complicated. Here it is:

```
RANDOMIZE VAL(RIGHT$(TIME$,2))
```

As I warned you, it looks pretty complicated. Don't worry about that, you can use it without being able to understand it (although, if you must know, it turns the value of the seconds, the two rightmost elements of the string TIME$ into a number using VAL and then uses this to seed the random number generator). Whenever you need random numbers in a program, just include this line near the start of the program to ensure that the program is not predictable.

FAST FOOD CRAZINESS

We'll look now at the result of seeding the random number generator manually with our next program, which makes an interesting use of the computer's ability to generate random numbers. As you can see, it creates a scene where you have turned up at a fast food outlet, desperate for something to eat, and you've decided to let the random number generator pick your food for you:

```
10  REM FAST FOOD
20  CLS
30  LET A = INT (RND*4) + 1
40  PRINT ''YOU'VE ORDERED '';
```

```
50  IF A = 1 THEN PRINT ''A HAMBURGER WITH
     EVERYTHING''
60  IF A = 2 THEN PRINT ''A LARGE FRENCH
     FRIES''
70  IF A = 3 THEN PRINT ''A SERVING OF
     RIBS''
80  IF A = 4 THEN PRINT ''TWO HOT DOGS WITH
     KETCHUP''
90  FOR Z = 1 TO 500:NEXT Z
100  PRINT
110  GOTO 30
```

When you run this, you'll get something like this list of food on the screen:

```
YOU'VE ORDERED A SERVING OF RIBS
YOU'VE ORDERED A HAMBURGER WITH EVERYTHING
YOU'VE ORDERED A SERVING OF RIBS
YOU'VE ORDERED TWO HOT DOGS WITH KETCHUP
YOU'VE ORDERED A SERVING OF RIBS
YOU'VE ORDERED TWO HOT DOGS WITH KETCHUP
YOU'VE ORDERED A HAMBURGER WITH EVERYTHING
```

When you look back at the listing, you'll see how the program sets the letter A to the value of the random number in line 30. In this case, the letter A is standing for a number. It is called a variable, or (because in this case it stands for a number), it is called a numeric variable. In computer jargon, we say that, in line 30, the computer has assigned the value of the random number to the variable A. And, as you can see in lines 50, 60, 70, and 80, the value assigned to A determines which food order you place. Read this over again if it seems incomprehensible the first time.

PLANTING A SEED

Now, we are going to modify the program so it reads like the following listing:

```
10  REM FAST FOOD
20  CLS
25  INPUT ''ENTER RANDOMIZE SEED '',S
26  RANDOMIZE S
30  LET A = INT (RND*4) + 1
35  PRINT ,,A
40  PRINT ''YOU'VE ORDERED '';
50  IF A = 1 THEN PRINT ''A HAMBURGER WITH
     EVERYTHING''
```

```
60  IF  A  =  2  THEN  PRINT  ''A  LARGE  FRENCH
      FRIES''
70  IF  A  =  3  THEN  PRINT  ''A  SERVING  OF
      RIBS''
80  IF A = 4 THEN PRINT ''TWO HOT DOGS WITH
      KETCHUP''
90  FOR Z = 1 TO 500:NEXT Z
100  PRINT
110  GOTO 30
```

When you run this, the computer will stop and the words ENTER RANDOMIZE SEED will appear at the top of the screen. This is an input prompt. When you place words in quotation marks after the word INPUT, these words appear on the screen to tell you precisely the input the computer is expecting. You can enter any number you like when prompted, and this number will be used to seed the random number generator. You'll soon learn which number to enter to get the food you like. (Note that there is comma after the close quotes following the word SEED. If this comma, were replaced with a semicolon, a question mark would be printed. You can therefore put a semicolon after an INPUT prompt if you want a question mark, or a comma if you do not.)

The screen looks like this when the program is running, with the random number generated by line 30 printed by line 35.

```
ENTER RANDOMIZE SEED 64
                                       3
YOU'VE ORDERED A SERVING OF RIBS
                                       4
YOU'VE ORDERED TWO HOT DOGS WITH KETCHUP
                                       3
YOU'VE ORDERED A SERVING OF RIBS
                                       1
YOU'VE ORDERED A HAMBURGER WITH EVERYTHING
                                       2
YOU'VE ORDERED A LARGE FRENCH FRIES
```

The final thing to note about this program is line 90, which—as I'm sure you have realized—is used to put a brief pause into the program before it continues. Try taking this line out and you'll see that the program runs so quickly it is almost impossible to read the words as they scroll rapidly by you.

CHAPTER FIVE

Round and Round We Go

In this chapter, we'll be introducing a very useful part of your programming vocabulary—FOR/NEXT loops. You'll recall that we mentioned FOR/NEXT loops when demonstrating the use of CLS to clear the screen. A FOR/NEXT loop was also used in our FAST FOOD program (line 90) to add a delay.

A FOR/NEXT loop is pretty simple. It takes the form of two lines in the program, the first of which is like this:

```
10  FOR A = 1 TO 20
```

With the second like this:

```
20  NEXT A
```

The control variable, the letter after FOR and NEXT, must be the same. (You can, in fact, leave the second A out altogether, as the computer will know what you mean. However, leaving the control variable out makes programs harder to read and alter, so this practice is not recommended in your early programming days.)

As a FOR/NEXT loop runs, the computer counts from the first number up to the second, as these two examples will show:

```
10  FOR A = 1 TO 20
20  PRINT A;
30  NEXT A
```

When you run it, you'll see the numbers 1 to 20 appear on the screen, much as you have expected.

Now try this version:

```
10  FOR A = 765 TO 781
20  PRINT A;
30  NEXT A
```

This is the result of running it:

```
765   766   767   768   769   770   771
772   773   774   775   776   777   778
779   780   781
```

STEPPING OUT

In the two previous examples, the computer has counted up in ones, but there is no reason why it should always count in this way. The word STEP can be used after the FOR part of the first line as follows:

```
10  FOR A = 1 TO 100 STEP 10
20  PRINT A;
30  NEXT A
```

When you run this program, you'll discover it counts (probably as you expected) in steps of 10, producing this result:

```
10  20  30  40  50  60  70  80  90  100
```

STEPPING DOWN

The STEP does not have to be positive. Your computer is just as happy counting backwards, using a negative STEP size:

```
10  FOR A = 100 TO 10 STEP -10
20  PRINT A;
30  NEXT A
```

This is what the program output looks like:

```
100  90  80  70  60  50  40  30  20  10
```

MAKING A NEST

It is possible to place one or more FOR/NEXT loops within each other. This is called *nesting loops*. In the next example, the B loop is nested within the A loop:

```
10  FOR A = 1 TO 3
20  FOR B = 1 TO 2
```

```
30  PRINT A;''TIMES'';B;''IS''; A*B
40  NEXT B
50  NEXT A
```

The nested program produces this result:

```
1 TIMES 1 IS 1
1 TIMES 2 IS 2
2 TIMES 1 IS 2
2 TIMES 2 IS 4
3 TIMES 1 IS 3
3 TIMES 2 IS 6
```

You need to be very careful to ensure that the first loop started is the last loop which is finished. That is, if FOR A . . . was the first loop you mentioned in the program, the last NEXT must be NEXT A.

Try swapping line 10 with line 20 in the program, and see what happens when you get your FORs and NEXTs mixed up.

As you may recall, I mentioned that you do not in fact have to mention the control variable with the NEXT if you do not want to. I also said that it was not good programming practice to leave it out, as this makes programs somewhat difficult to unravel. However, as I imagine you've realized by now, leaving off the control variables at least gets around the problem of wrongly specifying the NEXT in nested loops.

You can replace lines 40 and 50 of the program with either of the following (removing the old line 50 completely):

```
40  NEXT A:NEXT B
```

or

```
40  NEXT:NEXT
```

or

```
40  NEXT A,B
```

MULTIPLICATION TABLES

You can use nested loops to get the computer to print out the multiplication tables, from one times one right up to twelve times twelve, like this (note that you can omit the semicolon between the parts of the PRINT statement within quotation marks and those parts outside; this makes for quicker program entry, but diminishes the readibility of the program):

```
10  FOR A = 1 TO 12
20  FOR B = 1 TO 12
30  PRINT A''TIMES''B''IS''A*B
40  NEXT B
50  NEXT A
```

Here's part of the output:

```
7 TIMES 11 IS 77
7 TIMES 12 IS 84
8 TIMES 1 IS 8
8 TIMES 2 IS 16
8 TIMES 3 IS 24
8 TIMES 4 IS 32
8 TIMES 5 IS 40
8 TIMES 6 IS 48
8 TIMES 7 IS 56
8 TIMES 8 IS 64
8 TIMES 9 IS 72
```

There is no reason why both loops should be traveling in the same direction (that is, why both should be counting upwards) as this variation on the Times Table program demonstrates:

```
10  FOR A = 1 TO 12
20  FOR B = 12 TO 1 STEP -1
30  PRINT A''TIMES''B''IS''A*B
40  NEXT B
50  NEXT A
```

Here's part of the output of that program:

```
4 TIMES 4 IS 16
4 TIMES 3 IS 12
4 TIMES 2 IS 8
4 TIMES 1 IS 4
5 TIMES 12 IS 60
5 TIMES 11 IS 55
5 TIMES 10 IS 50
```

CRACKING THE CODE

It's time now for our first real program. This is a game called CODEBREAKER which uses several FOR/NEXT loops. The computer thinks of a four-digit number (like 5462) and you have eight guesses in which to work out what the code is. In CODEBREAKER (based on a program by Adam Bennett and Tim Summers) you not only have to work out the four numbers the computer has chosen, but also determine the order they are in.

After each guess, the computer will tell you how near you are to the final solution. A "white" is the right digit in the wrong position, and a "black" is a correct digit in the right position within the four digits of the code. As you can see from this, you are aiming to get four blacks. Digits may be repeated within the four-number code.

Enter the program and play a few rounds against the computer. Then return to the book and read the discussion of it, which will highlight the role played by the FOR/NEXT loops.

```
10 CLS
20 RANDOMIZE VAL (RIGHT$(TIME$,2))
30 PRINT "******************************"
40 PRINT "Codebreaker by A.J.B and T.S."

50 PRINT "******************************"
60 PRINT
70 PRINT "When you are told to do"
80 PRINT "so, enter a 4-digit number"
90 PRINT "and then press RETURN."
100 PRINT
110 PRINT "Digits can be repeated."
120 PRINT
130 PRINT "You have 8 goes to break"
140 PRINT "the difficult code."
150 PRINT "******************************"
160 FOR Z=1 TO 3000:NEXT Z
170 CLS
180 DIM B(4)
190 DIM D(4)
200 LET H=0
210 FOR A=1 TO 4
220 LET B(A)= INT (RND*9)+1
230 NEXT A
240 FOR C=1 TO 8
250 PRINT
260 PRINT "Enter guess number";C;
270 INPUT X
280 IF X>9999 THEN GOTO 260
290 IF X<1000 THEN GOTO 260
300 LET P=INT (X/1000)
310 LET Q=INT ((X-1000*P)/100)
320 LET R=INT ((X-1000*P - 100*Q)/10)
330 LET S=INT (X-1000*P - 100*Q - 10*R)
```

```
340 LET D(1)=P
350 LET D(2)=Q
360 LET D(3)=R
370 LET D(4)=S
380 FOR E=1 TO 4
390 IF D(E)<>B(E) THEN GOTO 440
400 PRINT " Black";
410 LET B(E)=B(E) + 10
420 LET D(E)=D(E) + 20
430 LET H=H + 1
440 NEXT E
450 IF H=4 THEN GOTO 650
460 FOR F=1 TO 4
470 LET D=D(F)
480 FOR G=1 TO 4
490 IF D<>B(G) THEN GOTO 530
500 PRINT "  White";
510 LET B(G)=B(G) + 10
520 GOTO 540
530 NEXT G
540 NEXT F
550 FOR G=1 TO 4
560 IF B(G)<10 THEN GOTO 580
570 LET B(G)=B(G) - 10
580 NEXT G
590 LET H =0
600 PRINT
610 NEXT C
620 PRINT:PRINT "You didn't get it..."
630 PRINT "The answer is:  ";B(1);B(2);B
(3);B(4)
640 END
650 PRINT:PRINT:PRINT "Well done, codebr
eaker!"
660 PRINT
670 PRINT:PRINT "You got the answer in"
680 PRINT TAB(5);"just";C;"goes"
690 PRINT
700 PRINT
710 END
```

Here's what the screen looks like at the beginning of the run:

```
******************************
Codebreaker by A.J.B and T.S.
******************************


When you are told to do
so, enter a 4-digit number
and then press RETURN.

Digits can be repeated.

You have 8 goes to break
the difficult code.
******************************
```

And here is the end of one round played against it:

```
 Black  White

Enter guess number 3 ? 9854
  White  White

Enter guess number 4 ? 3243

Enter guess number 5 ? 7854
 Black  White

Enter guess number 6 ? 6547
  White  White

Enter guess number 7 ? 8976
 Black Black  White  White

Enter guess number 8 ? 8967
 Black  White  White  White

You didn't get it...
The answer is:   7  9  8  6
```

We'll now go through the program, with line by line, a practice we'll be following with several of the programs in this book. If you don't want to read the detailed explanation now (and there may well be parts of it which are a bit difficult to understand at your present stage), by all means skip over the explanation now and then come back to it later when you know a little more.

Lines 30, 50, and 150 print a number of asterisks to rule off the title and instructions, with blank lines printed by 60, 100, and 120. The random number generator is seeded, as we discussed earlier, by taking the VAL of the seconds part of TIME$, in line 20. Line 160 pauses for a few seconds so that you can read the instructions, before the screen is cleared by line 170. Arrays are dimensioned in lines 180 and 190. We discuss arrays in a later chapter. For now, all you need to know is that by saying DIM B(4) you tell the computer you want to create a list of objects, with the list called B, in which the first item can be referred to as B(1), the second as B(2) and so on. You do not really need to dimension an array when less than 11 elements will be needed, but it helps to keep your thinking clear to always dimension arrays before using them in programs. In this program the arrays are used for storing the numbers picked by the computer, and for storing the digits which you pick each time you try to break the code.

H is a numeric variable (we've mentioned numeric variables before) which is set equal to 0 in line 200. In line 430, 1 is added to the value of H each time a black is found, so that if H ever gets to equal 4, the computer knows all the digits have been guessed, and goes to the routine from line 650 to print up the congratulations.

The lines from 210 to 230 work out the number which you will have to try and guess. Line 220 uses the RND function we've discussed before to get four random numbers between 0 and 9, and stores one each in the elements of the B array. Note that the first FOR/NEXT loop of our program appears here. The A in line 210 equals one the first time the loop is passed through, two the second time, and so on, so that the A in line 220 changes as well.

Our next FOR/NEXT loop, which uses C, starts in the next line. It counts from 1 to 8, to give you eight guesses. Line 270 accepts your guess, after the previous line has told you which guess it is you are entering. The numeric variable X is set equal to your guess, and line 240 checks to make sure you have not entered a five-digit number (line 280) or one which has less than four digits (line 290). If you have, the program goes back to line 260 to ask you once again to enter a guess.

The next section of the program, right through to line 590, works out how well you've done, using a number of FOR/NEXT loops (380 to 440, 460 to 540, 480 to 530, and 550 to 580). Line 610 sends the program back to the line after the FOR C = . . . to go through the loop again. If the C loop has been run through eight times, then the program does not go back to line 250, but "falls through" line 610 to 620 to tell you that you have not guessed the code in time, and to tell you what it is. Line 630 prints out the code.

If you do manage to guess it, so that H equals 4 in line 450, then the program jumps to line 650 to print out the congratulatory message.

Changing in Midstream

We pointed out at the beginning of the book that, in most situations, your computer follows through a program in line order, starting at the lowest line number and following through in order until the program reaches the final line.

This is not always true. The GOTO command sends action through a program in any order which you determine. Enter the following program, and before you run it, see if you can predict what the result of running it will be:

```
10 GOTO 40
20 PRINT "this is 20"
30 GOTO 60
40 PRINT "this is 40"
50 GOTO 20
60 PRINT "this is 60"
70 FOR Z=1 TO 300:NEXT Z
80 GOTO 40
```

This rather pointless program sends the poor computer jumping all over the place, changing its position in the program every time it comes to a GOTO command. Here's what you should have seen on your screen:

```
this is 40
this is 20
this is 60
this is 40
this is 20
this is 60
this is 40
this is 20
this is 60
this is 40
this is 20
```

The program starts at line 10, and finding GOTO 40 there, moves on to line 40 to print the message "this is 40." It then continues on to line 50 where it finds the instruction GOTO 20. Without question, it zips back to line 20 to print out "this is 20" then goes to line 30 which directs it to line 60. At line 60 it finds the instruction to print out "this is 60" which it obeys. The computer then follows through to line 70 where the Z loop inserts a short pause, before the computer moves on to line 80 to find yet another GOTO instruction, this time to line 40, which is just about where we began . . . and the whole thing starts over again.

