

Easy Programming for the Atari Micros

Eric Deeson

SHIVA's
friendly
micro
series



Easy Programming for the Atari Micros

Eric Deeson

Joseph Chamberlain College, Birmingham



Shiva Publishing Limited

SHIVA PUBLISHING LIMITED
64 Welsh Row, Nantwich, Cheshire CW5 5ES, England

© Eric Deeson, 1984

ISBN 1 85014 022 7

Figures 2.5, 5.1 and 18.1 courtesy of Atari International (U.K.) Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording and/or otherwise, without the prior written permission of the Publishers.

This book is sold subject to the Standard Conditions of Sale of Net Books and may not be resold in the UK below the net price given by the Publishers in their current price list.

Typeset and printed by Devon Print Group, Exeter

Contents

	Foreword	
	Introduction	
	Acknowledgements	
1	Open the box! (Setting up, Getting on with it)	5
2	Starring Atari (A program, What is a computer?, The parts of a computer, Efficient chips)	10
3	In and out (Taking command, Light program, Mind your language, Input-process-output, Atari input-process-output, Processing, Summary, Do it yourself)	16
4	Printing press (Looping the loop, A la mode, Today's text, Printed matter, Tab, All under control, Your own thing, Modus Operandi, Summary, Do it yourself)	37
5	World record (The Atari cassette recorder, Saving your gem, LOAD off your mind, Mergency, Your tape library, Staking your claim, Summary, Do it yourself)	62
6	The master plan (Planning)	72
7	Full of sound and fury (Theory, Atari sound, Into practice, Summary, Do it yourself)	74
8	Just a plot (Mode 3, Modes 5 and 7, Modes 4 and 6, Mode 8, Summary, Do it yourself)	83
9	Taking command (Commanding height. The Editor's chair, Squashing programs, Structure?, Oddments, Summary)	93

10	Valley of decision (Implied decisions, IF...THEN, More IFs and buts, Let's be logical, Conclusion, Summary, Do it yourself)	102
11	Get the picture (Planning, Program pictures, Top-down development, Ideas to modules, Modules to flowchart boxes, Coding the program, Do it yourself)	110
12	Yellow subroutine (GOSUB...RETURN, Summary, Do it yourself)	121
13	Numbering our days (Arithmetic, Number-crunching, Functions, Summary, Do it yourself)	128
14	Heart strings (What is a string?. String operations, Substrings, String functions, Summary, Do it yourself)	137
15	Bug in a rug (A bugged program, Tracing, Some closing tips, Moral)	144
16	Graphic description (Modes, Graph plotting, Get your fill, Modes 9–11, Summary, Do it yourself)	150
17	Graphics descriptions (Spritely work, DIY characters)	159
18	Ragbag (Timing, Interfacing, The special function keys)	167
19	Hip hip array! (A simple array, Search and sort, Complex arrays)	174
20	Onward!	179
	Appendix 1: Comments on questions	180
	Appendix 2: Some more programs	183
	Appendix 3: BASIC survey	188
	Appendix 4: PEEK-a-boo	194
	Appendix 5: The Atari character set	198
	Appendix 6: To err is human	204
	Appendix 7: Tipples	209
	Appendix 8: Resources	215
	Index	217

Foreword

When the founders of Atari created their first games back in 1972 they started to bridge the gap between the public and computer technology. Up until then, development of computers had been something shrouded in mystery. But since the arrival of those pioneering games, people everywhere have become fascinated by the capabilities of this new technology.

In the spirit of those individuals who started the ball rolling, Eric Deeson has written a friendly and informative guide to help you harness the power at your fingertips, by learning to program your Atari Home Computer.

I hope you find it useful and informative, and that it will help you enjoy your Atari computer to the full.

Andrew Swanston
Marketing Director
Atari International (UK) Inc

Introduction

Atari have welcomed you to your micro, and it may well be that you've had a good play with it. Perhaps you've even got a cassette or cartridge with a program on that guides the machine to let you enjoy some smashing game, or get your accounts together.

Computers like the Ataris can do all kinds of things like that and of course, that's what most users—there are millions of us now—spend their keyboard or joystick time on.

Yet each program you try out, whether it's on cassette or cartridge (or even on disc if you can use discs), is the result of someone's work. Someone sat down at a keyboard for tens or hundreds of hours and made up the instructions that tell the machine what to do. Doing that is called programming. It takes a lot of time; it takes effort to pick up; it can be huge fun; it can increase your income; it can lead you to new friends and a new way of life. Yes, programming can be a hobby like going to the races, painting in water colour, or playing squash.

To me there are few things more satisfying than spending a few evenings with a micro and ending up with a program that makes the machine do something I want in the way I want it. Well, perhaps getting a cheque for a few pounds from a computer magazine for the program is even more satisfying.

Having a micro at home, school or work and not being able to tell it to do even simple things seems a real shame in my view. All that fun you're missing! So in this book I try to give you the main ideas of making your micro your servant. Give it a go and you may gain an absorbing hobby that you'll enjoy the rest of your life. And if it makes you rich, well and good—if riches are what you want. Send my 10% to charity. . . .

This book is only the start, of course. I hope it appeals to you as a way of learning how to program your Atari (any model), and to do so with great enjoyment. Take as long as you like over it—this isn't a school book to be got through in a week. And when you've finished with it, I hope you'll find it worth keeping for reference in the future. There are a number of appendices just for the purpose, and a long index to help you get round the pages.

Two warnings, though. It may be that you don't after all find programming magic fun. OK, then—it doesn't matter. Not everyone is turned on by TV, collecting ferns or reading Dostoyevsky in Russian. But give it a try, and even if you don't click with program-writing as a hobby, you'll still be able to control your machine better.

On the other hand you may become hooked. You may lose family and friends, sell the car to buy a disc drive, get arthritis in your finger-joints from too much keyboard work. Micros can be highly addictive. Maybe the charity you send the 10% to should be one that helps the micro-junkies.

There are lots of programs in this book. I've written most of them with one purpose—to show you some technique or other. That fact, plus the restriction on space, means that not all are super-duper products that you'll want to use time and again. But this is *not* a book of listings. I find such books boring as they imply you're a person who doesn't care about *why* this is done and not that, or *how* a certain effect comes about. Of course you should look at such books, and at program listings in magazines—but do so with a basic knowledge and then you'll be able to improve them your own way.

That's why there are also lots of things for you to try yourself in this book as well—as you explore, your knowledge will deepen and the fun will increase. Enough of introducing. Let's get on with the book. Enjoy it.

Harborne, February 1984

Eric Deeson

Note: A box ☐ is used to represent a space when one is not otherwise obviously needed. Italic text in printouts denotes the use of inverse.

Acknowledgements

I'm not quite a computer junkie, but I daren't find out if I could manage more than a few days way from my micros. So I must start by sending guilty thanks to the family and friends who had less of my company than they should while I was working on this tome. In fact, I'll go further, and dedicate this book to all my friends (which includes the family).

The work on the tome is not quite all mine (even if the responsibility is). I couldn't have done so much without the kind help of folk at Atari (particularly Jon. Dean who did a great deal) and at the Birmingham branch of Lasky's. Atari supplied the high quality photos (blame me for the rest) and helped with the technical queries as well as my BASIC uncertainties.

Indeed I guess Atari deserves acknowledgement for making this book possible. It's not just that without Atari micros I wouldn't be writing about Atari micros. If Atari hadn't brought us the concept of home micro-based video games (their Pong, which I remember well, in fact started in pubs) perhaps there wouldn't now be many million micro-users.

I would also like to thank Dave Mackin of Hodder Educational for an important contact, the staff of Micro General of Reading who sorted out the interface between my Atari computer and my non-Atari printer and all the cheerful friendly folk at Shiva Publishing. The typing—no, I don't much word-process—was carried out with speed and efficiency by Lynne Nicholls. She claimed it was an adventure. I trust she meant it.

Thanks, everyone. And thank *you* too, dear reader, for buying the book. Well, I *hope* you bought it. I *am* to blame for errors and such, so do send in your comments.

P.S. Can I have a second dedication? The weekend I worked on the final proofs of this book, my brother's family were staying with us. Turns out they bought an Atari micro, so I'd like to dedicate the book to them—Pat, Allan, Stuart and Katie. Then they'll tell all their friends to buy. . . .

I Open the Box!

No, this isn't a perfect world. If it *were* a perfect world, you would now be on the verge of a quite painless entry into computing, a marvellous hobby that has already trapped millions of folk around the globe. Well, you *are* on the verge of the hobby, and it *is* a marvellous one, but I fear that the entry won't be quite painless. I'll do my best to smooth the path, of course.

So you have just got your new micro together with a copy of this book, and you have read this far without even unpacking the box!

Let's pretend that that is the case. You have in front of you this book, open at this page, and the great big exciting box that contains (you hope) your new Atari micro-computer. We'll unpack the box together, because *I* have just got my Atari too. It happens to be a 600 XL, but there's little difference here or on any other page between this and any of the company's other machines—the older 400 and 800, the 800 XL (same as mine but able to store more information) or the 1450 XLD, not yet available as I write this. (Most of this book is about Atari BASIC, you will need the BASIC cartridge if you are using a 400 or 800 micro.)

We'll start, then, by opening the box together, stopping for a moment to admire the pretty colours on it and perhaps to read the posh-sounding but rather incomprehensible descriptions. Inside my box is another box, white foam this time, a colour booklet telling us briefly what we've bought or been given and less briefly how we can spend thousands

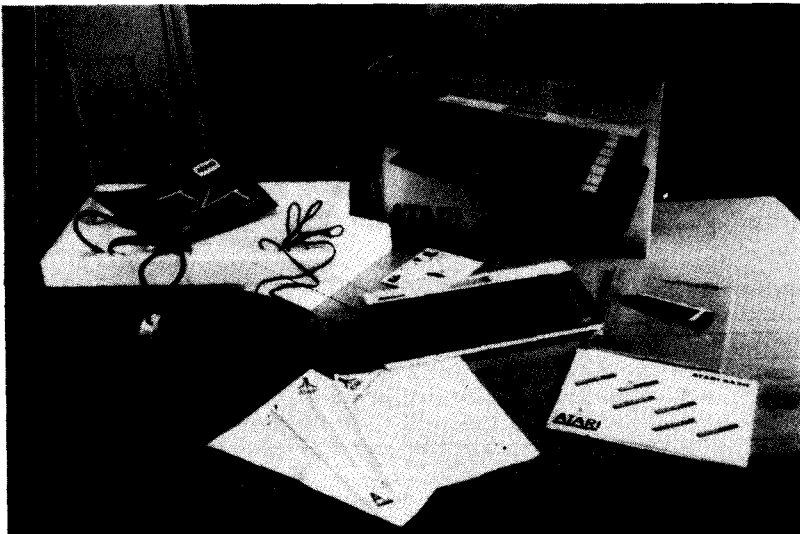


Figure 1.1 Atari 600XL.

of pounds more on feeding the machine with bits and bobs. Put that booklet aside for now, perhaps on a nearby table to impress family and friends.

Also in my box is a white envelope. Contents: a sheet of rather worrying notes on using your micro with a TV set (which you are going to have to do); guarantee card; a booklet detailing service centres (hmm, that's a bit worrying too!). Put the sheet aside—this chapter replaces it; put the service booklet aside as well, hoping you'll never need it. Deal NOW with the guarantee card just in case; I permit you to open the white box to find your micro's serial number—it's on a label on the base of the machine. Now we can officially open the white box. . . .

Contents check list

- 1 beautiful keyboard in plastic bag (dispose of safely, please)—that's your micro.
- 1 long black lead with TV-type plugs at each end.
- 1 incredibly heavy power supply unit with, phew, mains plug-top.
- 1 rather small booklet of notes in ten million languages: this chapter replaces that, too.
- 1 even smaller booklet on using the micro with your TV set; again this chapter should help.
- 1 more booklet—ATARI BASIC reference guide with (if mine's anything to go by) a fair number of errors: see the rest of this book.
- 1 sheet of "Important Software Information": I'll cover that as well.

And that leaves an empty box and a room scattered with objects and papers.

I suggest that for the time being you use the boxes to keep your equipment in when it isn't in use (you *must* sleep sometimes). And of course, if you don't keep your precious micro packed away when you have to go out, you are bound to find bits getting lost, damaged or chewed by the dog. Anyway, it's a nice box and you *may* need it one day to send your micro back in for repair. So, for a start, put all the paperwork in the base of the white box, put on the lid—and mark the lid clearly TOP, scratching it with a pencil in great big letters. That'll save accidents sooner or later!

SETTING UP

Action check list

- this book, this page
- the micro (the keyboard thing)
- the power supply unit (jargon-ready folk call this the PSU)
- the black TV lead
- a TV set—colour or black-white, 625 line model
- (alternative to the last two is a video monitor, for which you'll need a special lead)
- a table
- a chair

Step 1 Get yourself together

The best arrangement for starters is to have a table to yourself near a mains power point and to sit at it with the micro and TV set on the top laid out in easy comfortable reach. To do that you need a forgiving family and a lot of room. Maybe for now you'll have to send the others out to take the dog for a walk and spread yourself in front of the telly in the living room. A low table and a cushion on the floor will do. You can use a mains adaptor safely if your power point cannot manage TV set and micro PSU. Please take care with leads trailing over the floor. You won't like it if your aged grandma trips over one and brings all the costly gear tumbling down. She might even hurt herself as well.

If you are more methodical than anyone I know and want to set your computer corner up now—do so, of course. I deal with all this in Chapter 5, so feel free to check through there when you wish. Otherwise you'll just have to train those you live with in the new

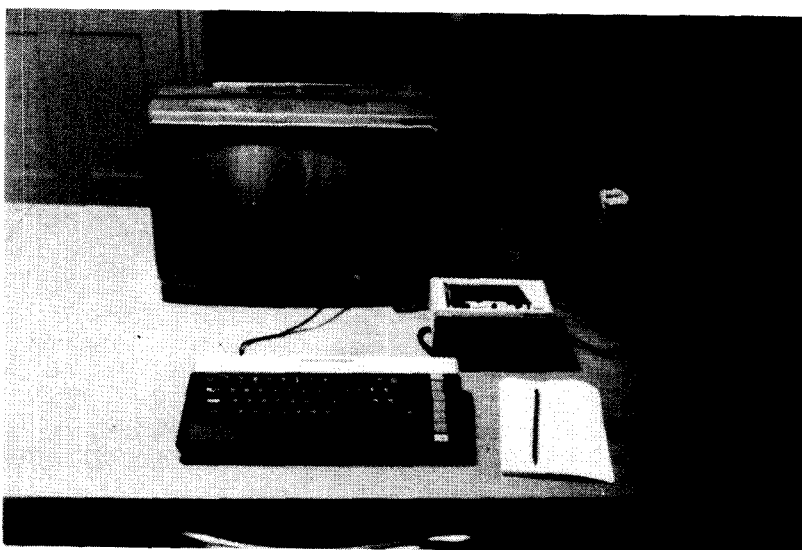


Figure 1.2 Starting at home.

priority: COMPUTING IS BETTER FOR YOUR BRAIN THAN SLUMPING IN FRONT OF THE TV WATCHING BROADCASTS! If you get to like programming—and I hope you will—be prepared for many hours sweating over a hot keyboard. Comfort is crucial.