RESTRICTIVE PRACTICES

Using GOTO in this way is called *unconditional branching*. The command is not qualified in any way, so the computer always obeys it. This brings us neatly to the next computer words we will consider. These are a pair of words IF and THEN, nearly always found (or implied) together, which impose conditions on branching by GOTO commands. This pair of words is easy to understand. IF something is true, THEN do something else. IF you are hungry, THEN order a hamburger. IF you want a big car, THEN save for it. IF something THEN something.

The next program, which "rolls a die" (using the random number generator) and then prints up the result of that die roll as a word, uses a number of IF/THEN lines:

```
10 REM DICE ROLLS
20 GOTO 140
30 PRINT "ONE"
40 GOTO 140
50 PRINT "TWO"
60 GOTO 140
70 PRINT "THREE"
80 GOTO 140
90 PRINT "FOUR"
100 GOTO 140
110 PRINT "FIVE"
120 GOTO 140
130 PRINT "SIX"
140 LET A=INT (RND*6) + 1
```

```
150 FOR Z=1 TO 200:NEXT Z
160 IF A=1 THEN GOTO 30
170 IF A=2 THEN GOTO 50
180 IF A=3 THEN GOTO 70
190 IF A=4 THEN GOTO 90
200 IF A=5 THEN GOTO 110
210 IF A=6 THEN GOTO 130
```

This is what you'll see when you run the program:

```
FOUR
THREE
TWO
TWO
ONE
SIX
ONE
SIX
SIX
TWO
THREE
```

So, we've looked at unconditional and conditional GOTO's to send action all over the place within a program.

SUBROUTINES, ANOTHER WAY TO FLY

There is another way to redirect the computer during the course of a program. This is by the use of subroutines. A subroutine is part of a program which is run twice or more during a program, and is more efficiently kept outside the main program than within it.

The next program should make this clear. In this program, the computer throws a die over and over again. The first time it is thrown, the computer is throwing it for you. The second time it throws the die for itself. After each pair of dice has been thrown, it will announce who is the winner (highest number wins). The program uses a subroutine to throw the die, so we do not need two identical *die-throwing* routines within a single program. Enter and run the program, then return to the book, and I'll explain where the subroutine is within the program, and how it works:

```
10 FOR Z=1 TO 500:NEXT Z
20 PRINT:PRINT
30 FOR C=1 TO 2
40 GOSUB 130
```

```
50 IF C = 1 THEN LET A = D
60 IF C = 2 THEN LET B = D
70 NEXT C
80 IF A>B THEN PRINT "I WIN"
90 IF A<B THEN PRINT "YOU WIN"
100 IF A=B THEN PRINT "IT'S A DRAW"
110 BEEP
120 GOTO 10
130 REM THIS IS SUBROUTINE
140 LET D = INT(RND*6) + 1
150 IF C = 1 THEN PRINT "I ROLLED A";D
160 IF C = 2 THEN PRINT "YOU ROLLED A";D
170 FOR Z=1 TO 100:NEXT Z
180 RETURN
```

This is what you'll see when you run it:

```
I ROLLED A 1
YOU ROLLED A 6
YOU WIN

I ROLLED A 1
YOU ROLLED A 6
YOU WIN

I ROLLED A 6
YOU ROLLED A 2
I WIN
```

The program pauses for a short while on line 10, prints two blank lines, then enters the C FOR/NEXT loop. When it gets to line 40, which it does (of course) once each time through the C loop, the program is sent to the subroutine starting at line 140. The "die" is "rolled" in line 140, and the numeric variable D is set equal to the result of the roll. The next two lines print out the result of the roll, using an IF/THEN to determine whether the computer should print "I ROLLED A . . ." or "YOU ROLLED A . . . ." There is a slight pause (line 170) and then the computer comes to the word RETURN. The word RETURN signals to the computer that it must return to the line after the one which sent it to the subroutine. In this program, that line (the one which is after the one which sent it to the subroutine) is 50.

There, the IF/THENs in lines 50 and 60 determine whether the value of the roll (D) should be assigned to the variable A or to B.

Line 70 ends the FOR/NEXT loop, and then lines 80 to 100 determine whether the computer has won (which it will have done if A is greater than B, a condition which is tested using the "greater than" sign, >, in line 80) or whether the human has won (which will happen if A is less than B, a condition tested in line 90 by the "less than" symbol, <). From here the program goes back to line 10, where it starts again. (By the way, you get the computer to stop running an "endless" program of this type by holding down the **Fn** key and then pressing the **B** key until the program halts. Remember to keep both keys down together.)

Study this program, until you're pretty sure you know how subroutines work.

LET'S ROLL AGAIN

You may wonder if it is possible to change the earlier program, which changed the number rolled by the die into a word, using subroutines. The answer is yes, although the program with subroutines seems at first sight not much shorter than the GOTO version, and certainly it is not any clearer. Here's one way it could be done:

```
10 REM DICE ROLLS
20 GOTO 140
30 PRINT "ONE"
40 RETURN
50 PRINT "TWO"
60 RETURN
70 PRINT "THREE"
80 RETURN
90 PRINT "FOUR"
100 RETURN
110 PRINT "FIVE"
120 RETURN
130 PRINT "SIX"
140 LET A = INT(RND*6) + 1
150 IF A = 1 THEN GOSUB 30
160 IF A = 2 THEN GOSUB 50
170 IF A = 3 THEN GOSUB 70
180 IF A = 4 THEN GOSUB 90
190 IF A = 5 THEN GOSUB 110
200 IF A = 6 THEN GOSUB 130
210 FOR Z=1 TO 300:NEXT Z
220 GOTO 140
```

ON . . . GOSUB

However, there is a way to do it cleanly, using ON . . . GOSUB. This means that the computer can choose from a number of subroutine destinations, depending on the value which has been assigned to a variable.

To try and make that clear, here is another version of the dice roll program, using ON . . . GOSUB:

```
10 REM ON..GOSUB ROLLS
20 FOR Z=1 TO 300:NEXT Z
30 A= INT(RND*6) + 1
40 ON A GOSUB 60, 80, 100, 120, 140, 160
50 GOTO 20
60 PRINT "ONE"
70 RETURN
80 PRINT "TWO"
90 RETURN
100 PRINT "THREE"
110 RETURN
120 PRINT "FOUR"
130 RETURN
140 PRINT "FIVE"
150 RETURN
160 PRINT "SIX"
170 RETURN
```

Look first at line 30. This assigns a value, chosen randomly between 1 and 6, to the variable A. You may have expected the line to read LET A= . . . and so on. However, the LET is optional. It often makes the meaning of the line clear, so you can leave it in if you like. But, as you'll see when you run the program, it makes no difference to the computer.

Now the most important statement in the program, line 40. This means that if A equals 1, GOSUB the first number to follow the GOSUB command. If A equals 2, go to the second number; if A equals three go to the third, and so on.

The program can be further condensed by the use of colons. Colons allow you to place more than one program statement after a single line number. When the RETURNs are placed on the same line as the PRINT statements, as in the following version, the program closes up even more:

```
10 REM ON..GOSUB ROLLS
20 FOR Z=1 TO 300:NEXT Z
30 A= INT(RND*6) + 1
40 ON A GOSUB 60, 70, 80, 90, 100, 110
50 GOTO 20
```

```
60 PRINT "ONE":RETURN
70 PRINT "TWO":RETURN
80 PRINT "THREE":RETURN
90 PRINT "FOUR":RETURN
100 PRINT "FIVE":RETURN
110 PRINT "SIX":RETURN
120 RETURN
```

Note also that the lines have been renumbered, so they are all in neat 10's. You can do this very simply, just by typing in RENUM and then pressing **Enter**.

Getting into Music

The SOUND command is a great way to add life to your programs. If you have Cartridge BASIC (with or without the disk system) you can also use the more complicated PLAY command, which we'll look at the end of the chapter. We'll start by exploring the SOUND command, which is available on all PCjr's. It is amazing, as you'll soon discover, what a little sound can do to enhance a program.

SOUND is always followed by two numbers (or by variables representing numbers). The first number is the pitch, or frequency, of the note to be played, and the second determines how long the note will sound.

Here's a simple program which puts the SOUND statement through a few of its paces:

```
10 REM SOUND DEMO
20 FOR A = 37 TO 370 STEP 2
30 SOUND A,1
40 NEXT
```

And you can combine more than one SOUND statement at a time within a loop for an even more effective result:

```
10 REM SOUND DEMO
20 FOR A = 37 TO 370 STEP 2
30 B = 720 - A
40 SOUND A,1: SOUND B,1
50 NEXT
```

SOUND ADVICE

The control numbers for the SOUND command can be the result of calculations, instead of being integers or assigned variables. In the next program, SOUND ADVICE, you have to guess the number between 1 and 50 which the computer has thought of. The feedback on each guess—which will help you home in on the correct number in the shortest number of guesses—is in the form of output produced by SOUND. The lower the note, the closer you are to the correct answer. Once you've played the game a few times, you'll see how easily you can interpret the output.

Here's what you see on the screen when playing the game:

```
              GUESS NUMBER 1
What number am I thinking of? 3

No, 3 is not correct
              GUESS NUMBER 2
What number am I thinking of? 45

No, 45 is not correct
              GUESS NUMBER 3
What number am I thinking of? 42

No, 42 is not correct
              GUESS NUMBER 4
What number am I thinking of? 47

No, 47 is not correct
              GUESS NUMBER 5
What number am I thinking of? 48

Yes, I was thinking of 48

You got it in 5 guesses
```

Here is the listing (note that ABS in line 110 stands for "absolute" and gives the result of the calculation, stripped of its sign; if the result of a calculation is positive, ABS of that is still positive, while the ABS of a negative number is the number without its negative sign; for example, ABS (−3) is 3):

```
10 REM sound advice
20 RANDOMIZE VAL (RIGHT$(TIME$,2))
30 CLS
40 R = INT (RND*50) + 1
```

```
50 GUESS = 1
60 PRINT ,"GUESS NUMBER"GUESS
70 INPUT "What number am I thinking of";A
80 IF A<1 OR A>50 THEN 70
90 IF A = R THEN 150
100 PRINT:PRINT "No,"A"is not correct"
110 M = 37*ABS (R - A)
120 SOUND M, 15
130 GUESS = GUESS + 1
140 GOTO 60
150 PRINT:PRINT "Yes, I was thinking of"A
160 BEEP
170 PRINT:PRINT "You got it in"GUESS"guesses"
```

The first line of the program is, of course, just a REM statement to tell you the name of the program. The next line (20) seeds the random number generator and line 30 clears the screen. Line 40 sets the variable R equal to a number chosen at random between 1 and 50. This is the number which you have to try and guess. The variable GUESS is set equal to one in line 50. As you've probably realized, this counts the number of guesses you make.

Line 60 prints up the number of the current guess, and line 70 asks you to enter a number. The following line checks the size of your guess, rejecting it if it is above 50 or below 1. Note that line 80 ends up with THEN 70, rather than THEN GOTO 70, as you may have expected. You are allowed to leave out the GOTO after THEN, as the computer will understand what you mean. If you feel, however, that the program is easier to understand with GOTO in place, by all means replace it (and do the same in the following line, 90).

Line 90 checks the answer you've given, and if it finds your answer (A) is the same as the number the computer has thought of (R), then sends action to line 150 where the congratulations message is printed.

If you are not right, the program continues on to line 100 where after printing a blank line, the "no, you are wrong" message is displayed. Now we come to the interesting bit. The variable M is set equal to 37 times the absolute difference between the computer's number and your guess. The number 37 was chosen because if the difference is only 1, then the note represented by 37—the lowest note the computer can produce—is sounded. The sound is produced in line 120, 1 is added to the value of the variable GUESS, and then the computer returns to line 60 (via line 140) for the next guess.

The computer is capable of producing some quite exciting effects on its own, as our next program, which demands no action from you except for admiration, convincingly demonstrates. Take your PCjr to Loch Lomond next time you go there, and conjure up some Highland fancies.

```
10 REM PC BAGPIPES
20 DIM NOTE(8)
30 FOR B = 1 TO 8
40 READ NOTE(B)
50 NEXT
60 DATA 523.25,587.33,659.26,698.46
70 DATA 783.99,880,987.77,1046.5
80 FREQUENCY = NOTE(INT (RND*8) + 1)
90 DURATION = INT(RND*3)*5
100 SOUND FREQUENCY, DURATION
110 GOTO 80
```

If you'd like to slow the music down a little, change line 90 so it reads as follows:

```
90 DURATION = INT (RND * 3 + 2) * 5
```

MAKING YOUR OWN MUSIC

If the "auto-bagpipes" are too much for you, try the next program, which allows you to use the bottom row of keys as a kind of organ. It is not too musical, but you should have some fun with it, and it will give you an insight into one way of using the SOUND command. Make sure you disengage the **Caps Lock** (that is, make sure the computer is printing in small letters) before running this program.

```
10 REM PC ORGAN
20 REM DISENGAGE CAPS LOCK
30 DIM B(8)
40 FOR B = 1 TO 8
50 READ B(B)
60 NEXT
70 DATA 130.81,146.83,164.81,174.61
80 DATA 196,220,246.94,261.63
90 WHILE A$<> "."
100 A$ = INKEY$
110 IF A$<"Z" AND A$>"." THEN 100
120 IF A$="z" THEN SOUND B(1),3
130 IF A$="x" THEN SOUND B(2),3
140 IF A$="c" THEN SOUND B(3),3
150 IF A$="v" THEN SOUND B(4),3
160 IF A$="b" THEN SOUND B(5),3
```

```
170 IF A$="n" THEN SOUND B(6),3
180 IF A$="m" THEN SOUND B(7),3
190 IF A$="," THEN SOUND B(8),3
200 WEND
```

To play the PC ORGAN, just touch the keys on the bottom row of the keyboard, following this plan:

Z	key	produces	the	note	C
X	"	"	"	"	D
C	"	"	"	"	E
V	"	"	"	"	F
B	"	"	"	"	G
N	"	"	"	"	A
M	"	"	"	"	B
,	"	"	"	"	C

The organ will continue playing until you press the key with the period (.) on it, to the right of the , key.

As well as gaining some insights into the use of the SOUND command, there are other things we can learn from this program. Look at lines 90 and 200. They seem extremely odd, and in contrast to many other commands in BASIC which seem almost like English, appear to have no English equivalent.

WHILE and WEND are always used as a pair, much like FOR and NEXT. FOR and NEXT are used when you want a section of the program to be cycled through a specified number of times. WHILE and WEND (which you should think of as "While END") by contrast continue cycling through a section of the program until a specified condition is met. In this case (see line 90) the condition is that A$ (the key you are pressing) does not equal the period (.) which is, you'll recall, used to indicate that you want the program to end. Note in line 90 that the "less than" (<) and "greater than" (>) signs are used here together to mean "not equal to" (the use of these and the other symbols for comparison is discussed in the next chapter). You use WHILE and WEND whenever you want part of the program to be cycled through over and over again, until a certain condition is fulfilled.

DISK SYSTEM ONLY—BIGGER SOUNDS

So far we have been using a single *voice*. But the PCjr is capable of playing three notes (or voices) at once. The PCjr's big brother, the IBM PC, is far more limited. It can sing with just one voice at a time.

You've seen—in the preceding programs—that sound will produce sounds if followed by just two numbers. The first one controls the pitch of the note, and the second looks after the length of time the note sounds (see line 100 of our BAGPIPES program). We can add two additional numbers. The third number, which must be in the range zero to fifteen, controls the volume, and the fourth (which can be zero, one, or two) controls the voice. Any program which uses the third and fourth numbers must include the line SOUND ON to avoid getting an error message. Enter and run the following program, then return to the book for an explanation of how it works.

```
10 REM THREE VOICE CHOIR
20 SOUND ON
30 DIM NOTE(8)
40 FOR B = 1 TO 8
50 READ NOTE(B)
60 NEXT
70 DATA 523.25,587.33,659.26,698.46
80 DATA 783.99,880,987.77,1046.5
90 DATA 783.99,880,987.77,1046.5
100 FREQUENCY = NOTE(INT (RND*8) + 1)
110 DURATION = INT(RND*3+2)*5
120 VOLUME=INT(RND*15)
130 SOUND FREQUENCY,DURATION,VOLUME,0
140 VOLUME=INT(RND*15)
150 SOUND (FREQUENCY+9),DURATION,VOLUME,1
160 VOLUME=INT(RND*15)
170 SOUND (FREQUENCY-9),DURATION,VOLUME,2
180 GOTO 100
```

As you can hear, this THREE VOICE CHOIR program (based on the PC BAGPIPES program) produces some very mournful, but quite tuneful music.

Lines 120, 140 and 160 control the volume of each voice, making sure that the sounds do not become monotonous. You'll see that line 20 turns the sound ON. You'll need to turn up the volume on your TV in order to be able to hear the music. As an exercise, you might like to go back and modify the ORGAN program so that it plays chords, rather than single notes.