Step 2 Give the micro energy

This is about the hardest thing in the whole book, if my experience is anything to go by. You need to spend a good few minutes untangling the mains cord of the power supply unit. Plug that into the mains (without switching on if you have the choice). Next unravel the thin wire the other side of the hefty transformer unit. At the end of this wire is (for some reason) a 7-pin DIN plug, that fits into the DIN socket with seven holes marked PWR IN at the back of the keyboard. (It won't fit into the 5-pin DIN socket thoughtfully placed next to it—but *don't* try or you'll cause damage!)

Switch on the mains and switch on at the back of the micro. The (very useful) POWER lamp at the bottom right of the keyboard will shine brightly to show you're on the right track. If you are.

Step 3 Prove it on screen

Connect the black video lead now. This has a 'co-ax' plug at each end. Push the one with the longer central spike firmly into the TV socket at the back of the computer; push the other plug firmly—but very gently—into the aerial socket of your telly. Of course you'll need to take out the existing aerial plug first.

WARNING If the TV set is often used for broadcast programs and computing, the aerial socket will start falling to bits. Collect a few bob from the family for a special adaptor that lets two inputs—aerial and micro—feed into the one socket. You can get adaptors like this from most TV and electronic shops.

Switch on the TV set. Unless it's 3 o'clock in the morning, you'll get on screen the TV program your family was watching before you turfed them out. Not a very good picture because you're not using the proper aerial. Select a spare channel on the set and tune it to BBC 2 (re-use the proper aerial lead if this helps). Now tune the channel (with the computer video plug in again) away from BBC 2 and the other broadcast channels. Within a few moments you should find the micro telling you on screen that it's there. You'll see something like Figure 1.3. Gently tune your Atari channel to get the best

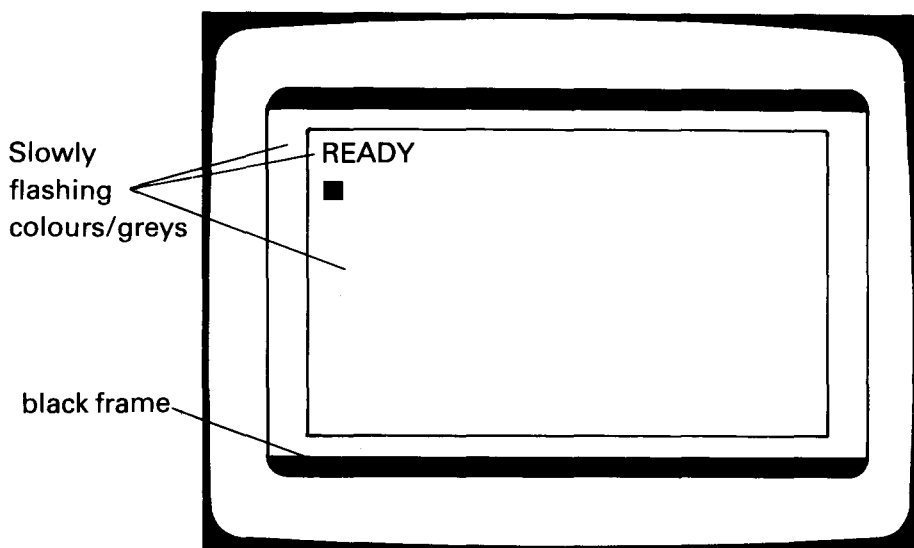


Figure 1.3

possible screen display—the clearest colours (grey shades if you're using black and white), the sharpest edges and lettering. Adjust all your TV controls now to improve matters further—brightness, colour, contrast, horizontal/vertical positions. Keep sound low if there's any annoying buzz at the best screen picture setting—but don't turn the sound off, as you'll be needing it.

Step 4 Test

Switch off the micro (get used to using the switch at the back of the box) and switch on again. You'll get a blank screen for a moment, then an ugly burping sound from the TV speaker, and finally Figure 1.3 again.

Now say 'goodbye' to the machine! What? Don't ask me why. Anyway, type BYE on the keyboard (getting a Space Invader beep from the speaker with each letter) and press the RETURN key on the right hand side of the board. The RETURN key tells the

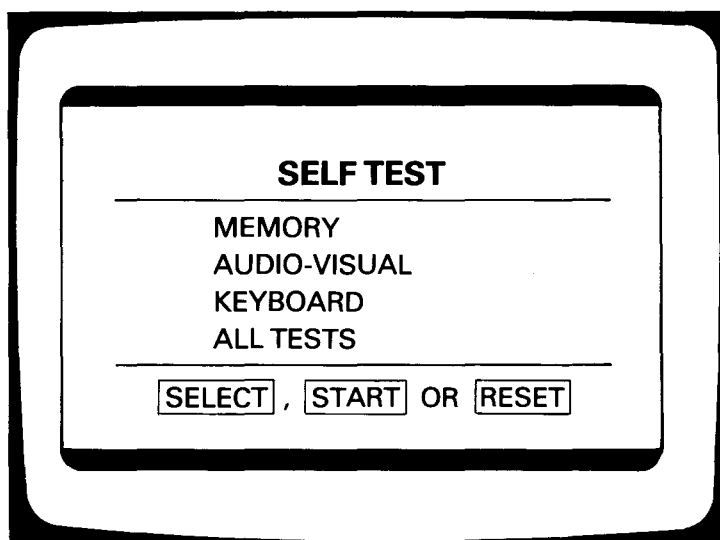


Figure 1.4

computer you have finished your message. At the moment the message is BYE, not BYE-BYE or BYELAW, so press RETURN after those three characters. Figure 1.4 shows what you should get now—it's the Atari self-test 'menu'. Press the SELECT key at the right hand side of the board three times, so that the message 'ALL TESTS' flashes on the screen, and press START to tell Atari to get testing. Taking its time, the machine laboriously tests memory, sound and keyboard. You can follow what's happening on Page 4 of the *Owner's Guide*, but I guess it is unlikely you'll find problems now or in the future. Store the testing routine in the back of your mind for the day you really feel something has gone wrong. I hope that day never comes for you!

GETTING ON WITH IT

Let's be a bit more ambitious. Try this set of 'commands' as your own self-test; the (R) at the end of each command is short for 'Press the RETURN key'.

```
SETCOLOR 1, 0, 0 (R)
SETCOLOR 2, 2, 10 (R)
SETCOLOR 4, 3, 0 (R)
PRINT CHR$(125) (R)
POSITION 10, 10 (R)
PRINT "Testing - 1, 2, 3!" (R)
```

Self-test? Yeah, well, if you managed all that perfectly, without a slip, you're either a genius or well used to typing commands to computers. Maybe I'd better say something about the keyboard now? More detail later.

The Atari keyboard's not *too* different from that of an ordinary typewriter, numbers at the top, letters in the middle, symbols at the right, and the long 'space' bar at the bottom. When pressed, either of the SHIFT keys (there's one on each side at the bottom) gives the characters on the top half of the number and punctuation keys. Thus to get " or " (they're the same to the computer), press SHIFT and key 2; I'll call that SHIFT and 2. In the same way you can get the \$ from key 3. SHIFT does not work as on a typewriter for capital and small letters however. Normally you get CAPITALS; to get small letters ('lower case') press the CAPS key. Logical? Now SHIFT and a letter key *will* give you the capital ('upper case') form. Get back to standard upper case output by having another go at CAPS.

After RETURN, if your typing's anything like mine, the most important key is called DELETE BACK SPACE. There it is on the top row. If you make a mistake, pressing that key will wipe out the last character typed and move the 'cursor' (the solid square) back one place. Pressing this key with SHIFT down is even more drastic—it wipes the whole line out of your life for ever.

What else? Oh yes—be careful to distinguish between 0 (the number zero on the top row) and O (the capital letter oh, just below it), and between 1 (the number one, top row) and I (capital eye, next to O) and l (lower case ell, below O). You may be able to tell which is which in any context, but the micro isn't so bright—give it the wrong one and it will get very upset. 2 and Z and 5 and S may cause problems of the same kind sometimes. Did you get your Atari telling you off with ERROR? Tut, tut.

Well, that is *all* rather complex. You'll soon get to know your way round the Atari keyboard though! Practice will make perfect.

This book is mainly a guide as to how to program your Atari microcomputer—how to tell it to do (roughly) what you want. As you go through the pages, you'll find out more and more about that. Meanwhile, if you are impatient, look through the *Owner's Guide* and try a few things. Or, if you are lucky enough to have a cartridge, pop it in the slot above the keyboard (not to be used for fingers!) with the label facing you, press the RESET key top right, and relax with someone else's program.

2 Starring Atari

I finished off the last chapter with some instructions for you to type at your Atari, to give you some feeling of superiority. And I'll start this chapter the same way—with a little weedy program that will practise your typing, and, just a bit maybe, impress any remaining on-lookers. 'Starring Atari' is in fact truly a 'program', the first in the book, you'll feel pleased about that!

A program is an ordered set of instructions the computer can follow. The instructions of 'Starring Atari' are written in a 'programming language' called BASIC. At this stage you need to know only one thing about that—if you make mistakes, the computer may not understand and will respond in a rather unfriendly way. No, I don't mean to put you off—no mistake you can make can do any harm at all to the machine. It's just that if you want the micro to follow your instructions, you have to phrase them just so. The notes that follow the 'listing' of the program give some important points. I think you had best read through these before you start typing.

Program 1: Starring Atari

```
10 GRAPHICS 0
20 PRINT "*****"
30 PRINT
40 PRINT
50 PRINT " * * *** * ** ** * * ***"
60 PRINT " * * * * * * * *** **"
70 PRINT " * * * * * * * * * * ***"
80 PRINT " * * * * * * * * * * ***"
90 PRINT " * * *** *** ** ** * * ***"
100 PRINT
110 PRINT
120 PRINT "          *** *"
130 PRINT "          * * *"
140 PRINT "          * * *"
150 PRINT "          * * *"
160 PRINT "          * * *"
170 PRINT
180 PRINT
190 GRAPHICS 18
200 PRINT #6;" A T A R I "
210 PRINT #6;"   A T A R I "
220 PRINT #6;"       A T A R I "
230 PRINT #6;"           A T A R I "
240 PRINT #6;"               A T A R I "
250 PRINT #6;" A T A R I "
260 PRINT #6;" a t a r i "
270 PRINT #6;"   a t a r i "
280 PRINT #6;"     a t a r i "
290 PRINT #6;"       a t a r i "
300 SOUND 0,126,10,10
310 SOUND 1,91,10,10
```

```
320 SOUND 2,72,10,10
330 SOUND 3,63,10,10
340 GOTO 340
```

Notes

Before you start entering this program, clear the micro's memory of any instructions no longer needed, by typing NEW (R). Then increase the contrast on screen, to make life a little easier for you, with SETCOLOR 1, 0, 14 (R). The italic text in the program printouts denotes the use of inverse, got with \blacksquare or \blacklozenge , and \square means space.

Enter each line exactly as given. When you have got it right by judicious use of the keyboard, and of the DELETE key to remove horrid slips, press RETURN. The cursor—the white blob, remember—should jump to the start of the next line. If it doesn't, you have made some slip that causes the micro to be rather unfriendly. After all, I warned you about that! The display will tell you off with an ERROR message. Ignore the details of any message like this you get, and just retype the offending line again, with a bit more care perhaps.

If the screen is getting too messy for you at any time press SHIFT and < to empty it, and type LIST (R) to get a cleaned-up copy of the lines entered so far. When you have finished, try the ultimate test—tell the computer to carry out the program instructions. You do this by typing RUN (R). Lo and behold, the program runs and you get screen and speaker output to tell you so. In fact, this program runs without end (that's the effect of the last instruction, line 340, if you want to know). You can stop it with the RESET key over on the right hand side of the board. LIST and RUN to your heart's content.

I only put the program in so that you could actually see one, and, if brave enough, try typing one in. I don't really believe in throwing folk into deep ends like this—in this chapter I really want to give you some background theory. So leave the program now, and the details of how it works—and let's do a bit of digging.

WHAT IS A COMPUTER?

Okay, I can hear you muttering the question—"what's the point of background theory"? After all, if you get a book on stamp-collecting, the author doesn't try to waste your time with lots of details of printing processes through the ages. Please don't reject the material in the rest of this chapter, however. For a start, I put a lot of work into it, and then—I think it's important. Computing is not like stamp-collecting: here you are trying to control a complex electronic machine and it helps to know a little bit about it. So—what *is* a computer?

The first thing you need to know is that there are lots of kinds of computers. I wouldn't be at all surprised if you have some around your house already—inside things like digital watches, electronic calculators, video games (perhaps even an Atari one!), and sophisticated features in a modern cooker, sewing machine, or video tape-recorder. All these contain what folk call special-purpose computers, complex electronic chips dedicated to a special task in each case. Your Atari computer, on the other hand, is general-purpose—given suitable instructions and attachments, it can do the work of any dedicated machine. If you've been able to use your micro with a few cartridges, you may have found it used for timing, calculating, gaming and so on—suitable instructions within the program of a general-purpose computer can handle all kinds of things.

In summary, then, a general-purpose computer such as the Atari can do many different things when properly instructed. A special-purpose computer is dedicated to a small range of functions and cannot (easily) have its instructions changed. When I am cornered, here's what I tend to use as a full definition of a general-purpose computer like ours:

A modern computer is a general-purpose, high-speed, digital electronic, stored program data processor.

I've said what 'general-purpose' means, so what about all those other posh words?