THE PLAY COMMAND—DISK SYSTEM ONLY

The PLAY command is very flexible, and can be used to produce musical sounds somewhat more simply than can be done with the SOUND command, even though the music is single voice only. Enter and run the following program on your PCjr, and listen to it carefully. Then return to the book for an explanation of how it works. Enter SOUND OFF as a direct command before you run the program.

```
10 REM PLAY DEMO
20 GOSUB 260
30 PLAY "O2"
40 GOSUB 260
50 PLAY "L64"
60 GOSUB 260
70 FOR G=1 TO 2000:NEXT G
80 PLAY "L32"
90 GOSUB 260
100 FOR G=1 TO 2000:NEXT G
110 PLAY "L16"
120 GOSUB 260
130 FOR G=1 TO 1000:NEXT G
140 PLAY "O0 C P4 D P16 E P8 E. D. C...."
150 PLAY "L32"
160 PLAY "C P4 D P16 E P8 E. D. C...."
170 PLAY "C P4 D P16 E P8 E. D. C...."
180 PLAY "T32"
190 GOSUB 260
200 PLAY "T255"
210 GOSUB 260
220 FOR G=1 TO 3000:NEXT G
230 PLAY "MLT32"
240 GOSUB 260
250 END
260 PLAY "CDEFGAB>C"
270 RETURN
```

You'll hear a number of clear, musical scales, plus a few trills. Look first at the subroutine in line 260. This contains the word PLAY followed by letters and a *greater than* sign within quote marks. PLAY is controlled by the contents of the string which follow it. In line 260 we have the notes of the scale, from C up to the C one octave higher. The *greater than* sign ensures that the second C comes out higher than the first one. If we had wanted to do so, we could have followed any of the letters with # or + to play the sharp of the note (so C# plays C sharp as does C+) or by a − sign to flatten it (so B– plays B flat). These symbols are only allowed to produce notes which actually appear as black notes on a piano (so E# is not allowed).

After the scale has been played by the subroutine call in line 20, the computer comes to line 30 and finds "O2" in the string. This is a letter O, not a zero. The O refers to *octave*. If you do not specify an octave the PCjr automatically plays in octave 4. The lowest octave is zero, the highest is six. Using the greater than sign (as in line 260) effectively raises the musical output one octave. Note that the "O2" could have been within the original string, as "O2CDEFGAB>C". The strings which follow PLAY can combine any of the controls we are discussing. They've been included as single strings (in lines like 30 and 50) to make them easier to understand. Note that any such control, such as "O2", modifies all the following music until another command of the same type changes it, so the music will stay in octave two until it hits another octave control.

Line 40 plays the scale again, in octave two, then line 50 sets the length of the notes which follow it. "L64" makes the following notes play very quickly (as sixty-fourth notes) and "L1" (the lowest number allowed) gets the computer to play long, whole notes.

After line 260 plays the scale again (very quickly), line 70 puts in a pause so you can hear the difference between the sixty-fourth note version and the thirty-second note version, produced by the "L32" in line 80. There is another pause (line 100) then line 110 slows the scale down to sixteenth notes. Line 140 forgets about the scale, and introduces the two new controls to add to your power to create music. The "O0" (capital letter "oh" followed by a numeral zero) at the start of line 140 sets the music to the lowest possible octave. The "P4", "P16", and "P8" produce pauses in the music, with the number which follows the P working the same way as the number which followed the L before. That is, "P64" will produce a pause (or *rest*, in musical terms) equal in length to one sixty-fourth note. The periods which follow the notes E, D, and C add to the length of time they play, causing the note to be played as if it was a dotted note in music.

Line 150, "L32", is (of course) another length control. The same musical phrases produced by line 140 is played (at a different speed) by lines 160 and 170. Note the the "O0" which is at the start of line 140 is not needed again in lines 160 and 170 which play the same low notes as did 140.

Line 180 is "T32". T stands for *tempo* and sets the number of quarter notes in a minute, from 32 to 255. You can hear how fast "T255" is by listening to the scale when called by line 210 after line 200 has set T to the highest possible value. T equals 120 if you do not specify a tempo. After the pause created by line 230, we come across two commands in one string: "MLT32". The "T32", as we know, sets the tempo to the lowest possible number of quarter notes per minute. What is the "ML"? This is where it helps to know a little music jargon. If you do not specify "ML", the music plays "MN", or *music normal*—with each note sounding for seven-eighths of the time specified by L. "ML" means *music legato*. This smooths out the sound, so that each note plays for the full length specified. Play with "ML" and "MN" to see the effect of each. Other controls you can try which are not illustrated in this program include "MS" (*music staccato*), which produces clipped, short notes; "MF", which ensures that subsequent notes will not start until the earlier one has finished and that the program waits until the notes are over before proceeding; and "MB" (*music background*), which stores up to 32 notes in a buffer to be played in sequence without holding up the continuing execution of the program (see your IBM PCjr BASIC manual for more complete documentation for these more complicated commands).

CHAPTER EIGHT

Making Comparisons

We all know the equals sign (=) and we've seen it in use in several programs before. We've also seen "greater than" (>), "less than" (<), and "not equal to" (<>). At this point it would be useful to briefly recap on what each of these signs are, and what they mean:

=	equals
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
<>	not equal to

You'll see these used in many programs in this book, such as the next one, which allows you to challenge the computer to a game of BRICKBAT. This is based on a program written by Graham Charlton.

This is our first moving graphics game. In it, a ball (o) moves in a sort of Squash court made up of hash (#) symbols. At the bottom is your bat (---).

You move it left and right with the Z and M keys respectively. Every time you manage to get underneath the ball so it bounces back up into the court, you get 127 points. You have 10 lives, and the aim, obviously enough, is to keep the ball in action for as long as possible. You should be able to get a score in excess of 1000 without too much trouble.

Here's what the screen looks like in the early stages of the game:

```
#################################
#Lives left: 2      Score 1400  #
#                               #
#xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx#
#xxx xxxxx xxxxxxxxxxxxxxxxxxxxx#
#xx   x x   x xxx xxxxxxxxxxxxxx#
#x              x  xxxxxxxxxxxxx#
#x x x   x          xxxxxxxxxxxx#
#                               #
#                               #
#                               #
#                               #
#      o                        #
#                               #
#
#
#
#
#            ---


High score: 0
```

```
#################################
#Lives left: 5      Score 902   #
#                               #
#xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx#
#xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx#
#xxx xxxxxxxxxxxxxxxxxxxxxxxxxxx#
#xx   x x xxxxxxxxxxxxxxxxxxxxxx#
#x         x   xx x xxxxxxxxxx x#
#                               #
#                               #
#                               #
#                               #
#                               #
#                               #
#                               #
#            o                  #
#                               #
#                               #
#       ---                     #


High score: 1568
```

And this is the program listing:

```
10 REM brickbat
20 DEFINT A - Z
30 HS = 0
40 RANDOMIZE VAL (RIGHT$(TIME$,2))
50 WHILE NOT K
60 CLS
70 KEY OFF
80 PRINT "##############################
###":REM 32 #'s
```

```
90 PRINT "#
  #":REM # 30 spaces #
100 PRINT "#
   #":REM # 30 spaces #
110 FOR A = 1 TO 5
120 PRINT "#xxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxx#"
130 NEXT A
140 FOR A = 1 TO 12
150 PRINT "#
   #":REM # 30 spaces #
160 NEXT A
170 T = 10: SC = 0
180 LOCATE 23,1: PRINT "High score:"HS
190 LOCATE 2,2:PRINT "Lives left:"T"   S
core"SC
200 WHILE T > 0
210 C = -1: D = -1: X = 9: Y = INT(RND*1
0)+10: M = Y - 2
220 A = SCREEN (X - 1, Y + 1): IF A = 12
0 THEN SC = SC + 56: C = - C: SOUND 1000
 - 60*X,1: LOCATE X - 1, Y + 1: PRINT "
":LOCATE 2,25: PRINT SC
230 IF X < 4 THEN C = - C: Y = Y + INT(R
ND*2): SOUND 40,1
240 IF X > 20 THEN 350
250 LOCATE X,Y+1:PRINT "o"
260 LOCATE 19,M - 1:PRINT "  ---   "
270 A$=INKEY$
280 IF (A$ = "Z" OR A$ = "z") AND M > 4
THEN M = M - 2
290 IF (A$ = "M" OR A$ = "m") AND M < 25
 THEN M = M + 2
300 IF X = 19 AND ABS ( Y - M) < 3 THEN
C = - C: SOUND 200,1: SC = SC + 3:LOCATE
 2,25:PRINT SC
310 IF Y + D <2 OR Y + D >30 THEN D = -D
:SOUND 500,1
320 LOCATE X,Y+1:PRINT " "
330 X = X + C: Y = Y + D
340 GOTO 220
350 T = T - 1: C = - C
360 FOR W = 100 TO 1000 STEP 40
370 SOUND W,1
```

```
380 NEXT W
390 IF INKEY$ <> "" THEN 390
400 LOCATE 2,2:PRINT "Lives left:"T"
Score"SC
410 LOCATE 19,2:PRINT "
            "
420 WEND
430 FOR X = 1 TO 25
440 SOUND (37 + RND*2020),1
450 NEXT X
460 IF SC > HS THEN HS = SC: LOCATE 22,1
:PRINT "NEW HIGH SCORE!!":FOR A = 1000 T
O 1030: SOUND A,1: NEXT
470 LOCATE 23,1: PRINT "High score:"HS
480 TIME$ = "0"
490 A = VAL (RIGHT$(TIME$,2))
500 WHILE A < 4
510 A = VAL (RIGHT$(TIME$,2))
520 SOUND 100*(A + 1),1
530 WEND
540 WEND
```

In BRICKBAT, as I said, you use the Z and M keys to move your bat (---) back and forth at the bottom of the screen, bouncing the ball back up into the court if you can. The ball will make a good "bounce" sound when it hits a wall, your bat, or the top row of x's near the top of the court. There will be a flurry of aural activity if the ball misses your bat and exits through the bottom of the court. A similar tuneful display greets you at the end of your ten lives, before a new game begins.

Notice that our *comparison symbols* are used frequently in this program. They perform a number of tests in conjunction with IF/THEN statements, bouncing the ball off the bat at the bottom of the screen, the walls or the top row of x's and deciding when the ball has missed the bat.

The words AND and OR are also used in comparison lines, chaining tests together, as you can see in lines 280, 290, 300, and 310. These two words work as follows:

AND	The computer does what follows the THEN if both of the conditions chained by the AND are true
OR	The computer carries out the instruction following the THEN if either of the conditions is true

Let's see how this works in practice. In line 280, the computer checks to see if you are pressing either the **z** OR the **Z** key AND whether

M is greater than 4. If either of the first two conditions is true AND the second one is true, THEN the final statement is executed. Look at the other lines which use AND and OR, and see if you can work out how the logic is followed.

CHAPTER NINE

Two Games and a Test

It's time now to take a break from the serious business of learning to program. As you can see in this chapter, we have three major programs which use many commands which have not yet been explained. I suggest you enter the programs just as they are, play them for your own enjoyment, then come back to the explanations which follow the listings after you've mastered the rest of the book.

I do not think it's fair to keep you waiting for major programs until you've covered everything on the computer. Also, entering short demonstration programs can get pretty boring if you're longing to really see your computer in action. I hope you'll enter the programs "on trust," returning to this chapter for the explanations when you feel you are ready. Of course, you do not have to enter the programs right now. If you'd prefer to continue with the learning, then move straight along to chapter 10.

PLAYING ALONE

Our first listing allows you to use your computer as a solitaire board. It is believed that solitaire was invented by an imprisoned nobleman in France in the late 16th Century, and was brought into England in the closing decade of the 18th Century, and from there spread to America.

The aim of the game is simple to explain, but not so simple to achieve. You start on a board which has 33 "holes" on it arranged in a cross pattern. There are marbles in 32 of the holes, and the middle position is empty.

When you run the program you'll see the solitaire board on the screen, and the number of "moves so far" will be shown. To play, you simply jump over any of your pieces vertically or horizontally, so that you end up in an empty position. The piece which you have jumped over is removed from the board. The aim of the game is to end up with just one piece in the center position.

You are told, as I said, the number of the move, and how many pieces are left on the board. You move by entering the coordinates of the piece you want to move, using the number down the side first, followed by the number across the top. These are entered as a single, two-digit number.

For example, if you wanted to move the piece which was two positions below the central hole at the start of the game, you'd enter 64, then press **Enter**, followed by 44, and **Enter** again. The board is then reprinted, and you're offered another move.

Here's the listing of the SOLITAIRE program:

```
10 REM SOLITAIRE
20 GOSUB 430
30 GOSUB 280
40 WHILE COUNT <> 1
50 REM ACCEPT MOVE
60 LOCATE 22,1:PRINT "Which peg do you w
ant to move";
70 INPUT A
80 LOCATE 22,1:PRINT "
                        "
90 IF A = 99 THEN GOTO 270
100 IF A<11 OR A>77 THEN GOTO 60
110 IF A(A) <> 1 THEN GOTO 60
120 LOCATE 22,1:PRINT A"to where";
130 INPUT B
140 LOCATE 22,1:PRINT "
                        "
150 IF B<11 OR B>77 THEN GOTO 130
160 IF A(B) <> E THEN GOTO 130
170 LET A((A + B)/2) = E: LET A(A) = E:L
ET A(B) = 1
180 LET MOVE = MOVE + 1
190 LET COUNT = 0
200 FOR F=11 TO 75
210 IF A(F) = 1 THEN LET COUNT = COUNT +
 1
220 NEXT F
230 GOSUB 280
240 LOCATE 1,1:PRINT "There are";COUNT;"
pegs on the board"
250 WEND
```

```
260 IF A(44) = 1 THEN PRINT "You did it,
 in just";MOVE;"moves!":END
270 LOCATE 22,1:PRINT "The game is over,
 and you've failed!":END
280 REM print out
290 LOCATE 3,5
300 PRINT "Enter side co-ordinate first"
310 PRINT TAB(5);"Enter 99 to concede"
320 LOCATE 8,5:PRINT "1 2 3 4 5 6 7"
330 PRINT TAB(5);
340 FOR D = 11 TO 75
350 T = 10*(INT(D/10))
360 IF D - T = 8 THEN LET D = D + 2: PRI
NT T/10:PRINT TAB(5);:GOTO 380
370 PRINT CHR$(A(D));" ";
380 NEXT D:PRINT "     7"
390 PRINT:PRINT:PRINT
400 PRINT "Moves so far:";MOVE
410 PRINT: PRINT
420 RETURN
430 REM INITIALISE
440 CLS
450 KEY OFF: REM this removes function
 key information from the bottom of the
 screen
460 DIM A(87)
470 LET E = 2
480 FOR D = 11 TO 75
490 LET T = 10*(INT(D/10))
500 IF D - T = 8 THEN LET D = D + 3
510 READ A(D)
520 NEXT D
530 LET MOVE = 0
540 RETURN
550 DATA 32,32,1,1,1,32,32
560 DATA 32,32,1,1,1,32,32
570 DATA 1,1,1,1,1,1,1
580 DATA 1,1,1,2,1,1,1
590 DATA 1,1,1,1,1,1,1
600 DATA 32,32,1,1,1,32,32
610 DATA 32,32,1,1,1
```

Line 20 sends action to the subroutine from line 430 which initializes the variables. After clearing the screen in line 440, 450 turns the key indicators off on the bottom of the screen. The A array is dimensioned in line 460. This array actually holds the board.

E is the value of an empty square. You can see what it is by typing in PRINT CHR$(2). The loop from 480 to 520 fills many of the elements of the array from the DATA statements from line 550 through to 610. Lines 490 and 500 cause the loop to jump over certain values which are not used in the game.

On returning from the initialization routine, the program directs the computer to the routine from line 280 which prints out the board. Next, the major WHILE/WEND loop (from 40 to 250) swings into action, accepting your move, and making the necessary changes to the elements of the A array.

The routine from line 190 to 220 runs through the board, using the variable COUNT to count the pieces on the board, before going to the printout routine (from 280). If COUNT equals 1 (which means there is just one "marble" left on the board) the computer "falls through" the WEND in line 250 to the end of game routine. Here the computer checks (line 260) to see if the array element which contains a marble is 44 (see line 260) and if it is, the computer knows you have won the game, and prints out a relevant message. If not, line 270 is triggered, and the mournful message "The game is over, and you've failed!" appears on the screen.

TESTING YOUR SPEED

The next program, REACTION TEST, is much shorter than SOLITAIRE, but just as much fun to display. You enter the program, type in RUN, and the message STAND BY appears. After an agonizing wait, STAND BY will vanish, to be replaced with the words "OK, PRESS THE 'Z' KEY!". As fast as you can, you leap for the Z key and press it, knowing that the computer is counting all the time.

The computer then tells you how quickly you reacted, and compares this with your previous best time. "BEST SO FAR IS . . ." appears on the screen, and the computer then waits for you to take your hands off the keyboard to prevent cheating (as if you'd do such a thing!) before the whole thing begins again.

```
STAND BY

OK, PRESS THE 'Z' KEY!

YOUR SCORE WAS 449

BEST SO FAR: 152
```

The game continues until you manage to get your reaction time to below 8, which is not an easy task.

Here's the listing of REACTION TEST:

```
10  REM REACTION TEST
20 LET HISCORE = 1000
30 RANDOMIZE VAL(RIGHT$(TIME$,2))
40 WHILE HISCORE > 7
50 CLS
60 PRINT:PRINT:PRINT "STAND BY"
70 FOR A = 1 TO 700 + RND*2000
80 NEXT A
90 LET A$ = INKEY$
100 IF A$ <> "" THEN GOTO 70
110 PRINT:PRINT:PRINT "OK, PRESS THE 'Z'
 KEY!"
120 LET COUNT = 0
130 LET COUNT = COUNT + 1
140 LET A$ = INKEY$
150 IF A$ <> "Z" AND A$ <> "z" THEN GOTO
 130
160 PRINT:PRINT:PRINT "YOUR SCORE WAS";C
OUNT
170 IF COUNT<HISCORE THEN LET HISCORE =
COUNT:BEEP
180 PRINT:PRINT "BEST SO FAR:";HISCORE
190 FOR A = 1 TO 1000: NEXT A
200 LET A$ = INKEY$
210 IF A$ <> "" THEN GOTO 200
220 WEND
230 PRINT:PRINT "YOU'RE THE CHAMP!"
240 BEEP
250 END
```

Line 20 sets the variable HISCORE to 1000 and the random number generator is seeded in line 30. The variable COUNT is set to zero in line 120 and incremented by 1 every time line 130 is revisited, which occurs when you have not managed to get to the Z key. Lines 140 and 150 check to see if you have touched the Z key, and if not, sends the program back to 70 where COUNT is incremented.

Once you've managed to get to Z, the program "falls through" to line 160, where you are told your score. This is compared with the best score (variable name HISCORE) in the following line, and HISCORE is adjusted to COUNT if COUNT is the lower of the two.

The next line (190) puts in a short pause, and then line 200 checks to make sure you have taken your hands off the keyboard. It stays cycling through 200 and 210 until you take your hands off the keys.The WEND then sends the program back to the line after the WHILE (line 40) and the next round of the game begins.

The WHILE/WEND continues only so long as HISCORE stays greater than 7 (as you can see in line 40). Once you get a high score below 8, the program continues through the WEND to lines 230 and 240 where the words "YOU'RE THE CHAMP!" appear on the screen.

A CORNER ON GO

Our next program is a fascinating one. It allows you to play against the computer in the Japanese board game HASAMI SHOGI. Whereas the board games with which you might be familiar—such as chess and checkers—are played on an eight by eight board, HASAMI SHOGI is played on a nine by nine board. The corner of a GO board (which is 19 by 19) is often used.

Each player starts the game with 18 "stones." Your stones are rounded, and the computer's stones are square. You start at the bottom of the board playing up the screen, and the computer starts at the top. The aim of the game is to capture seven of your opponent's pieces. Actually in real-life HASAMI SHOGI, the aim of the game is to capture all your opponent's pieces, but—as you'll see when you play this game—this would make each round last a long, long time.

This may be satisfactory when you're playing against another human being and can chat as the game proceeds, but you'll find it takes far too long to complete a game when playing against the machine. Therefore, we've changed the aim of the game so that each of you is trying to capture seven of your opponent's pieces.

You take turns moving, and you can only move one piece per turn. You move vertically or horizontally, but not diagonally. With each move you have three choices:

1. You can move into a vacant square which is above, below, or beside your piece
2. You can jump over one of your own pieces into an empty square
3. You can jump over an opponent's piece into an empty square

Unlike what happens in checkers, a piece which has been jumped over is not captured and is not removed from the board. The only way you can capture a piece is by moving so that you trap a computer piece between two of yours. It captures your pieces in a similar way.

You do not lose your piece simply by moving in between two computer pieces. Therefore, you aim to get your piece next to a computer piece, with an empty square beyond that, so you can move into an empty square on a subsequent move.

Here's the listing for HASAMI SHOGI:

```
10 REM HASAMI SHOGI
20 REM Engage Caps Lock!
30 GOSUB 850
40 WHILE NOT K
50 GOSUB 100
60 GOSUB 490
70 GOSUB 660
80 GOSUB 490
90 WEND
100 REM CAPTURE
110 A = 99
120 IF A(A) <> C THEN 210
130 IF A(A - 10) = E THEN IF A(A - 9) = H
 THEN IF A(A - 8) = C THEN SOUND 50,1:B
 = A - 10:GOTO 380
140 IF A(A - 10) = E THEN IF A(A - 11) = H
 THEN IF A(A - 12) = C THEN SOUND 100,
 1:B = A - 10: GOTO 380
150 IF A(A - 10) = E THEN IF A(A + 11) = H
 THEN IF A(A + 12) = C THEN SOUND 150,
 1:B = A - 10: GOTO 380
160 B = 1
170 IF A + 2*C(B) < 11 OR A + 2*C(B) > 99
 THEN GOTO 190
180 IF A(A + C(B)) = E AND A(A + 2*C(B)) = H
 AND A(A + 3*C(B)) = C THEN A(A + 2*
 C(B)) = E:CS = CS + 1: GOTO 360
190 IF B<4 THEN B = B + 1: GOTO 170
200 IF RND < .05 THEN SOUND 37 + 2*A,1
210 IF A > 11 THEN A = A - 1: GOTO 120
220 REM Non-capture
230 COUNT = 0
240 COUNT = COUNT + 1: SOUND 37 + 16*COUNT,1
250 A = RND*89 + 11
260 IF A(A) = C THEN SOUND 200,1: GOTO 290
270 IF COUNT < 200 THEN 240
280 LOCATE 1,1:PRINT "Hasami Shosi Master":
 PRINT "I give you the victory!":END
290 B = 1
300 IF A + 2*C(B) < 11 THEN 320
310 IF (A(A + C(B)) = C OR A(A + C(B)) =H)
```

```
 AND A(A + 2*C(B)) = E THEN B = A + 2*
 C(B):GOTO 380
320 IF A(A + C(B)) = E THEN 350
330 IF B < 4 THEN B = B + 1: GOTO 300
340 GOTO 270
350 REM PCjr MOVES
360 B = A + C(B)
370 SOUND 400,1
380 B1 = B - 10*(INT(B/10))
390 A(B) = C: A(A) = E
400 IF B1 > 7 THEN 420
410 IF A(B + 1) = H AND A(B + 2) = C THEN
 A( B + 1) = E:CS = CS + 1
420 IF B1 < 3 THEN 440
430 IF A(B - 1) = H AND A(B - 2) = C THEN
 A(B - 1) = E: CS = CS + 1
440 IF A > 89 THEN 460
450 IF A( B + 10) = H AND A( B + 20 ) = C
 THEN A( B + 10 ) = E: CS = CS + 1
460 IF A < 29 THEN RETURN
470 IF A( B - 10) = H AND A( B - 20) = C
 THEN A( B - 10) = E: CS = CS + 1
480 RETURN
490 REM Board printout
500 IF RND < .3 THEN SOUND 800,1
510 LOCATE 4,14:PRINT " 1 2 3 4 5 6 7 8 9":
 LOCATE 4,14
520 FOR M = 90 TO 10 STEP - 10
530 PRINT TAB(13)CHR$(M/10 + 64);" ";
540 FOR N = 1 TO 9
550 PRINT CHR$(A(M + N));" ";
560 NEXT
570 PRINT CHR$(M/10 + 64)
580 NEXT
590 PRINT TAB(15);"1 2 3 4 5 6 7 8 9"
600 PRINT:PRINT ,"Junior:"CS
610 PRINT ,"Human:"HS
620 IF CS > 6 OR HS > 6 THEN 640
630 RETURN
640 IF CS > HS THEN PRINT:PRINT "I win!":END
650 PRINT:PRINT "You win!":END
660 REM player move
670 LOCATE 20,7:INPUT "From (letter,no)";A$
680 IF ASC(A$) > 96 AND ASC (A$) < 123 THEN
```

```
 LOCATE 20,7:PRINT "ENGAGE CAPS LOCK":FOR Z
 = 37 TO 100: SOUND Z, 1:NEXT: GOTO 670
690 IF A$ = "S" THEN END
700 IF LEN (A$) <> 2 THEN 670
710 LOCATE 20,7:PRINT "                    "
720 LOCATE 20,7:PRINT "From "A$" to ";:INPUT
 B$
730 IF LEN (B$) <> 2 THEN 720
740 LOCATE 20,7:PRINT "                    "
750 A = 10 *(ASC (A$) - 64) + VAL (RIGHT$
 (A$,1))
760 B = 10 *(ASC (B$) - 64) + VAL (RIGHT$
 (B$,1))
770 Y = VAL (RIGHT$(B$,1))
780 A(B) = H: A(A) = E
790 IF A(B + 1) = C AND A(B + 2) = H AND Y
 <= 7 THEN A(B + 1) = E: HS = HS + 1
800 IF A(B - 1) = C AND A(B - 2) = H AND Y
 >=3  THEN A(B - 1) = E: HS = HS + 1
810 IF B > 79 THEN 830
820 IF A(B + 10) = C AND A(B + 20) = H THEN
 A(B + 10) = E:HS = HS + 1
830 IF B >= 31 THEN IF A(B - 10) = C AND A
 (B - 20) = H THEN A(B - 10) = E:HS = H
 S + 1
840 RETURN
850 REM initialise
860 RANDOMIZE VAL(RIGHT$(TIME$,2))
870 KEY OFF
880 DEFINT A - Z
890 CLS
900 DIM A(129), C(4)
910 H = 72: C = 67: E = 42
920 FOR Z = 11 TO 29
930 IF Z = 20 THEN Z = 21
940 A(Z) = H
950 NEXT
960 FOR Z = 31 TO 79
970 IF 10*INT(Z/10) = Z THEN Z = Z + 1
980 A(Z) = E
990 NEXT
1000 FOR Z = 81 TO 99
1010 IF Z = 90 THEN Z = 91
1020 A(Z) = C
```

```
1030 NEXT
1040 HS = 0
1050 CS = 0
1060 FOR Z = 1 TO 4
1070 READ C(Z)
1080 NEXT
1090 DATA -10, -1, 1, 10
1100 GOSUB 490
1110 RETURN
```

The program listing may seem very long, but it is somewhat simpler to understand than it may appear at first sight. This is because it has been written in "modules," each of which is called by a GOSUB near the start of the program. Programming in this way is a good idea if you are developing a program which could be fairly long and involved, as it enables you to keep track of what each section is doing.

As well, it is easy to track down bugs (the common computer word for a program error, and named when a malfunction in an early computer was found to have been caused by a moth which was caught inside the cabinet) when the program is in modules. The module which contains the bug should be relatively easy to isolate.

The idea of programming in "modules" in this way is perhaps easier to understand if we show how it has worked in this program:

Line 30: GOSUB 850. This sends the program to the subroutine which sets up all the starting variables. This kind of subroutine is often called the *initialization subroutine.* On returning from the subroutine, the computer comes to the line WHILE NOT K.

Line 40: WHILE NOT K. This is a convenient way of ensuring that a perpetual loop is created. So long as K is not a variable used in the program, WHILE NOT K will ensure that the program circles endlessly between that line, and the WEND line. It is a useful dodge to use whenever you want a program to cycle through a program forever. Use any name in place of the K which has not been used elsewhere in the program.

Line 50: GOSUB 100. This is the subroutine which determines the computer's move.

Line 60: GOSUB 490. The subroutine starting from line 490 prints out the board after each move.

Line 70: GOSUB 660. This subroutine accepts and acts on the player's move.

Line 80: Again the subroutine to print the board is called.

Line 90: WEND. This sends action back to the line after the WHILE NOT K.

Now if you take a moment to consider the implications of these numbers, you should see what I was driving at when talking about modules. Here is the breakdown of what is really happening as the program runs:

```
50  PRINT THE BOARD
60  LET THE COMPUTER MADE A MOVE
70  PRINT THE BOARD AGAIN
80  LET THE HUMAN MAKE A MOVE
90  GO BACK TO 50 AND START AGAIN
```

As the conditions for terminating the game are within the other modules, it will continue to run in this way until one of the conditions is met. You can see from this breakdown that if, for example, the computer consistently misprinted the board, I would know the problem almost certainly was within the subroutine beginning at line 490, because that is the subroutine which prints the board.

In the same way, if the computer moved, for example, the piece next to the one I intended, every time, I would know there was a problem within the subroutine which dealt with the player's move, the subroutine starting from line 660.

This ability to know where in your program the error is likely to be, before you start trying to track it down, is a very useful one.

It is likely, in a program like this, that the real problems will lie within the routines which give the computer its "intelligence." I have found that the best way to work is to write the first few lines (such as lines 10 to 90 in this program) before anything else is written. Then, I write the other parts module by module. Once the initialization is over, I can work on the board printing routine, and not worry about the fact that I do not have a clue as to how I would eventually get the computer to play.

Then, I could work on the other modules, one by one, perfecting each one before having to bother about the others.

You'll find that you can follow this routine no matter what kind of program you're writing. Programs for business or home applications should be handled in the same way. Break the program into self-contained modules, and then write a "loop" at the beginning of the program to cycle through all the subroutine calls. Although this can make programs a little longer than they otherwise might be, the gain in easy understanding more than outweighs the additional typing which might be needed.

CHAPTER TEN

Stringing Along

You'll recall that several times in this book so far we have referred to numeric variables (letters like A or B, words like COUNT and GUESS, and combinations such as R2D2 and C3PO) and to string variables (one or more letters followed by a dollar sign, such as NAME$, A$, or AGE$ is a string variable). In this chapter, we'll be looking at strings, and at things you can do with them.

THE CHARACTER SET

Every letter, number or symbol the PCjr prints has a code (the code, by the way, is an ASCII code; ASCII is explained in the glossary). Telling the computer to print the character of that code produces the character.

It is easy to understand this. As the code is an ASCII code, as I pointed out above, the computer word for the code is ASC. Note that the ASC value for the letter "A" has nothing to do with the value assigned to A when it is a numeric variable, but refers to "A" when we actually want the computer to print the letter "A." Note that we put the "A" in quotation marks when we're referring to it as a letter.

Try it now. Enter the following into your computer, and see what you get:

```
PRINT ASC(''A'')
```

Note that the letter for which you want the ASC must be within parentheses and also within quotation marks, as above. Now when you get the computer to run the above line, it should give the answer 65. (If you didn't get that, you either left something out in the line or you're using an a instead of an A.)

From this we can see that 65 is the ASC (ASCII code) of "A". We can turn a 65 back into an "A" by asking the computer to print the character which corresponds to ASC code 65. We do this with the BASIC word CHR$, as follows:

```
    PRINT CHR$(65)
```

Run this, and the letter "A" will appear. You can get your PCjr to print out every ASC code and its character with the next short program. Enter it, and watch closely:

```
    10 REM SHOWING ASC AND CHR$
    20 FOR A = 32 TO 255
    30 PRINT A;CHR$(A);" ";
    40 FOR B = 1 TO 50:NEXT B
    50 NEXT A
```

This is the start of the printout you'll see:

```
                32     33 !   34 "   35 #   36 $   37 %
38 &   39 '   40 (   41 )   42 *   43 +   44 ,   45 -
46 .   47 /   48 0   49 1   50 2   51 3   52 4   53 5
54 6   55 7   56 8   57 9   58 :   59 ;   60 <   61 =
62 >   63 ?   64 @   65 A   66 B   67 C   68 D   69 E
70 F   71 G   72 H   73 I   74 J   75 K   76 L   77 M
78 N   79 O   80 P   81 Q   82 R   83 S   84 T   85 U
86 V   87 W   88 X   89 Y   90 Z   91 [   92 \   93 ]
94 ^   95 _   96 `   97 a   98 b   99 c   100 d   101 e
102 f   103 g   104 h   105 i   106 j   107 k   108 l
109 m   110 n   111 o   112 p   113 q   114 r   115 s
116 t   117 u   118 v   119 w   120 x   121 y   122 z
123 {   124 |   125 }   126 ~
```

TESTING YOUR CHARACTER

Our next program is a reaction tester like the one you experienced earlier. However, you are not just being tested on speed. In this program, you have to try and find the *right key* on the keyboard as quickly as possible.

Make sure that **Caps Lock** is engaged when you run the program. A letter will appear on the screen. As quickly as you can, find that letter on the keyboard and press it. You'll be told how long it took you, and this time will be compared with your best time.