High-speed Allied with its accuracy, the high speed of operation of a computer is a crucial feature. Indeed, people have been looking for centuries for machines to offer such facilities to help them in their work. Computers can in fact do things much faster than people. They don't get tired, go on strike, have off-days, either; well, not usually. I'll come back to this high-speed thing again in due course—but you ought to know that modern micros can carry out thousands, or even millions, of actions each second. That means that they can do very complex operations very quickly and also keep up with fast-changing situations.

Stored program As I've already said, a program is an ordered set of instructions for a computer to carry out. (Note that the spelling is not 'programme', by the way—the American spelling is used here to distinguish the computing word from the broadcast one.) Now, just think what it would be like to give the machine each instruction in turn, when it is able to carry out millions in a second. Yes, that's right, the poor micro would be sitting around for almost 100% of the time waiting for the human operator to tell it the next thing to do. That would be a great waste of its potential, and, let's be honest, not many people would feel like using computers if that had to be done.

So the computer must be able to store the program instructions somehow, so that it can carry them out in a flash on demand. It needs some place to hold them, and that place, strangely enough, we call the *store*. The fact that computers can remember instructions for use on demand is what makes them so important.

Digital electronic I'll agree that that sounds a bit technical. But all it means is that this is an electronic system like your TV set or bedside clock/radio, working on tiny electric currents. (Actually those currents are *really* tiny, and really this is a *micro*electronic device.) The 'digital' bit refers to the fact that these currents run around in pulses rather than in a continuous stream. Pulses can be counted, which means they can act like numbers. The word 'digits' means 'numbers', and the word 'computer' means something for calculating.

Data processor And that's what computing's all about. Any computer processes data (which represents information that people are interested in) to produce more data (more interesting information). Stop and think for a moment how computers are used in the wide world—think how we have already used the Atari in this book—and then you can see why some people call computing 'data processing'. Really there is not much difference between a data processor and a food processor!

THE PARTS OF A COMPUTER

Don't worry—I'm not going to get any more technical. But I do think that it would help you to understand that *all* digital computers have the same basic structure as I shall now describe. It is true of your Atari, it is true of the massive multi-million pound machines used by the Pentagon and the Inland Revenue for their seedy purposes, and it is true of the massive machines of the 1940's. (I show one in Figure 2.1—it was a tremendous step forward, but as I'm sure you can guess it was far more difficult and far less fun to use than the Atari.)

The section of any computer that does the actual hard work is called the processor. Its job is to process electronic data, as you know. I said above that the data is in the form of 'electrical numbers', and the part of the processor that deals with this is called the *arithmetic and logic unit* (ALU to its friends); next to it in the central processor is the *control unit* that conducts the whole operation.

What does the central processor process?—data. How does it know what to do?—through programs. Data and programs must be stored, as we've discussed. So close to the processor is the main *store*, or memory.

If you tried typing in the program at the start of this chapter, you'll have realised that keyboard entry is not the best way to give computers instructions. Bear in mind, after all, that there are millions of different tasks that a general-purpose computer could be expected to do, so that there are millions of possible programs. All but the very simplest

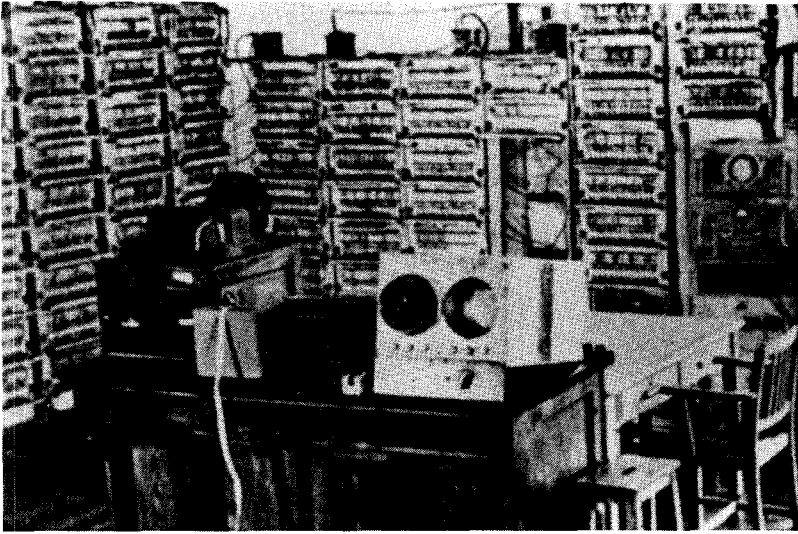


Figure 2.1 A 1940's computer

computers therefore have an extra memory, the so called *back-up store*. In the case of the Atari this can be a cassette recorder and cassettes (see Chapter 5), or what's called a disc system, or those costly cartridges.

All computers must also be in communication with their users. Each user wants to be able to feed in commands, programs, and data. The *input unit* deals with that need—in the case of the Atari that is, of course, the keyboard.

Also the computer has to be able to give the user messages and data—it needs an *output unit*. The TV screen that we have been using is a standard output unit for microcomputers. (A printer is another one.) In Figure 2.2 I show the way that all these bits fit together. It's worth noting that people often call the input/output units and back-up store the computer's *peripherals* (outside parts). There, that wasn't so hard was it?

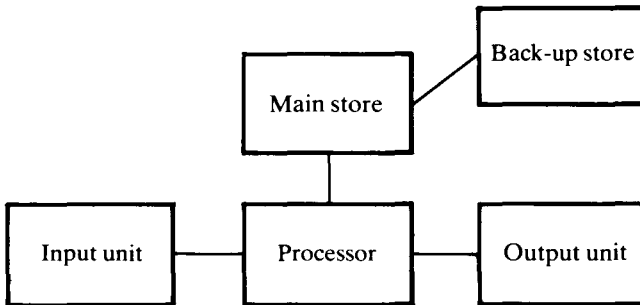


Figure 2.2

EFFICIENT CHIPS.

As I have emphasized, Figure 2.2 applies to all computers in essence, whether large or small. Figure 2.3 shows a typical modern 'main frame' (equals hefty) computer installation. All the boxes you can see there can be classed into one or other of those shown in Figure 2.2. Who knows, by the time you've worked through this book, you may be able to get a job running an installation like that. . . .



Figure 2.3 Modern mainframe computer.

Figure 2.4 shows a typical Atari set-up. It's not quite on the same scale but perhaps it is a little bit more friendly and easy to understand. The keyboard itself is of course the input unit, the box it sits on contains the processor and main store. You should have no problem in recognizing the output unit, nor the back-up stores used.

It's no particular task to undo the screws at the bottom of your Atari keyboard and open the box. However, if you do that, you will lose the power of your Guarantee, and anyway you may lose the screws and not be able to put the thing together again. So here's a brief description. Inside the keyboard case is a 'printed circuit board' layout, protected by a sheet of metal (to reduce any possibility of interference on your family's broadcast TV set): so really you couldn't see much even if you did use your screwdriver.