Notice how the letter which is printed on the screen uses CHR$ in line 70, printing the character of the number chosen at random by line 40 and assigned there to variable A. A$ is set equal to INKEY$ (which is explained a little later in the book) in line 80 and compared with the letter the computer has chosen in line 90.

```
20 CLS
30 BEST = 1000
40 A = 65 + INT(RND*26)
50 B = 0
60 LOCATE 13,7
70 PRINT CHR$(A)
80 A$ = INKEY$
90 IF A$ = CHR$(A) THEN BEEP:GOTO 160
100 B = B + 1
110 LOCATE  9,5
120 PRINT B
130 IF B < 200 THEN 80
140 PRINT "Sorry, time is up"
150 GOTO 180
160 PRINT "Well done, you scored"
170 PRINT B;"on that one"
180 IF  B < BEST THEN BEST = B
190 PRINT:PRINT:PRINT
200 PRINT "The best score so far is";BEST
210 FOR G = 1 TO 15*B
220 NEXT G
230 CLS
240 GOTO 40
```

CUTTING THEM UP

One of the very useful aspects of the BASIC on your PCjr is the way it can be used to manipulate strings. The words used to handle strings are:

```
LEFT$
MID$
RIGHT$
```

(By the way, these are usually spoken aloud as *left-string*, *mid-string*, and *right-string*.)

The next program shows them in action. Enter it and run it on your computer, then return to the book for a discussion to show what can be learned from it.

```
10 CLS
20 A$ = "FIFTH*AVENUE"
30 PRINT
40 PRINT "LEFT$(A$,3) = ";LEFT$(A$,3)
50 PRINT
60 PRINT "LEFT$(A$,5) = ";LEFT$(A$,5)
70 PRINT
80 PRINT "RIGHT$(A$,3) = ";RIGHT$(A$,3)
90 PRINT
100 PRINT "RIGHT$(A$,5) = ";RIGHT$(A$,5)
110 PRINT
120 PRINT "MID$(A$,3) = ";MID$(A$,3)
130 PRINT
140 PRINT "MID$(A$,5) = ";MID$(A$,5)
150 PRINT
160 PRINT "MID$(A$,5,4) = ";MID$(A$,5,4)
170 PRINT
180 PRINT "MID$(A$,2,7) = ";MID$(A$,2,7)
```

As you can see, the program first (in line 20) sets A$ equal to "FIFTH*AVENUE". Then it uses LEFT$, RIGHT$ and MID$ to extract the original string, A$.

Here's what it looks like when you run it:

```
LEFT$(A$,3) = FIF

LEFT$(A$,5) = FIFTH

RIGHT$(A$,3) = NUE

RIGHT$(A$,5) = VENUE

MID$(A$,3) = FTH*AVENUE

MID$(A$,5) = H*AVENUE

MID$(A$,5,4) = H*AV

MID$(A$,2,7) = IFTH*AV
```

Look at the first line of the output. LEFT$(A$,3) = FIF. LEFT$ takes the *leftmost* portion of the string as far as the number which follows the string. That is, when we have LEFT$(A$,3) it takes the three leftmost characters of the string. The next printout, LEFT$(A$,5) takes the five leftmost characters of the string, producing in this case FIFTH (because they are five leftmost characters of the overall string).

It can be used slightly differently. If we said:

```
PRINT LEFT$(''FIFTH*AVENUE'',3)
```

the computer would print out FIF. The string, then, can either be a string variable (A$) or the string in full ("FIFTH*AVENUE").

As you've probably worked out by now, RIGHT$ does the same thing as LEFT$, except it starts at the right-hand end of the string. Therefore, RIGHT$(A$,3) selects the three rightmost characters of the string, in this case NUE. Again, as above, this is the same as saying:

```
PRINT RIGHT$(''FIFTH*AVENUE'',3)
```

MID$ is a little more flexible. It selects a portion from the middle of the string, *starting from* the character number which follows the string. Therefore, MID$(A$,4) prints all the string starting with the fourth character.

If there is only one number (such as the 4 above), then MID$ selects *all* of the string to the end of it. However, if there is another number, this second number dictates the *length* of the string which will be extracted.

You can see in the last two printouts from the program that MID$(A$,5,4) prints the extract of the string four characters long, starting from character five. MID$(A$,2,7) produces a string seven characters long starting from the second character.

Rerun the program now, putting your name in place of FIFTH* AVENUE in line 20.

PUTTING THEM BACK TOGETHER

Strings can be added together on the PCjr. The process of adding strings is called the frightening-looking word *concatenation*. You can concatenate two or more complete strings together, or just add bits of them, as our next program shows:

```
20 RANDOMIZE VAL(RIGHT$(TIME$,2))
30 A$ = "AMERICA"
40 B$ = "COLUMBUS"
50 C$ = A$ + B$
60 PRINT "A$ = ";A$
70 PRINT
80 PRINT "B$ = ";B$
90 PRINT
100 PRINT "C$ = ";C$
110 PRINT
```

```
120 D = INT(RND*6) + 1
130 E = INT(RND*6) + 7
140 PRINT "MID$(C$";D;",";E;") = ";MID$(C$,
    D,E)
150 PRINT
160 D$ = MID$(C$,D,E)
170 E$ = A$ + D$
180 PRINT "E$ = ";E$
```

When you run this program, which creates C$ in line 50 by concatenating A$ and B$, you'll see results like these two:

```
A$ = AMERICA

B$ = COLUMBUS

C$ = AMERICACOLUMBUS

MID$(C$ 1 , 9 ) = AMERICACO

E$ = AMERICAAMERICACO
```

```
A$ = AMERICA

B$ = COLUMBUS

C$ = AMERICACOLUMBUS

MID$(C$ 3 , 9 ) = ERICACOLU

E$ = AMERICAERICACOLU
```

PLAYING AROUND

You can do a number of things with string manipulation, as our next program demonstrates. NAME PYRAMID allows you to enter your name to produce a very interesting display. Once you've seen the program running, you'll understand why the program has been given the name it has.

This is the listing of NAME PYRAMID:

```
10 REM NAME PYRAMID
20 CLS
```

```
30 INPUT "What is your full name";A$
40 IF LEN (A$) > 15 THEN A$ = LEFT$(A$,15)
50 A = LEN (A$)
60 CLS
70 FOR G = 1 TO A
80 PRINT TAB(16 - G);
90 FOR H = 1 TO 2*G
100 PRINT MID$(A$,G,1);
110 NEXT H
120 PRINT
130 NEXT G
```

And here are two runs of the program, one using my name, and the other the name of a character who has been widely used in advertising the PCjr:

```
              TT
             iiii
            mmmmmm

          HHHHHHHHHH
         aaaaaaaaaaaa
        rrrrrrrrrrrrrr
       tttttttttttttttt
      nnnnnnnnnnnnnnnnnn
     eeeeeeeeeeeeeeeeeeee
    llllllllllllllllllllll
   llllllllllllllllllllllll
```

```
              CC
             hhhh
            aaaaaa
           rrrrrrrr
          llllllllll
         iiiiiiiiiiii
        eeeeeeeeeeeeee

      CCCCCCCCCCCCCCCCCC
     hhhhhhhhhhhhhhhhhhhh
    aaaaaaaaaaaaaaaaaaaaaa
   pppppppppppppppppppppppp
  llllllllllllllllllllllllll
 iiiiiiiiiiiiiiiiiiiiiiiiiiii
nnnnnnnnnnnnnnnnnnnnnnnnnnnnnn
```

PLAYING IT BACK

Our final program in this chapter shows one very effective use of string manipulation, in which a string is progressively reduced by one element.

When you run ECHO GULCH, you'll see a letter appear on the screen. It will then vanish. Once it has vanished, you will have a limited amount of time in which to press the key yourself.

If you've pressed the right key, a beep will sound, and the letter will be replaced with a new one. This will stay on the screen for a shorter time than the previous one.

Each time a new letter appears, you will be given less time to see it, before you have to press that particular key on the keyboard. If you make a mistake, the "SORRY, THAT IS WRONG" message will appear, along with your score. If you manage to get the whole list of letters right, you'll be rewarded with a "YOU'RE THE CHAMP!!" message.

Here's the listing:

```
10 REM Echo Gulch
20 REM Maximum score is 30
30 S = 0
40 A$ = "ABSCHDEUFKJHJHEUSKCJKMLKESJKDHC"
50 CLS
60 LOCATE 12,15
70 PRINT MID$(A$,1,1)
80 FOR G = 1 TO 17*LEN(A$):NEXT G
90 CLS
100 B$ = INKEY$
110 IF B$ < "A" OR B$ > "Z" THEN 100
120 LOCATE 12,15
130 PRINT B$
140 IF B$ = MID$(A$,1,1) THEN BEEP:BEEP:S =
    S + 1
150 LOCATE 5,1
160 PRINT "Your score is"S
170 LOCATE 12,15
180 PRINT " "
190 IF B$ <> MID$(A$,1,1) THEN 240
200 A$ = MID$(A$,2)
210 IF LEN(A$) = 1 THEN 270
220 FOR G = 1 TO 500:NEXT G
230 GOTO 60
240 PRINT:PRINT "SORRY THAT IS WRONG"
250 PRINT:PRINT "YOU SCORED"S
260 END
270 PRINT "YOU'RE THE CHAMP!!"
```

The variable S, which holds your score, is set to 0 in line 30, and line 40 sets the string variable A$ to a long line of letters. Line 70 prints the first letter only of the string, and line 80 insets a short delay loop, which uses the LEN function.

This is another string function, and returns the length of a string, that is, the number of characters which make it up. LEN does not make any distinction between letters, numbers, symbols, or spaces, as you'll discover if you enter a number of PRINT LEN A$ statements, after setting A$ to equal various words, symbols, and sentences.

Because, in our program A$ is reduced by one character by line 200 each time the program cycles, LEN A$ is a smaller number each time. Therefore, the delay produced by line 80 (which dictates how long the character will be on the screen before it vanishes) becomes shorter.

INKEY$

Line 100 uses INKEY$ to read the keyboard. INKEY$, as you've probably worked out by this point in the book, does not demand that you press RETURN after touching a key. INKEY$ always returns the key you have pressed as a string. INKEY$ does not, in contrast to INPUT, wait until you have pressed a key before the program continues.

If you are not touching a key when the program comes to an INKEY$, it simply passes right through the line, reading your non-touching of the keyboard as the null string (two quotation marks with nothing, not even a space, between them, as "").

Line 110 looks at B$, the variable which is set equal to whichever key is being pressed as the program goes through line 100. As you can see in line 110, you can use the "greater than"(>) and the "less than" (<) symbols, which we discussed in chapter 8, in connection with strings. These look at all elements of a string and compare them in terms of alphabetical order.

As well, you can compare strings using "equals" (=) and "not equals" (<>), as shown by the next few lines of the program. Line 140 compares the key you have pressed with the first element of A$, and if they are the same, continues through the multi-statement line to BEEP and then adds 1 to your score (variable S).

Line 180 then blanks out the letter, in preparation for the next one to appear. Line 190 compares B$ with the first element of A$ again, and if it finds they are not equal, sends the program to lines 240 and 250 where you are told "SORRY THAT IS WRONG" and your score is given.

Line 200 strips the string A$ of its first character, by setting A$ equal to MID$(A$,2). Line 210 checks to see if the length of A$ equals 1 (that is, if LEN A$ = 1) and if it finds that it is, goes to line 270 to print out the "YOU'RE THE CHAMP!!" message. If not, the program cycles back to line 60 to print out the next letter for you.

CHAPTER ELEVEN

Reading DATA

In this chapter, we'll be looking at three very useful additions to your programming vocabulary: READ, DATA, and RESTORE. They are used to get information stored in one part of the program to another part where it can be used.

Enter and run this program, which should make this a little clearer:

```
10 REM READ, DATA and RESTORE
20 DIM A(5)
30 FOR B = 1 TO 5
40 READ A(B)
50 PRINT A(B)
60 NEXT B
70 DATA 88,8965,23,-94,3
```

Using line 40, the program READs through the DATA statement in line 70 in order, printing up each item of DATA with line 50.

RESTORE returns the program to using the *first* item of DATA, as you'll discover if you modify the above program by adding line 55, so that it reads as follows:

```
10 REM READ, DATA and RESTORE
20 DIM A(5)
30 FOR B = 1 TO 5
40 READ A(B)
50 PRINT A(B)
55 IF B = 3 THEN RESTORE
60 NEXT B
70 DATA 88,8965,23,-94,3
```

It does not matter where in the program the DATA is stored. The computer will seek it out, in order from the first item of DATA in the program to the last, as our next program (which scatters the DATA about in an alarming way) convincingly demonstrates:

```
1 DATA 45
10 REM READ, DATA and RESTORE
20 DIM A(5)
22 DATA 888
30 FOR B = 1 TO 5
40 READ A(B)
50 PRINT A(B)
55 DATA 432
60 NEXT B
70 DATA 933,254
```

READ and DATA work just as well with string information:

```
10 REM READ/DATA with strings
20 FOR B = 1 TO 5
30 READ A$
40 PRINT A$
50 NEXT B
60 DATA test,one,nine,after,noon
```

Note that string DATA do not have to be enclosed within quotation marks, unless leading or trailing spaces and/or punctuation and symbols are significant and must be considered part of the DATA.

You can mix numeric and string DATA within the same program, as long as you take care to ensure that when the program wants a numeric item, a number comes next in the program, and when it wants a string item, it finds it:

```
10 REM READ/DATA with strings
20 FOR B = 1 TO 5
30 READ A$:READ A
40 PRINT A$,A
50 NEXT B
60 DATA test,12,one,989892,nine,3,after,
-892781,noon,23
```

CHAPTER TWELVE

Adding Life to Programs

The PCjr has many capabilities, and you should make the most of them. In this chapter we are going to take a simple program and elaborate it by adding such things as sound, flashing text, and graphics.

You'll find this is a painless way to introduce the capabilities. Once you've worked through this chapter, you should find it easy to apply the ideas to other programs you are writing.

The program we're going to use as the core of our development work is a "Duck Shoot" one, in which little objects fly across the screen, and you have to try and shoot them down.

In the first version of the program, the little objects are letters chosen at random, and you are the letter X. You fire at the "ducks" by pressing the F key. You move yourself left using the Z key and to the right with the M key. The **Caps Lock** key must be engaged before you run the program.

Although there is no time limit within the program, so you do not have to shoot all the ducks as quickly as you can, there is a limit on the number of shots you can fire. In three of the versions of this game in this part of the book, you'll see (line 50) that the program starts with a limit of 15 shots. In the last two, more complex, versions, you have 50 shots. The number of shots is deliberately kept low in the first versions so you will not be able to get a high score just by leaving your finger on the F key and waiting for the ducks to fly into the line of fire.

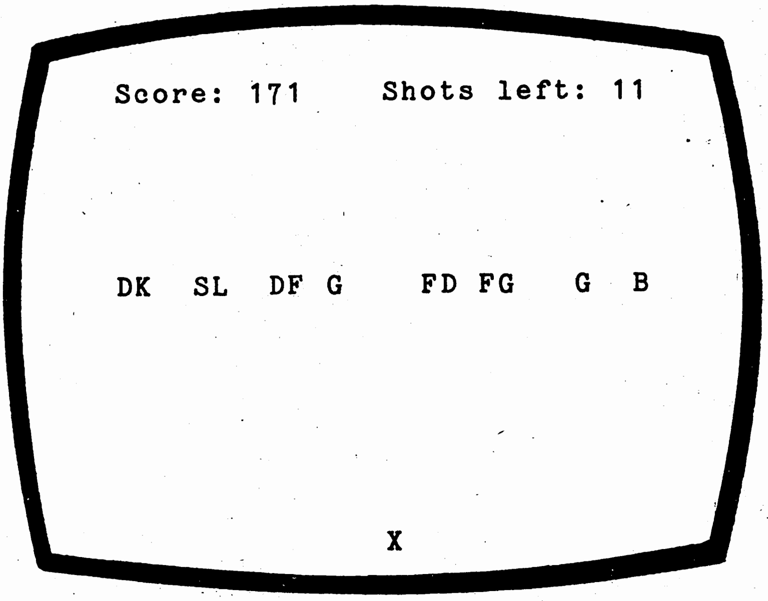
Here, then, is the first program. Type it into the PCjr and then run it:

```
10 REM DUCK SHOOT
20 CLS
30 DEFINT A - Z
40 SCORE = 0
50 SHOTS = 15
60 A$ = "ZAB    DK   SL   DF G    FD FGG
G"
70 ACROSS = 15
80 DOWN = 19
90 WHILE NOT K
100 LOCATE 11,1:PRINT A$
110 LOCATE DOWN, ACROSS -1:PRINT " X "
120 B$ = INKEY$
130 IF B$ = "F" THEN SHOTS = SHOTS - 1:I
F MID$(A$,ACROSS,1) <> " " THEN SCORE =
SCORE + 57:MID$(A$,ACROSS,1) = " "
140 LOCATE 5,1:PRINT "Score:"SCORE,"Shot
s left:"SHOTS"  "
150 IF SHOTS < 1 THEN LOCATE 10,1:PRINT
"That's the end of the game":END
160 IF B$ = "Z" THEN ACROSS = ACROSS - 1

170 IF B$ = "M" THEN ACROSS = ACROSS + 1

180 A$ = MID$(A$,2) + LEFT$(A$,1)
190 WEND
```

Here's an indication of what the screen looks like when it is up and running:



```
Score: 171     Shots left: 11



DK   SL   DF  G      FD  FG     G    B




                    X
```

You'll see the letters which are held in A$ (see line 60) moving across near the top of the screen. You (the X) will be further down the screen, about a quarter of the way across. You can—as I mentioned a few paragraphs ago—move yourself back and forth using the Z and M keys to get yourself into the position which you think gives you the best possible chance.