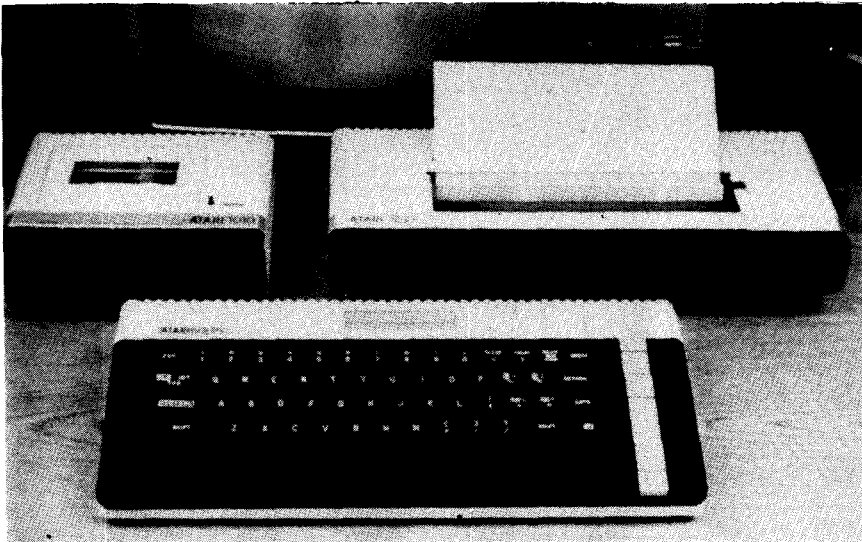


Figure 2.4 Atari with peripherals.

The layout contains several 'chips'—amazingly complex microelectronic circuits packed into a tiny slice of a substance called silicon. There aren't many chips, because in modern computers like your Atari those circuits are truly complex. Amongst them are processor (ALU and control) chips and memory devices. There too are the famous chipper trio charmingly called ANTIC, POKEY and GTIA. These have, respectively, tasks associated with graphics/screen, sound production and display.

That's enough, I guess. Let's put the metaphorical lid back on, and get down to some real program development.

3 In and Out

I threw you in the deep end in the last two chapters by suggesting BASIC instructions for you to try on your Atari. It *was* the deep end, but I know jolly well that if I didn't do anything about instructing a micro till Chapter 3, you'd think this a rotten programming guide. Maybe you still do—well, it gets better.

The first two chapters provided necessary setting-up practice and theory. Now we can really get on to telling the idling beast what to do, and the rest of the book will be dealing with that. By now you should have some practice in setting the thing up ready to work, and some knowledge of using the keyboard. So—let's check.

TAKING COMMAND

Set up your Atari (with the BASIC cartridge in, if it's a 400 or 600 model) and switch on. Here's the set of instructions we looked at in Chapter 1. Enter each in turn. Can you do so without ERROR shouted visually at you?

```
SETCOLOR 1, 0, 0
```

```
SETCOLOR 2, 2, 10
```

```
SETCOLOR 4, 3, 0
```

DON'T FORGET:

- a It's SETCOLOR not SETCOLOUR: this is a US-designed machine, so needs (for the nonce) US spelling—it won't understand if you make even a weeny spelling error.
- b The stuff after the SETCOLOR consists of numerals and commas. Again, if you don't get them dead right, the computer will cut you right dead. *Note*, however, that here (though not in all instructions), the Atari doesn't mind if you put in extra spaces. I've put one in after the 'keyword' SETCOLOR, but this is to aid readability for you, not for the machine.
- c If you don't press RETURN at the end of the instruction, the machine won't know you have finished—so it'll politely wait (for ever if need be). RETURN (the key at the right of the main block) tells the micro that the current input is at an end. In fact, it truly means 'carriage return'—it signals the end of the line of typing on a pre-micro keyboard, as on a typewriter. The RETURN key:
 - 1. marks the end of the current input;
 - 2. tells the 'print-head', in other words the cursor, to return to the start of the line;
 - 3. provides a 'line-feed', a shift down one line ready for the next.

All this harks from the days when computer users faced a roll of paper in a printer. Things are different now, but the carriage return/line feed bit is fairly convenient even on a screen.

That's a lot of notes. Sorry. What does the instruction SETCOLOR 1, 0, 0 (R) do? It changes the colour of the 'printing' (there I go again) on the screen from light blue to very dark. We'll study SETCOLOR fully later—see Chapters 4 and 8—but meanwhile, in

SETCOLOR a, b, c

SETCOLOR is the Atari BASIC keyword for (surprise, surprise) setting a color, I mean colour.

The *parameter a* decides what the colour is of:

- 1 Foreground ("ink" or text)
- 2 Background ("paper")
- 4 Border

The *parameter b* decides what the colour is:

0 grey	8 pale blue
1 pale orange	9 cyan (very pale blue)
2 orange	10 turquoise
3 red	11 pale green
4 pink	12 green
5 magenta	13 yellow-green
6 purple	14 orange-green
7 blue	15 gold

The *parameter c* decides the brightness of the colour, ranging from 0 (darkest) to 15 (brightest).

Play with SETCOLOR a bit, though, as I say, we'll be going much deeper into the subject later. Some notes are of value here. Firstly, there are restrictions in theory:

1. Ink colour has to be the same as paper colour, and the latter wins any conflict. So in SETCOLOR 1, b, c, b can take any value you like as the micro in effect sets it to be the same number as in SETCOLOR 2, b, c.
2. The brightness has in fact only eight possible settings, so really there is no point in using values of the parameter c other than 0, 2, 4, 6, 8, 10, 12, and 14.

Secondly, there are restrictions in practice. You may not agree with the colour names in the table under parameter b above (especially if you are using a black-and-white telly!). For a start different colour TV receivers and monitors react differently to colour signals. You'll know this if you've ever been faced with a bank of sets in a TV store. And folk's eyes would also cause them not to agree which a given colour is. Me, I'm one of the many with poor colour vision—it was a nightmare setting up the above table of colour values. I have problems with traffic lights too. . . .

That was more of a digression than I meant. Let's get back, with fewer notes, to the series of instructions I am plagiarizing from Chapter 1. Carry on entering (and RETURNing):

PRINT CHR\$(125)

Don't forget: \$ is SHIFT and 4 and the brackets come with SHIFT and the 9 and 0 keys. This strange instruction (which we'll explore further, in Chapters 4 and 13) is the Atari's tortuous way of getting the screen clear. You could also use SHIFT and < or CONTROL and < for a similar effect. Most modern micros have the simpler instructions CLS (meaning CLeaR the Screen!).