When you judge a "duck" is directly overhead, press the F key to fire your patented anti-duck missile. The number after the words "Shots left" near the top right-hand corner of the screen will decrease and, if you have been accurate, the number after the word "Score" will increase.

Note, by the way, that I have deliberately used *explicit names* for the variables within this program. That is, the variable name for the score is SCORE, for the shots it is SHOT, and for your position across the screen, the variable name is ACROSS. Even though it takes a little longer to type long variable names into a program, the advantages of using explicit names to keep the purpose of various parts of the listing clear outweighs the extra time it takes to type them in.

If, for example, you were writing a program like this, and you decided it would be better if the X was printed further down the screen, you would not have to search through the program to work out which variable held your "down" coordinate. If you had used explicit names, as in this case, you would find it very easy to locate the variable you were looking for.

Run the DUCK SHOOT program a few times, then return to the book for the first part of a discussion on it.

Line 60 defines the string variable A$ as a long series of letters and spaces. The letters can be anything you like; do not feel you have to copy mine. The important thing, however, is that the string is 32 characters long. You can check this by running the program briefly, stopping it, then typing in PRINT LEN (A$). If your string is the correct length, PRINT LEN (A$), followed by **Enter**, will give you the answer 32.

The appearance of movement given to the ducks is created by the string-handling commands which were explained a little earlier in the book. Refer back to this section now if you need to remind yourself how they work.

The vital line for movement is 180, which resets A$ equal to all the string without its first character—that is to MID$(A$,2)—and then adds to the very end of it the character of the string which was at the beginning, LEFT$(A$,1). The string is reprinted over and over again at 11,1 (see line 100, where LOCATE 11,1 is used to move the cursor to the position where we next wish to print) which is eleven lines down, and at the first position across the line.

Because the string is, in effect, being "shifted along" one character at a time before it is reprinted, the elments within the string appear to move smoothly along. Using strings in this way is one of the simplest ways to create smoothly moving graphics on the PCjr.

The string handling also makes it very simple to cause the shot duck to disappear from the sky. As the string is 32 characters long, each character "shot" can be referred to as MID$(A$,ACROSS,1).

Look at line 130. When the computer comes across an IF/THEN statement—as you know—it checks to see if it is true. If it finds that it is not true, then it moves along to the next line in the program, without bothering to carry out any further instructions which may be on the same line. Note that before the computer gets to line 130, it sets B$ equal to the key you are pressing (INKEY$). If the computer finds, at the start of line 130, that B$ does not equal F (as will be the case when you are not pressing the F key), then it proceeds to line 140, missing all the information and instructions which follow the IF B$ = "F" . . . line.

If, however, you are pressing F when the computer reads the keyboard, the PCjr continues working through the line, and reduces the value of the variable SHOTS by one. Then it hits another IF/THEN condition in which it looks at the element of A$ which is directly above the position of the X at that moment.

If line 130 discovers that this element of A$ is anything but a space, you have hit a duck, so the computer continues working through the line. The variable SCORE is incremented by 57 and, finally in line 130, that particular element of A$ is set to a blank, so the duck disappears.

Now, this takes some time to explain, but you'll find the computer does it apparently instantaneously. You press F, the score increases by 57 (if you're a good shot), the number of shots left drops by one, and the duck disappears. You'll see (line 150) that the game continues until you run out of shots, when the game terminates. Take note of your score at this point, and see if you can beat it on subsequent runs.

Once you have the program running to your satisfaction, and you have a pretty good idea of how it works, modify it to read like the following program. You do not have to NEW the computer. Just compare the program you have in your PCjr, line by line, with the next listing, and make any changes you need to, by adding and amending lines:

```
10 REM DUCK SHOOT - II
15 COLOR 0,7
20 CLS
30 DEFINT A - Z
40 SCORE = 0
50 SHOTS = 15
60 A$ = "ZAB    DK   SL   DF  G     FD FGG
G"
70 ACROSS = 15
80 DOWN = 19
90 WHILE NOT K
100 LOCATE 11,1:PRINT A$
110 LOCATE DOWN, ACROSS -1:PRINT " X "
120 B$ = INKEY$
130 IF B$ = "F" THEN SHOTS = SHOTS - 1:I
F MID$(A$,ACROSS,1) <> " " THEN SCORE =
SCORE + 57:MID$(A$,ACROSS,1) = " ":BEEP
140 LOCATE 5,1:COLOR 31:PRINT "Score:"SC
ORE,"Shots left:"SHOTS"  ":COLOR 0,7
150 IF SHOTS < 1 THEN LOCATE 10,1:PRINT
"That's the end of the game":COLOR 7,0:E
ND
160 IF B$ = "Z" THEN ACROSS = ACROSS - 1

170 IF B$ = "M" THEN ACROSS = ACROSS + 1

180 A$ = MID$(A$,2) + LEFT$(A$,1)
190 WEND
```

Now when you run this, you'll see an immediate and striking improvement. The entire screen is white with black writing, except for the

SCORE/SHOTS LEFT line which is flashing black and white. Line 15—COLOR 0, 7—produces the inverse image. In line 140, COLOR 31 causes the blink and highlight effect, and the COLOR 0,7 at the end of the line turns the color control back to where it was before, so the whole screen doesn't start flashing off and on.

The BEEP at the end of line 130 only sounds if you've actually hit a duck. Notice that the computer defaults to the standard white on black at the end of line 150 before ENDing.

Now let's have a look at the sound possibilities of this program.

Modify the listing so that it looks like this third version, run it a few times, and then we'll talk about the things it is doing:

```
10 REM DUCK SHOOT - III
15 COLOR 0,7
20 CLS
30 DEFINT A - Z
35 FOR G = 100 TO 1000 STEP 50: SOUND G,
 .5: SOUND 2000 - G, .5:NEXT G
40 SCORE = 0
50 SHOTS = 15
60 A$ = "ZAB    DK   SL   DF G     FD FGG
G"
70 ACROSS = 15
80 DOWN = 19
90 WHILE NOT K
100 LOCATE 11,1:PRINT A$:NOTE = 2*(ASC(L
EFT$(A$,1))):IF NOTE <> 64 THEN SOUND NO
TE,.5
110 LOCATE DOWN, ACROSS -1:PRINT " X "
120 B$ = INKEY$
130 IF B$ = "F" THEN SHOTS = SHOTS - 1:S
OUND 3700,.9:IF MID$(A$,ACROSS,1) <> " "
 THEN SCORE = SCORE + 57:MID$(A$,ACROSS,
1) = " ":SOUND SCORE, 3
140 LOCATE 5,1:COLOR 31:PRINT "Score:"SC
ORE,"Shots left:"SHOTS"  ":COLOR 0,7
150 IF SHOTS < 1 THEN LOCATE 10,1:PRINT
"That's the end of the game":COLOR 7,0:E
ND
160 IF B$ = "Z" THEN ACROSS = ACROSS - 1

170 IF B$ = "M" THEN ACROSS = ACROSS + 1
```

```
180 A$ = MID$(A$,2) + LEFT$(A$,1)
190 WEND
```

Immediately when you run the program, you will notice a difference. First, there is a dramatic double run of musical notes. This is produced by line 35. You'll recall, from chapter seven, that the SOUND command is followed by two numbers. The first one—which must lie between 37 and 32767—controls the frequency of the note which is produced, and the second one—which must be between 0 and 65535—controls the duration of the note. In line 35, the G which is the control variable of the loop, is the pitch sounded by the computer.

The material added to the end of line 100 ensures that each time a duck flies off the left-hand side of the screen, a note sounds. The pitch of the note is determined by the individual letter which is meant to be the duck at that point.

At the end of line 130, the BEEP has been replaced by a note which is related to the score, so the higher your score, the higher the note which is produced. As well, there is a high-pitched sound which you hear each time you press F, whether you hit anything or not.

Now these changes, as you can hear, enhance the program even more. However, we have not yet finished, as version IV shows:

```
10 REM DUCK SHOOT - IV
12 RANDOMIZE VAL(RIGHT$(TIME$,2))
15 COLOR 0,7
20 CLS
30 DEFINT A - Z
35 FOR G = 100 TO 1000 STEP 50: SOUND G,
 .5: SOUND 2000 - G, .5:NEXT G
40 SCORE = 0
50 SHOTS = 50
52 A$ = ""
54 FOR T = 1 TO 32
56 IF RND > .7 THEN A$ = A$ + CHR$(INT(R
ND*6) + 1) ELSE A$ = A$ + " "
60 NEXT T
70 ACROSS = 15
80 DOWN = 19
90 WHILE NOT K
100 LOCATE 11,1:PRINT A$:NOTE = 30 + 7*(
ASC(LEFT$(A$,1))):IF NOTE <> 254 THEN SO
UND NOTE,.5
105 LOCATE 12,1:PRINT A$
```

```
110 LOCATE DOWN, ACROSS -1:PRINT " ";CHR
$(8);" "
120 B$ = INKEY$
130 IF B$ = "F" THEN SHOTS = SHOTS - 1:S
OUND 3700,.9:IF MID$(A$,ACROSS,1) <> " "
 THEN SCORE = SCORE + 57:MID$(A$,ACROSS,
1) = CHR$(15):LOCATE 11,1:PRINT A$:LOCAT
E 12,1:PRINT A$:MID$(A$,ACROSS,1) = " ":
SOUND SCORE, 3
140 LOCATE 5,1:COLOR 31:PRINT "Score:"SC
ORE,"Shots left:"SHOTS"  ":COLOR 0,7
150 IF SHOTS < 1 OR A$ = "
                         " THEN LOCATE 10,1:PRI
NT "That's the end of the game":LOCATE 1
5,9:PRINT "Rating:"SHOTS*197:COLOR 7,0:L
OCATE 22,1:END
160 IF B$ = "Z" THEN ACROSS = ACROSS - 1

170 IF B$ = "M" THEN ACROSS = ACROSS + 1

180 A$ = MID$(A$,2) + LEFT$(A$,1)
190 WEND
```

Instead of letters, in this version the ducks are characters from the IBM PCjr's character set. Line 12 seeds the random number generator, and then the loop from 54 through to 60 creates a random duck pattern. You'll find this changes each time you run the program. As well as little characters forming the line of ducks, you'll see it is printed twice.

Another change—in line 130—gets the computer to turn a shot duck into a little starlike explosion before it vanishes. The number of shots (see line 50) has been increased to 50. You'll see in line 150 that the game will terminate if you run out of shots—as before—or if you manage to destroy all the ducks before you run out of shots. In this last case, you're given a "rating" (see end of line 150) before the program terminates. As well—see the end of line 150—the cursor is moved to 22, 1 (with LOCATE) before the program ends, to stop the terminating message from ruining the display.

By now, as you can see, your program listing is getting quite complex, as the program develops. You should follow a similar series of steps when creating your own programs: get a raw "basic" program up and running, and then add to it, and modify it, until you have created a masterpiece.

Both PCjr BASICs have great color possibilities, with Cartridge BASIC being more flexible than Cassette BASIC. Despite this, you can do a great deal with Cassette BASIC, as our next program—DUCK-SHOOT V—demonstrates:

```
10 REM DUCK SHOOT - V
11 SCREEN 1
12 RANDOMIZE VAL(RIGHT$(TIME$,2))
15 COLOR 1,0:REM SET BACKGROUND COLOR AND
 PALETTE
20 CLS
30 DEFINT A - Z
35 FOR G = 100 TO 1000 STEP 50: SOUND G, .5:
 SOUND 2000 - G, .5:NEXT G
40 SCORE = 0
50 SHOTS = 50
52 A$ = ""
54 FOR T = 1 TO 32
56 IF RND > .7 THEN A$ = A$ + CHR$(INT(RND*
 6) + 1) ELSE A$ = A$ + " "
60 NEXT T
70 ACROSS = 15
80 DOWN = 19
90 WHILE NOT K
100 LOCATE 11,1:PRINT A$:NOTE = 30 + 7*(ASC
 (LEFT$(A$,1))):IF NOTE <> 254 THEN SOUND
 NOTE,.5
105 LOCATE 12,1:PRINT A$
110 LOCATE DOWN, ACROSS -1:PRINT "  ";CHR$(8)
 ;" "
120 B$ = INKEY$
130 IF B$ = "F" THEN SHOTS = SHOTS - 1:SOUND
 3700,.91:IF MID$(A$,ACROSS,1) <> "
 " THEN SCORE = SCORE + 57:MID$(A$,ACROSS,1)
 = CHR$(15):LOCATE 11,1:PRINT A$:LOCATE 12,
 1:PRINT A$:MID$(A$,ACROSS,1) = " ":SOUND
 SCORE, 3
140 LOCATE 5,1:COLOR ,2:PRINT "Score:"SCORE,
 "Shots left:"SHOTS"  "
150 IF SHOTS < 1 OR A$ = "        " THEN LOCATE
 10,1:PRINT "That's the end of the game":
 LOCATE 15,9:PRINT "Rating:"SHOTS*197:COLOR
 7,0:LOCATE 22,1:END
160 IF B$ = "Z" THEN ACROSS = ACROSS - 1
```

```
170 IF B$ = "M" THEN ACROSS = ACROSS + 1
180 A$ = MID$(A$,2) + LEFT$(A$,1)
190 WEND
```

You'll see the screen turns a very attractive blue color (with the exact shade depending on your TV), with the letters and *ducks* in a lighter color (which looks vaguely yellow on *my* TV) printed on that blue background. Look carefully at the letters on the screen. They are quite different from the ones you've seen to date, and although a successful *shot* produces the same star effect as before, the ducks look quite different.

Get the listing back on the screen (or refer to the printed one) so we can discuss the changes between DUCKSHOOT IV and DUCKSHOOT V. (By the way, you'll probably find the on-screen listing easier to read and modify if you enter SCREEN 0 directly, before entering LIST). Line 11 sets SCREEN (which is set to zero if you do not specify any SCREEN) to one. There are seven possible screens (only zero through three are available in Cassette BASIC). SCREEN 0 (the one the computer defaults to if you do not specify a SCREEN) is the lowest mode, and is called *text mode*. SCREEN 1, the one we've used in DUCKSHOOT V, is the *medium resolution graphics mode*, in which up to four colors can be displayed at once. Line 15, as you can see from the REM statement, sets the background color and the *palette*.

Let me explain what those words mean in this context. The meaning of background color is pretty obvious. But what is this *palette*? The word palette is used, outside computer circles, as the name of the curved board artists squeeze their selection of paint colors onto. They choose the colors they actually paint with from this palette. And that's just what the PCjr does in this case.

The palettes allow us to choose any background color from the following list:

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

Save DUCKSHOOT V, then type in NEW and **Enter** the command, and enter the following program to see all the possible background colors:

```
10 REM BACKGROUND COLOR DEMO
20 SCREEN 1
30 FOR J=0 TO 15
40 COLOR J,1
50 CLS
60 BEEP
70 LOCATE 8,8:PRINT "BACKGROUND"J
80 FOR M=1 TO 500:NEXT M
90 NEXT J
100 SCREEN 0
110 LIST
```

The first number which follows the word COLOR (notice that line 20 has set screen 1) controls the background color. The second number controls the palette from which the foreground color will be chosen.

There are two palettes, called 0 and 1. Here they are:

PALETTE 0	PALETTE 1
1—GREEN	1—GREEN
2—RED	2—MAGENTA
3—BROWN	3—WHITE

In both palettes, color 0 is the background color. This next routine will run through the three foreground colors in each of the palettes:

```
10 REM FOREGROUND COLOR DEMO
20 SCREEN 1
30 FOR B=0 TO 15
40 FOR P=0 TO 1
50 FOR F=1 TO 3
60 COLOR B,P
70 CLS
80 LOCATE 8,8:PRINT "BACKGROUND"B
90 PRINT:PRINT TAB(8);"PALETTE"P
100 PRINT:PRINT TAB(8);"FOREGROUND"F
110 LINE (53,140)-(260,150),F
120 FOR M=1 TO 500:NEXT M
```

```
130 NEXT F
140 NEXT P
150 NEXT B
160 SCREEN 0
170 LIST
```

Here you see the PCjr going through its full spectrum of colors, with the jagged line in the bottom third of the screen showing the relevant foreground color. The actual hue depends on your TV. (On mine, it takes a bit of imagination to relate the actual color I see to its name.) You'll notice that if the background and foreground colors are the same (such as magenta on magenta) no jagged line is visible. You'll also notice that the exact shade of the foreground color is modified somewhat by color *bleeding* from the background color. This demonstration just hints at the enormous range of color possibilities you can get from your PCjr.