POSITION 10, 10

This shifts the cursor down 10 lines and across 10 places. Well, it would do if things were better organized. We'll organize them better in a moment.

```
PRINT "Testing - 1, 2, 3!"
```

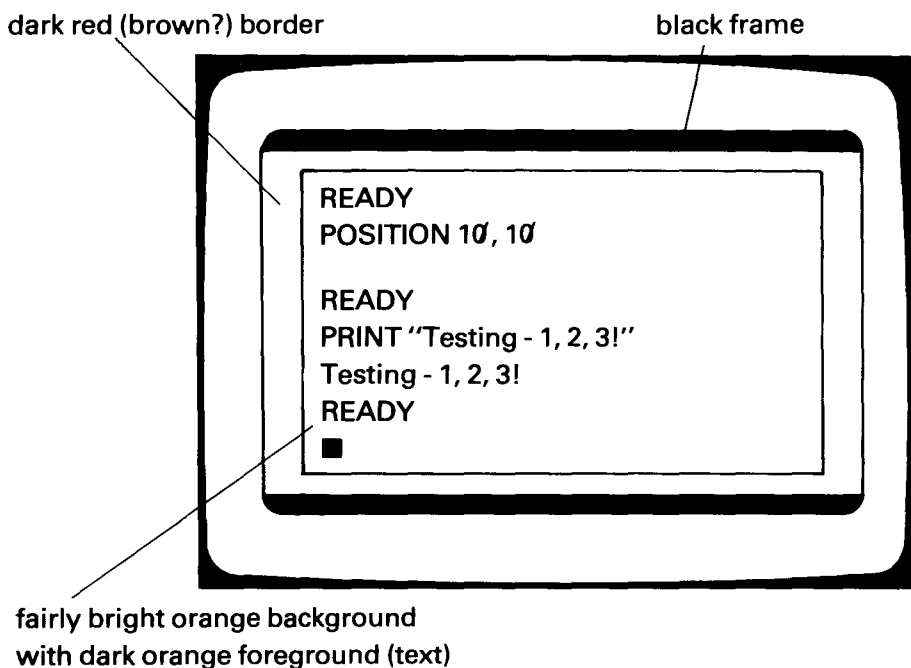
(Don't forget—press the CAPS key to go into/come out of lower case—small letters—mode. The system with the Atari computers 400 and 800 is slightly different. And of course, “ and ” are got with SHIFT and 2. Computers don't distinguish between “ and ” by the way, but most computer book printers, including Shiva's (hint, hint) have to use 66, 99 rather than 11!)

What this instruction does is to print out the message inside the speech marks (without showing the speech marks) on the next line (where the cursor would have gone if you'd just pressed RETURN).

Here's that list of instructions again, with no comments at all. (I'm trying to be strict with myself.)

```
SETCOLOR 1, 0, 0
SETCOLOR 2, 2, 10
SETCOLOR 4, 3, 0
PRINT CHR$(125)
POSITION 10, 10
PRINT "Testing - 1, 2, 3!"
```

(Figure 3.1 shows what you should have got if *you* hadn't digressed at all, *and* you'd managed to keep the dreaded ERROR at bay.)



(only crudely to scale)

Figure 3.1

The above six instructions are truly called *commands*—the micro carries each one out as soon as you press (R). Folk sometimes say that the machine is here working in direct, or immediate, mode.

Using commands with a micro is often useful—but really it treats the machine as no more than a very posh calculator. In fact, that's the main use of working in direct mode—calculating. (We'll meet other uses later.) Like this:

```
PRINT 36 + 65
```

Pressing (R) gives the answer at once—101.

You could (I hope) have worked that out in your head, so let's try something harder:

```
PRINT 36.75 * 0.321    (* is the computer's times sign, key next to CAPS)
```

Answer, in a flash: 11.79675. My calculator agreed and I shan't check on my fingers. (That last sentence isn't really a joke: strange as it may seem, micros are not as good as pocket calculators at doing some sums. They are not bad, but they can be less accurate. That applies particularly to the older Atari models.)

```
PRINT 36.75 / Ø.321    (/ means divide: key over the right hand end of the space bar)
```

```
PRINT 36.75 ^ Ø.321    (^ means raise to the power if you know about such things; key it with SHIFT and * and note that this time the machine hesitates a fraction of a second before revealing the answer.)
```

Practise with PRINT (some sum) if you like, as well as with PRINT "(some message)". But I'll get on with my main theme, which is that of commands.

There are two reasons why commands are not always the best way to instruct a computer. Firstly the machine doesn't remember them, so if you want to repeat one you must type it in again. (There's a way round that in some cases; we'll meet it in Chapter 9.) Secondly there is the reason I mentioned in Chapter 2—typing in a series of commands to get to some end point is not an efficient use of a computer as it can work so much faster than you can.

We can get over both problems in one fell swoop by putting the instructions in a *program*. Previously I defined a program as an ordered set of instructions the computer can carry out. Now let's recall the point about storage made in Chapter 2—a program is a *stored*, ordered, set of instructions that the computer can carry out. We met a program at the start of the last chapter, but I am not going to repeat that with millions of notes. Let's just turn our latest set of commands into a simple program.

LIGHT PROGRAM

The obvious difference (at least I hope it's obvious) between a set of direct commands as on Page 9 and the program listed at the start of Chapter 2 is that each instruction line starts with a number. By a brilliant piece of verbal footwork, the early users of the BASIC programming language came up with the name 'line numbers' for these. So let's just add numbers in front of each instruction in our set. No problem. Type this in with me:

Program 2: A testing program

```
10 SETCOLOR 1,0,0
20 SETCOLOR 2,2,10
30 SETCOLOR 4,3,0
40 PRINT CHR$(125)
50 POSITION 10,10
60 PRINT "Testing - 1,2,3!"
```


Notes

Before you start entering this program, clear the micro's memory of any instructions no longer needed by typing NEW. Then, increase the contrast on the screen with SET-COLOR 1, 0, 14. Don't forget the DELETE key if, as just may happen, you note an error before pressing RETURN. If you realize a RETURNed line is wrong, re-type it. If you get a line number wrong, type the number alone and then press RETURN. Don't worry about spacing.

Use SHIFT and < to clear screen if it gets too messy, and type LIST whenever you want a nice copy (with the 'right' numbers of spaces).

NEW and LIST are, of course, commands. Yes, you can use commands while programming. The most important one is this—RUN. It tells the machine to carry out the stored instructions in order.

Try RUN then, when you've done the entering. Hope it works. Hope you get what I got—Figure 3.2:

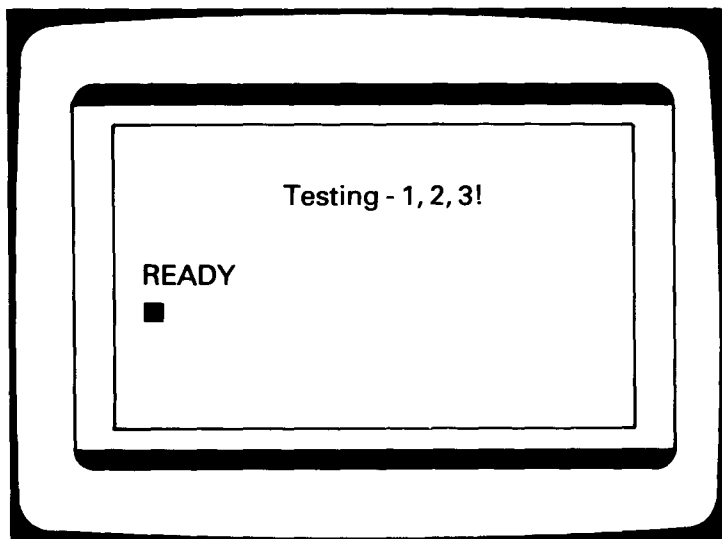


Figure 3.2

Now *that*, unlike Program 1, is a program I want you to understand. Only when you understand it, ought you to go through to the next round. . . . So I'll give *more* notes, some old and some new.

Instructions are set of characters that tell a micro to do something. They must follow the rules laid down.

Commands are instructions entered in direct mode, in other words without line numbers. The machine carries each command out immediately (after (R)), but does not store it in memory.

Statements are instructions entered in indirect mode, in other words with line numbers, thus forming part of a stored program. The computer puts each statement in its memory and does not carry it out until it gets told to RUN. RUN tells the machine to carry out the instructions ('execute' the program statements) in number order.