DISK SYSTEM ONLY

There are seven SCREEN modes available on a PCjr fitted with Cartridge BASIC (but only three in Cassette BASIC). Here's a program to demonstrate the effectiveness of SCREEN 3, the low resolution graphics mode.

```
10 REM SCREEN THREE DEMO
20 SCREEN 3
30 FOR B=1 TO 15
40 COLOR B,(16-B)
50 CLS
60 FOR F=0 TO 15
70 LOCATE 8,3:PRINT "FOREGROUND"F
80 PRINT:PRINT TAB(3);"BACKGROUND"16-B
90 LINE (10,120)-(120,120),F
100 LINE (10,130)-(120,130),F
110 LINE (10,140)-(120,140),F
120 FOR M=1 TO 500:NEXT M
130 NEXT F
140 NEXT B
150 SCREEN 0
160 LIST
```

As you can see, this screen supports 16 foreground colors, and produces big chunky letters which are both attractive and easy to read from a distance. You'll find this mode especially useful for programs that you want to show to a number of people at once. It is also a very useful mode for board games, as it makes more effective use of the screen than do the lower modes.

CHAPTER THIRTEEN

Getting Listed

An array is used when you want to create a list of items, and refer to the item by just mentioning the *position* within the list the item occupies. You set up an array by using the command DIM (for dimension). If you type in DIM A(20), the PCjr will set up a list in its memory called A, and will save space for twenty-one items: (A(0), A(1), . . . and so on . . . up to A(20). Each of these items—the A(7) and the rest—are called *elements* of the array.

When you *dimension,* or set up, an array, the computer creates the list in its memory and then fills every item in that list with a zero. So if you told your computer to PRINT A(3) it would print a 0. You fill the items in an array with a statement like A(2) = 1000, or by using READ and DATA as we saw in chapter eleven. Once you've given an element a value, you can get the computer to tell you what value the element has by saying PRINT A(*n*). You can also manipulate the element as if it were the number. That is, A(4)*6 is valid, as is 45 - A (6), and so on.

The PCjr will let you use an array of up to 11 elements (that is, A(0) through to A(10), or TEST(0) through to TEST(10)) without having to use the DIM statement first. The moment it comes across a reference to an element of an array, where the *subscript* (the number which follows in parentheses) is between 0 and 10, it automatically creates an array. However, it is good practice always to dimension arrays, even if you are using fewer than 12 elements.

You may like to *forget* about the element which has the subscript zero, and pretend that the array starts at one. Many times you'll find it simpler to assume DIM A(80) gives you an array of 80 elements (rather than 81 as is the case), and that the first element is A(1).

The first program in this chapter dimensions (sets up, or creates) an array called A with room for sixteen elements. We will ignore the element with the subscript 0. The B loop, from lines 40 to 60, fills the array with random digits between 0 and 9, and then prints them back for you with the loop from 70 to 100 (with a slight pause being created by line 90).

Here is the listing:

```
10 REM ARRAYS
20 DIM A(15)
30 CLS
40 FOR B = 1 TO 15
50 A(B) = INT(RND*9)
60 NEXT B
70 FOR Z = 1 TO 15
80 PRINT "A("Z") IS "A(Z)
90 FOR T = 1 TO 100:NEXT T
100 NEXT Z
```

And here's one example of it in use:

```
A( 1 ) IS  5
A( 2 ) IS  1
A( 3 ) IS  5
A( 4 ) IS  7
A( 5 ) IS  6
A( 6 ) IS  7
A( 7 ) IS  0
A( 8 ) IS  8
A( 9 ) IS  1
A( 10 ) IS  1
A( 11 ) IS  6
A( 12 ) IS  4
A( 13 ) IS  7
A( 14 ) IS  0
A( 15 ) IS  0
```

This is called a one-dimensional array, because a single digit follows the letter or name which labels the array.

You can also have multi-dimensional arrays, in which more than one number follows the array label after DIM. In our next program, for example, the computer sets up a two-dimensional array, called A again, consisting of five elements by five elements (that is, it is dimensioned by DIM A(4,4), as you can see in line 20):

```
10 REM MULTI-DIMENSIONAL ARRAYS
20 DIM A(4,4)
30 CLS
40 FOR B = 1 TO 4
50 FOR C = 1 TO 4
60 A(B,C) = INT(RND*9)
70 NEXT C
80 NEXT B
90 COLOR 0,7
100 PRINT "   1  2  3  4"
110 FOR B = 1 TO 4
120 COLOR 0,7:PRINT B;:COLOR 7,0
130 FOR C = 1 TO 4
140 PRINT A(B,C);
150 NEXT C
160 PRINT
170 NEXT B
```

When you run it, you'll see something like this:

```
   1  2  3  4
1  3  3  2  6
2  6  0  7  6
3  4  5  8  1
4  2  8  4  1
```

You specify the element of a two-dimensional array by referring to both its numbers, so the element 1,1 of this array (the element in the top left-hand corner of the preceding printout) is 3 and is referred to as A(1,1). The 8 in the printout is A(3,3), and the 4 below it is A(4,3).

Your computer also supports string arrays. Enter and run the following short program to see string arrays in operation:

```
10 REM STRING ARRAYS
20 DIM A$(5)
30 CLS
40 FOR B = 1 TO 5
50 A$(B) = CHR$(INT(RND*26) + 65)
           +CHR$(INT(RND*26)+65)
60 NEXT B
```

```
70 FOR B = 1 TO 5
80 PRINT "A$("B") IS "A$(B)
90 NEXT B
```

Here's one printout of this program:

```
A$( 1 ) IS NV
A$( 2 ) IS RD
A$( 3 ) IS EL
A$( 4 ) IS IZ
A$( 5 ) IS LL
```

You can, of course, fill the elements of an array—string or numeric—via DATA or INPUT statements. Here is a string array which is filled by a DATA statement:

```
10 REM STRING ARRAYS
20 DIM A$(5)
30 CLS
40 FOR B = 1 TO 5
50 READ A$(B)
60 NEXT B
70 FOR B = 1 TO 5
80 PRINT "A$("B") IS "A$(B)
90 NEXT B
100 DATA THINKING, IS, A, PAINFUL, TASK
```

This is the result of running it:

```
A$( 1 ) IS THINKING
A$( 2 ) IS IS
A$( 3 ) IS A
A$( 4 ) IS PAINFUL
A$( 5 ) IS TASK
```

ESCAPE FROM MURKY MARSH

The next program demonstrates the use of a two-dimensional array for "holding" an object, and for moving it around within the array. The object in this case is a little shape with a pointed top. The shape is trapped in a murky marsh, and by moving totally at random, it hopes one day to be able to escape from the marsh. The shape is free if it manages to stumble onto the outer rows.

(By the way, the shape in this program demonstrates Brownian motion, the random movement shown by such things as tiny particles in a drop of water when viewed under a microscope, or of a single atom of gas in a closed container. Brownian motion explains why a drop of ink gradually mixes into the water into which it has been placed.)

Here is the program listing:

```
10 REM
20 RANDOMIZE VAL(RIGHT$(TIME$,2))
30 DIM A(10,10)
40 COLOR 0,7
50 CLS
60 M = 0
70 GOSUB 300
80 WHILE Q<9 OR P<9
90 IF RND > .35 THEN P = P + 1 ELSE P = P
   - 1
100 IF RND > .35 THEN Q = Q + 1 ELSE Q = Q
    - 1
110 IF Q<1 THEN Q = 1
120 IF Q>10 THEN Q = 10
130 IF P<1 THEN P = 1
140 IF P>10 THEN P = 10
150 M = M + 1
160 LOCATE 3,1
170 PRINT TAB(20);"Attempt #"M"   "
180 A(P,Q) = 127
190 LOCATE 7,20
200 FOR X = 1 TO 10
210 FOR Y = 1 TO 10
220 PRINT CHR$(A(X,Y));" ";
230 NEXT Y
240 PRINT:PRINT TAB(20);
250 SOUND 37 + X*Y,.5
260 NEXT X
270 A(P,Q) = 46
280 WEND
290 GOTO 370
300 Q = INT(RND*3) + 4
310 P = INT(RND*3) + 4
320 FOR X = 1 TO 10
```

```
330 FOR Y = 1 TO 10
340 A(X,Y) = 254
350 NEXT Y,X
360 RETURN
370 PRINT:PRINT
380 PRINT TAB(20);"Whew...free at last!"
390 FOR T = 1 TO 3000:NEXT T
400 RUN
```

CHAPTER FOURTEEN

Time to Relax

You've been working pretty hard, and you've really earned a break. Therefore this final chapter will have nothing in it to study. Instead you'll just find three games to play with and against your PCjr. The programs won't take you long to enter, and are a lot of fun to play.

CITY BOMBER

In this first program, based on one written by Paul Toland, your plane's engine has stopped as you are flying over a large city. Your altitude is decreasing rapidly, and your only hope of landing in one piece is to flatten the city with your bombs. Due to a fault in your missile tracking system, only one bomb can be in the air at any one time.

You drop your bomb by pressing the **F** key.

A word of warning: This is an extremely difficult game to play. You'll have a great deal of trouble managing to win it, but you'll have a lot of fun trying.

Note how this program uses the SCREEN function to detect the top of the building—for both the falling bomb and your plane.

```
10 REM city bomber
20 GOSUB 340
30 COLOR 0,7
40 FOR Z = 2 TO 31
50 FOR J = 21 TO 14 + RND*10 STEP -1
60 LOCATE J,Z:PRINT CHR$(203)
70 NEXT J
80 NEXT Z
90 COLOR 7,0
100 X = 1:Y = 4
```

```
110 BY = -1: BX = X
120 LOCATE Y,X:PRINT " "
130 X = X + 1
140 IF X = 32 THEN Y = Y + 1:X = 1:IF Y = 25
 THEN 320
150 Q = SCREEN(Y,X)
160 IF Q = 203 THEN 320
170 LOCATE Y,X:PRINT CHR$(16)
180 W$ = INKEY$
190 IF (W$ = "F" OR W$ = "f") AND BY = -1
 THEN BY = Y:BX = X
200 IF BY = -1 THEN SOUND 40+RND*9,.5:GOTO
 120
210 LOCATE BY,BX:PRINT " "
220 BY = BY + 1
230 IF BY = 22 THEN BY = -1:GOTO 120
240 Q = SCREEN(BY,BX)
250 LOCATE BY,BX
260 IF Q <> 32 THEN PRINT " ":BY = -1:SOUND
 900,.5:SOUND 300,.5:SOUND 800,.5:GOTO 120
270 PRINT CHR$(25):SOUND 40 + RND*20,.5
280 GOTO 120
290 LOCATE 24,31:PRINT CHR$(16)
300 LOCATE 3,1:PRINT "Congratulations, you
 made it"
310 GOTO 330
320 LOCATE Y,X:COLOR 0,7:PRINT CHR$(16):
 LOCATE 3,1:PRINT "     You crashed and are
 now part":PRINT "     of the city skyline":
 SOUND 3000,6:SOUND 2000,5:SOUND 1000,4
330 COLOR 7,0:LOCATE 22,1:FOR Z = 1 TO 3:
 FOR T = 1000 TO 5000 STEP 500:SOUND T,1
 :NEXT:NEXT:END
340 REM initialise
350 SCREEN 0
360 CLS
370 DEFINT A - Z
380 KEY OFF
390 RANDOMIZE VAL(RIGHT$(TIME$,2))
400 FOR Z = 1 TO 4:FOR T = 1000 TO 5000
 STEP 500:SOUND T,1:NEXT:NEXT
410 RETURN
```

BAGATELLE

This is a simplified pinball machine in which you and the computer take turns rolling balls down a frame which contains a series of letters and numbers. Your score tends to increase each time you hit an obstacle on the way down the frame, and the ball will bounce off the obstacle, increasing your chances of hitting more obstacles.

If you look at the sample printouts, you'll see the numbers 1 to 9 at the top of the frame (by the way, the whole thing looks much more impressive on the screen—using inverse for the whole frame—than it does in this printout). You start your ball rolling down the frame from any one of these numbers.

```
  /  123456789   /
  /              /
  /   1  1  1    /
  /              /    Human 2025
  /  Z X Z X Z   /
  /              /  Machine 932
  /  7  7  7  7  /
  /              /
  /   7 7 7 7    /
  /              /
  /  5  5  5  5  /
  /              /
  /              /
  /   9   9   9  /  Ball #  4  Me
  /              /
  /  8   8   8   /
  /              /
  /////////////////
  /////////////////

I will start with 4
```

You and the PCjr have five balls each to roll down the bagatelle frame, and you take it in turns to do so. You have the first roll. Once your ball has made it to the bottom of the frame, the computer will choose a number from 1 to 9, tell you the number it has chosen, then roll its ball down the frame.

```
/  123456789   /
/              /
/   1  1  1    /
/              /   Human 2528
/  Z X Z X Z   /
/              /  Machine 2345
/  7  7  7  7  /
/              /
/   7 7 7 7    /
/              /
/  5  5  5  5  /
/              /
/              /
/   9   9   9  / The winner is
/              /
/  8   8   8   /  the human
/              /
////////////////
////////////////
```

```
10 REM Bagatelle
20 GOSUB 610
30 GOSUB 410
40 FOR Y = 1 TO 5:LOCATE 14,26:PRINT Y
50 FOR X = 1 TO 2
60 LOCATE 14,26
70 IF X = 2 THEN PRINT Y;" Me " ELSE PRINT
 Y;"You "
80 IF X = 1 THEN LOCATE 21,1:INPUT "Which
 number to start with ";K:IF K<1 OR K>9
 THEN 80
90 IF X = 2 THEN LOCATE 21,1:K = INT(RND*6)
 + 1:PRINT "I will start with"K:FOR T = 100
 TO 135:SOUND T,.5:FOR DELAY=1 TO 20:SOUND
 (1550-10*DELAY),.03:NEXT DELAY: NEXT T
100 LOCATE 21,1:PRINT STRING$(30," ")
110 LOCATE 1,2+K:PRINT K:FOR T = 1000 TO
 1005:SOUND T,.5:NEXT T
120 BA = K + 3:EBA = BA
130 BD = 1:EBD = 1
140 IA = 1
150 ID = 1
```

```
160 LOCATE EBD,EBA:PRINT " "
170 EBD = BD:EBA = BA
180 LOCATE BD,BA:PRINT "*"
190 FLAG = Ø
200 A = SCREEN(BD+1,BA)
210 IF A <> 32 THEN C(X) = C(X) + A:FLAG =
 1:GOTO 260
220 A = SCREEN(BD,BA+1)
230 IF A <> 32 THEN C(X) = C(X) + A:FLAG =
 1:GOTO 260
240 A = SCREEN(BD,BA-1)
250 IF A <> 32 THEN C(X) = C(X) + A:FLAG = 1
260 IF FLAG = 1 THEN SOUND 500,.5:IF RND <
 .3 THEN IA = - IA:ID = - ID
270 IF A = 27 THEN C(X) = C(X) - 27:A = 12
280 LOCATE 4,26:PRINT C(1):LOCATE 6,26:PRINT
 C(2)
290 BD = BD + (ID*RND) + .75
300 IF BA < 3 THEN IA = -IA:BA = 5
310 IF BD < 1 THEN ID = -ID:BD = 3
320 IF BA > 15 THEN IA = -IA:BA = 13
330 BA = BA + (IA*RND)
340 IF BD < 19 THEN 160
350 FOR G = 1500 TO 2000 STEP 35:SOUND G,.5:
 NEXT G
360 GOSUB 410
370 NEXT:NEXT
380 LOCATE 14,18:PRINT "The winner is    ":
 LOCATE 16,19
390 IF C(1) < C(2) THEN PRINT "the machine"
 ELSE PRINT "the human"
400 END
410 LOCATE 4,21:PRINT "Human"C(1)
420 LOCATE 6,19:PRINT "Machine"C(2)
430 LOCATE 14,19:PRINT "Ball #"
440 LOCATE 1,1:COLOR Ø,7:PRINT " 123456789 "
450 FOR Q = 1 TO 19
460 LOCATE Q,1:PRINT "/"
470 LOCATE Q,16:PRINT "/"
480 NEXT Q
490 LOCATE 18,1:PRINT "////////////////"
500 LOCATE 19,1:PRINT "////////////////"
510 COLOR 7,Ø
520 LOCATE 3,5:PRINT "1   1   1"
```

```
530 LOCATE 5,4:PRINT "Z X Z X Z"
540 LOCATE 7,4:PRINT "7   7   7   7"
550 LOCATE 9,5:PRINT "7 7 7 7"
560 LOCATE 11,4:PRINT "5   5   5   5"
570 LOCATE 14,5:PRINT "9    9    9"
580 LOCATE 16,4:PRINT "8    8    8"
590 RETURN
600 END
610 REM initialise
620 KEY OFF
630 SCREEN 0
640 CLS
650 DEFINT A - Z
660 RANDOMIZE VAL(RIGHT$(TIME$,2))
670 RETURN
```

FULL FATHOM FIFTY

The final program in this chapter is a simple dice game—FULL FATHOM FIFTY. The program uses two arrays (B and D), which we studied in the previous chapter, to store the players' scores and dice rolls respectively.

It is an easy game. You and the computer are in a race to roll a total of 40 or more. You roll two dice at once, but the only time you score is when both dice come up with the same number. When they do, you'll get three times the value of one of the dice added to your growing score.

You should know enough about how programs work at this stage to be able to determine what each section does, so I won't bother explaining this relatively simple program.

Get it up and running and then work out for yourself how each section works. You'll possibly gain more from the program by working it out yourself than having it explained in detail by me.

Here's the listing, then, of FULL FATHOM FIFTY, which ends this chapter.

```
10 REM FULL FATHOM FIFTY
20 RANDOMIZE VAL (RIGHT$(TIME$,2))
30 KEY OFF
40 SCREEN 0
50 CLS
60 ROUND = 1
70 FOR A = 1 TO 2
80 LOCATE 5,14:COLOR 15,0:PRINT "Full Fathom
 Fifty"
```

```
90 LOCATE 18,15:COLOR 15,0:PRINT "Round
 number"ROUND
100 LOCATE 8,17:PRINT"Junior"B(1)
110 LOCATE 11,17:PRINT "Human"B(2)
120 IF B(1) > 39 OR B(2) > 39 THEN 400
130 LOCATE 14,12
140 IF A = 1 THEN PRINT "I'll now roll the
 dice     ":GOTO 230
150 PRINT "Press 'R' to roll the dice"
160 IF INKEY$ = "" THEN SOUND RND*1000 +
 100,.5:GOTO 160
170 SOUND 40,0
180 LOCATE 14,12
190 PRINT "                              "
200 FOR K=500 TO 1000 STEP 25
210 SOUND K,.8:NEXT K
220 FOR H=1 TO 1000:NEXT H
230 FOR C = 1 TO 2
240 LOCATE 2,1:PRINT "Rolling die"C"
250 LOCATE 2,14:PRINT STRING$(34,CHR$(219))
 :FOR Q = 1 TO 62:SOUND 2500-7*Q,.03:NEXT
 Q:BEEP
260 D(C) = INT(RND*6) + 1
270 IF C = 2 AND RND >.7 THEN D(2) = D(1)
280 LOCATE 2,15:PRINT "It came up"D(C)
290 FOR Q = 1 TO 1300:NEXT Q
300 LOCATE 1,1:PRINT STRING$(39," "):FOR Q
 = 1 TO 200:NEXT Q
310 NEXT C
320 LOCATE 1,1:PRINT STRING$(39," ")
330 IF D(1) <> D(2) THEN LOCATE 14,12:PRINT
 "Those rolls do not count"
340 IF D(1) = D(2) THEN B(A) = B(A) + 3*D(1)
 :LOCATE 14,12:PRINT " The roll was"D (1)
 "and"D(1)
350 FOR Q = 1 TO 1500:NEXT Q
360 NEXT A
370 ROUND = ROUND + 1
380 FOR Q = 1 TO 1000:NEXT Q
390 GOTO 70
400 LOCATE 20,1:PRINT "The winner is the";
410 IF B(1) < B(2) THEN PRINT " human"
420 IF B(1) > B(2) THEN PRINT " PCjr"
```

[illegible]

Appendices

Using Math

Here is a summary of the major mathematical symbols on the PCjr:

Usual symbol	Computer symbol:
+ (plus)	+
− (minus)	−
× (multiply)	*
÷ (divide)	/
m^n	m ^ n

The mathematical functions include:

Computer word:	Meaning:
ATN	Arctangent
SIN	Sine
COS	Cosine
TAN	Tangent
INT	Reduce to next lowest whole number
SGN	Sign (returns −1 if negative, 0 if zero, 1 if positive)
ABS	Returns number without its sign (so ABS (−5) is 5)
SQR	Square root
LN	Natural log

Taking Care of Disks

There are a few common-sense rules for looking after your disks, which will ensure you get the maximum value out of them. It will also reduce the change of losing valuable data through damage to a disk.

If anything happens to the disk—such as it getting bent or having something spilled on the cardboard cover (and *not* on the disk itself)—repair the damage as best you can, and make a copy of the disk immediately. Then discard the original disk.

If you put a contaminated disk in the drive, you run the risk of damaging the read/write head.

Don't touch the part of the disk which shows through the slot in the outer cover, and don't put anything (such as cups of coffee, ashtrays or heavy objects like books) on top of either a disk, or the cardboard cover.

Magnetism can corrupt data on a disk. If you have anything which is, or could be, magnetized (such as a radio speaker, a pair of scissors, a telephone handset) keep it well away from your disks.

Disks, as I guess you've realized by now, are pretty sensitive creatures. They don't like heat too much, so leaving them inside a car can cause problems. (Disks work best in the range 50°F to 125°F, 10°C to 51°C; if your disk is exposed to temperatures outside this range, leave it for at least half an hour to adjust to room temperature before putting it in the disk drive.) You should never bend disks, or use rubber bands, adhesive tape, or paper clips on them.

If you want to write on the cardboard outer sleeve, take the disk out first. Write on the label which is stuck onto the disk itself before you stick it on. If you have to add something later, write gently, and with a felt-tipped pen. Do not use a ball-point pen or sharp pencil.

Glossary of Computer Words

Address - a number which refers to a location, generally in the computer's memory, where information is stored

Algorithm - the sequence of steps used to solve a problem

Alphanumeric - generally used to describe a keyboard, and signifying that the keyboard has alphabetical and numerical keys. A numeric keypad, by contrast, only has keys for the digits 1 to 9, with some additional keys for arithmetic operations, much like a calculator

APL - this stands for Automatic Programming Language, a language developed by Iverson in the early 1960s, which supports a large set of operators and data structures. It uses a nonstandard set of characters

Application software - these are programs which are tailored for a specific task, such as word processing, or to handle mailing lists

ASCII - stands for American Standard Code for Information Exchange. This is an almost universal code for letters, numbers, and symbols, which has a number between 0 and 255 assigned to each of these, such as 65 for the letter A

Assembler - this is a program which converts another program written in an assembly language (which is a computer program in which a single instruction, such as ADD, converts into a single instruction for the computer) into the language the computer uses directly

BASIC - stands for Beginner's All-purpose Symbolic Instruction Code, the most common language used on microcomputers. It is easy to learn, with many of its statements being very close to English

Batch - a group of transactions which are to be processed by a computer in one lot, without interruption by an operator

Baud - a measure of the speed of transfer of data. It generally stands for the number of bits (discrete units of information) per second

Benchmark - a test which is used to measure some aspect of the performance of a computer, which can be compared to the result of running a similar test on a different computer

Binary - a system of counting in which there are only two symbols, 0 and 1 (as opposed to the ordinary decimal system, in which there are ten symbols, 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9). Your computer "thinks" in binary

Boolean Algebra - the algebra of decision-making and logic, developed by English mathematician George Boole, and at the heart of your computer's ability to make decisions

Bootstrap - a program, run into the computer when it is first turned on, which puts the computer into the state where it can accept and understand other programs

Buffer - a storage mechanism which holds input from a device such as keyboard, then releases it at a rate which the computer dictates

Bug - an error in a program

Bus - a group of electrical connections used to link a computer with an ancillary device, or another computer

Byte - the smallest group of bits (see *bit*) which makes up a computer word. Generally a computer is described as being "eight bit" or "16 bit," meaning the word consists of a combination of eight or sixteen zeros or ones

Central Processing Unit (CPU) - the heart of the computer, where arithmetic, logic, and control functions are carried out

Character code - the number in ASCII (see *ASCII*) which refers to a particular symbol, such as 32 for a space and 65 for the letter "A"

COBOL - stands for Common Business Oriented Language, a standard programming language, close to English, which is used primarily for business

Compiler - a program which translates a program written in a high level (human-like) language into a machine language which the computer understands directly

Concatenate - to add (adding two strings together is known as "concatenation")

CP/M - stands for Control Program/Microcomputer, an almost universal disk operating system developed and marketed by Digital Research, Pacific Grove, California

Data - a general term for information processed by a computer

Database - a collection of data organized to permit rapid access by computer

Debug - to remove bugs (errors) from a program

Disk - a magnetic storage medium (further described as a "hard disk," "floppy disk," or even "floppy") used to store computer information and programs. The disks resemble, to a limited extent, 45 rpm sound records, and are generally eight, five and a quarter, or three inches in diameter. Smaller "microdisks" are also available for some systems

Documentation - the written instructions and explanations which accompany a program

DOS - stands for Disk Operating System (and generally pronounced "doss"), the versatile program which allows a computer to control a disk system

Dot-matrix printer - a printer which forms the letters and symbols by a collection of dots, usually on an eight by eight, or seven by five, grid

Double-density - adjective used to describe disks when recorded using a special technique which, as the name suggests, doubles the amount of storage the disk can provide

Dynamic memory - computer memory which requires constant recharging to retain its contents

EPROM - stands for Erasable Programmable Read Only Memory, a device which contains computer information in a semipermanent form, demanding sustained exposure to ultraviolet light to erase its contents

Error messages - information from the computer to the user, sometimes consisting only of numbers or a few letters, but generally of a phrase (such as "Out of memory") which points out a programming or operational error which has caused the computer to halt program executions

Field - a collection of characters which form a distinct group, such as an identifying code, a name or a date; a field is generally part of a record

File - a group of related records which are processed together, such as an inventory file or a student file

Firmware - the solid components of a computer system are often called the *hardware,* the programs, in machine-readable form on disk or cassette, are called the *software,* and programs which are hard-wired into a circuit, are called "firmware." Firmware can be altered, to a limited extent, by software in some circumstances

Flag - this is an indicator within a program, with the "state of the flag" (i.e., the value it holds) giving information regarding a particular condition

Floppy disk - see "disk"

Flowchart - a written layout of program structure and flow, using various shapes, such as a rectangle with sloping sides for a computer action, and a diamond for a computer decision, is called a flowchart. A flowchart is generally written before any lines of program are entered into the computer

FORTRAN - a high-level computer language, generally used for scientific work (from FORmula TRANslation)

Gate - a computer "component" which makes decisions, allowing the circuit to flow in one direction or another, depending on the conditions to be satisfied

GIGO - acronym for "Garbage In Garbage Out," suggesting that if rubbish or wrong data is fed into a computer, the result of its processing of such data (the output) must also be rubbish

Global - a set of conditions which affects the entire program is called "global," as opposed to "local"

Graphics - a term for any output of computer which is not alphanumeric or symbolic

Hard copy - information dumped to paper by a printer

Hardware - the solid parts of the computer (see *software* and *firmware*)

Hexadecimal - a counting system much beloved by machine code programmers because it is closely related to the number storage methods used by computers, based on the number 16, as opposed to our "ordinary" number system, which is based on 10

Hex pad - a keyboard, somewhat like a calculator, which is used for direct entry of hexadecimal numbers

High-level languages - programming languages which are close to English. Low-level languages are closer to those which the computer understands. Because high-level languages have to be compiled into a form which the computer can understand before they are processed, high-level languages run more slowly than do their low-level counterparts

Input - any information which is fed into a program during execution

I/O - stands for Input/Output port, a device the computer uses to communicate with the outside world

Instruction - an element of programming code, which tells the computer to carry out a specific task. An instruction in assembly language, for example, is ADD, which (as you've probably guessed) tells the computer to carry out an addition

Interpreter - converts the high-level ("human-understandable") programs into a form which the computer can understand

Joystick - an analog device which feeds signals into a computer which is related to the position which the joystick is occupying; generally used in games programs

Kilobyte - the unit of language measurement; one kilobyte (generally abbreviated as K) equals 1024 bits (see *bit*)

Line printer - a printer which prints a complete line of characters at one time

Low-level language - a language which is close to that used within the computer (see *high-level language*)

Machine language - the step below a low-level language; the language which the computer understands directly

Mainframe - the term for "giant" computers such as the IBM 370. Computers are also classified as minicomputers and microcomputers (such as the computer you own)

Memory - the device or devices used by a computer to hold information and programs being currently processed, and for the instruction set fixed within a computer which tells it how to carry out the demands of the program. There are basically two types of memory (see *RAM* and *ROM*)

Microprocessor - the "chip" which lies at the heart of your computer. This does the "thinking"

Modem - stands for MOdulator/DEModulator, a device which allows one computer to communicate with another via the telephone

Monitor - (a) a dedicated television screen for use as a computer display unit, containing no tuning apparatus; (b) the information within a computer which enables it to understand and execute program instructions

Motherboard - a unit, generally external, which has slots to allow additional "boards" (circuits) to be plugged into the computer to provide facilities (such as high-resolution graphics, or "robot control") which are not provided with the standard machine

Mouse - a control unit, slightly smaller than a box of cigarettes, which is rolled over the desk, moving an on-screen cursor in parallel to select options and make decisions within a program. "Mouses" work either by sensing the action of their wheels, or by reading a grid pattern on the surface upon which they are moved

Network - a group of computers working in tandem

Numeric pad - device primarily for entering numeric information into a computer, similar to a calculator

Octal - a numbering system based on 8 (using the digits 0, 1, 2, 3, 4, 5, 6, and 7)

On-line - device which is under the direct control of the computer

Operating system - this is the "big boss" program or series of programs within the computer which controls the computer's operation, doing such things as calling up routines when they are needed and assigning priorities

Output - any data produced by the computer while it is processing, whether this data is displayed on the screen or dumped to the printer, or is used internally

PASCAL - a high-level language, developed in the late 1960s by Niklaus Wirth, which encourages disciplined, structured programming

Port - an output or input "hole" in the computer, through which data are transferred

Program - the series of instructions which the computer follows to carry out a predetermined task

PILOT - a high-level language, generally used to develop computer programs for education

RAM - stands for Random Access Memory, the memory on board the computer which holds the current program. The contents of RAM can be changed, while the contents of ROM (Read Only Memory) cannot be changed under software control

Real time - when a computer event is progressing in line with time in the "real world," the event is said to be occurring in real time. An example would be a program which showed the development of a colony of bacteria which developed at the same rate that such a real colony would develop. Many games, which require reactions in real time, have been developed. Most "arcade action" programs occur in real time

Refresh - the contents of dynamic memories (see *memory*) must receive periodic bursts of power in order for them to maintain their contents. The signal which "reminds" the memory of its contents is called the refresh signal

Register - a location in computer memory which holds data

Reset - a signal which returns the computer to the point it was in when first turned on

ROM - see *RAM*

RS-232 - a standard serial interface (defined by the Electronic Industries Association) which connects a modem and associated terminal equipment to a computer

S-100 bus - this is also a standard interface (see *RS-232*) made up of 100 parallel common communication lines which are used to connect circuit boards within microcomputers

SNOBOL - a high-level language, developed by Bell Laboratories, which uses pattern recognition and string manipulation

Software - the program which the computer follows (see *firmware*)

Stack - the end point of a series of events which are accessed on a last in, first out basis

Subroutine - a block of code, or program, which is called up a number of times within another program

Syntax - as in human languages, the syntax is the structure rules which govern the use of a computer language

Systems software - sections of code which carry out administrative tasks, or assist with the writing of other programs, but which are not actually used to carry out the computer's final task

Thermal printer - a device which prints the output from the computer on heat-sensitive paper. Although thermal printers are quieter than other printers, the output is not always easy to read, nor is the paper used easy to store

Time-sharing - this term is used to refer to a large number of users, on independent terminals, making use of a single computer, which divides its time between the users in such a way that each of them appears to have the "full attention" of the computer

Turnkey system - a computer system (generally for business use) which is ready to run when delivered, needing only the "turn of a key" to get it working

Volatile memory - a memory device which loses its contents when the power supply is cut off (see *memory, refresh, ROM,* and *RAM*)

Word processor - a dedicated computer (or a computer operating a word processing program) which gives access to an "intelligent typewriter" with a large range of correction and adjustment features

ABOUT THE AUTHOR

Tim Hartnell has written more than 30 books on personal computers, and owns a computer publishing company, called Interface Publications, based in London. He is the author of a number of programming guides, including *How to Program Your Apple IIe* and *How to Program Your Commodore 64* as well as *Tim Hartnell's Giant Book of Computer Games* (all published by Ballantine Books). He lives in Australia.

TIM HARTNELL TAKES THE BYTE OUT OF COMPUTER PROGRAMMING

Now you can learn to program your IBM PCjr in a matter of hours. Tim Hartnell takes you by the hand and leads you step-by-step in this complete guide for the first-time programmer.

You start to program your IBM PCjr by writing and playing games, so you have fun while you're learning.

In Chapter One, you'll learn how to run your disk operating system, save, name, and erase programs, and copy disks.

By the end of Chapter Three, you'll run your first program and take control of your screen and printer.

You go on to read, write, and compare data, to program complex mathematical functions, and use sound and graphics to enhance your programs.

Finally, you'll be able to create your own programs and change existing ones.

Through dozens of game programs—including solitaire, pinball, and Go—Tim Hartnell helps you understand, enjoy, and feel comfortable with your computer.

Computer/Programming

31945

0 70999 00695

ISBN 0-345-31945-1

Cover printed in USA