Line numbers are BASIC's symbol for program statements. A line number must be a whole number, from 0 to 32767 inclusive. We step line numbers up in jumps of ten mostly, so that later we can add other lines in between if we need to.

NEW: to clear the memory of program statements no longer needed.
LIST: to display on screen the lines of a program in number order.

Keywords are the opening words of instructions. You must enter them in capitals (upper case). As well as NEW, LIST, and RUN, we've met:

SETCOLOR: to define the colour and brightness of foreground, background, and border.

POSITION: to move the cursor to a given site on screen.

PRINT: to cause material such as messages to appear on screen; sometimes to control output, as in `PRINT CHR$(125)`—to clear the screen.

All keywords can be used in commands or statements, but there may not always be much point (LIST and NEW within programs have few benefits, for instance!).

MIND YOUR LANGUAGE

I think it's time for another theoretical break. Well, really I think it's time for some more theory, but if I call it a break it sounds more exciting. This break concerns what are called programming languages. I guess you know by now that you give your Atari micro instructions in the BASIC language. Truly BASIC isn't a language—it's a programming system used to communicate instructions to computers designed to understand it.

In fact no computer actually understands BASIC! Recall what I said in Chapter 2 about digital electronics? All a computer's circuits can work with are those tiny pulses of electric current. To keep things simple, digital computers nowadays can work with only two levels of current, on or off. If there is a pulse of current, the computer in effect counts it as the number 1; if there is no current, the number is called 0.

So computers can act on instructions only if they appear as a string of current pulses—a string of 0s and 1s. Here's an instruction written in that form:

0010110010100111

In the good old days of computing, the programmer had to enter all instructions like that. Figure 3.3 shows what he or she had to do before pressing the ‘RETURN’ button. Get all the switches in the row on the computer’s ‘front panel’ in the right positions.

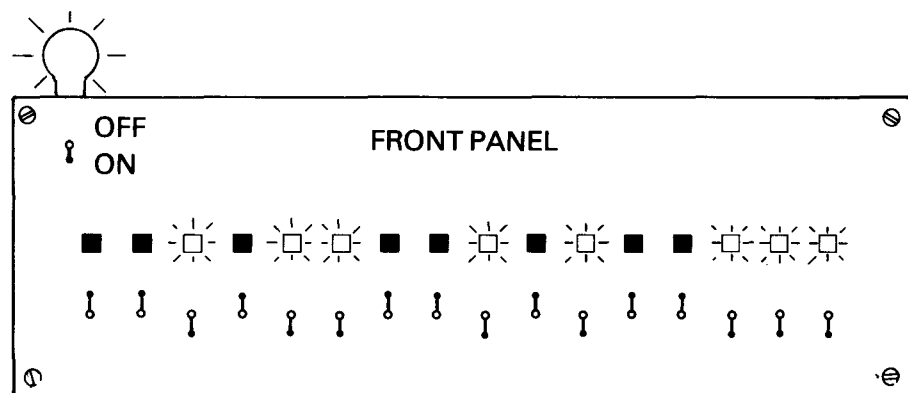


Figure 3.3

All instructions (direct commands or those in programs) had to be entered in this way. Each one had to be worked out as a set of 0s and 1s, checked a million times, entered, and

checked a second million times. Just think how easy it would have been to make mistakes and how long getting even a simple program into store would have taken!

Programming like that is (was) incredibly tough on humans, though the computers of the time (both of them) could 'understand' the instructions and carry them out without problem. We say that programs with instructions like that are written in *machine code*—a coded form that the machine could follow.

But computers are supposed to make life easy for humans! So along came the idea of programming in so-called higher-level code systems. The systems folk nowadays call machine code and *assembly language* are like this. They are easier for humans to work with—easier to write, check, enter, and test—but of course the computer can't follow. What the computer needs inside it now is a program—written in true machine code—to translate the higher-level instructions into its baby 0s and 1s. Tough on the machine, but tending to make life easier for the programmer.

Even more recently, a couple of decades ago, systems that were even simpler (for humans) came along. Baptised high-level languages, these were much more like plain English (yes, English-speakers have a real advantage in this field!)—and even harder on the machines.

Now there are hundreds of high-level program languages. You may have heard of COBOL (common in commerce), and ForTran (super in science), and even ADA (ADvantageous in Armaments), but there are lots and lots more. The most popular of all is BASIC designed (in 1964) as an all-round system that's easy to start work in. It is easy—after all, you'll be proficient in it after only a couple of dozen hours with this book.

I said that BASIC was the most popular high-level language. By that I mean that it's the one the most programmers use. The reason for that is not just its ease for beginners, but its almost universal use with microcomputers. Nearly all microcomputers have BASIC as their standard high-level system.

With that doubtful honour comes a big, big problem—each new micro is likely to have a different version of BASIC from its predecessors. Indeed micro marketing is often at least partly based on new BASIC features—colour, sound, graphics (ability to draw lines and shapes) and so on. Atari BASIC is not like any other (as far as I know), for instance, but if you can follow this version, you'll have little problem understanding programs

```
10 INK 0
20 PAPER 3
30 BORDER 6
40 CLS
60 PRINT AT 10,8;"Testing - 1,
  2,3!"
```

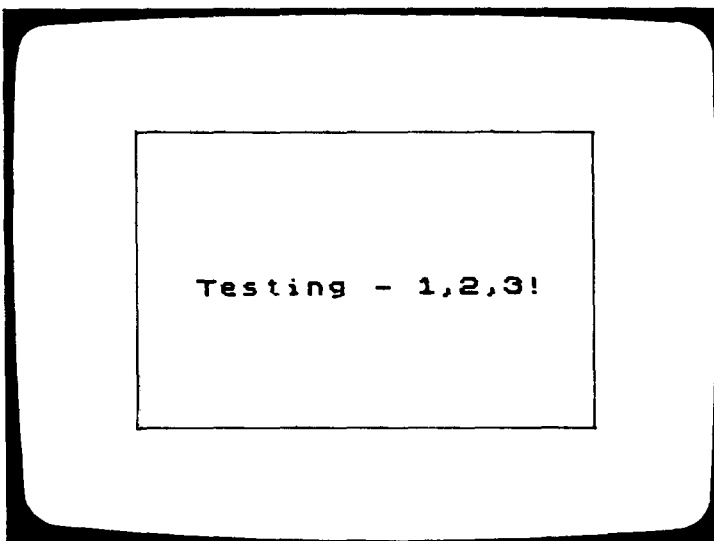


Figure 3.4a Program 2 in Sinclair BASIC, and its result.

written in BASIC for other machines (or, indeed, programs written in closely-related languages like ForTran and COMAL).

Folk call programming systems like these 'languages' because they are used for communication, and because it's convenient to think of their having a vocabulary (the keywords) and a grammar (the rules for using the keywords). As a result, the various versions of BASIC are 'dialects'. The dialects may differ from each other, but they have many things in common—line numbers and the main keywords in particular. Figure 3.4 shows how Program 2 would appear in two other major BASIC dialects.

```
5 MODE 5
10 COLOUR 3
20 COLOUR 129
40 CLS
60 PRINT TAB(2,13) "Testing - 1,2,3!"
```

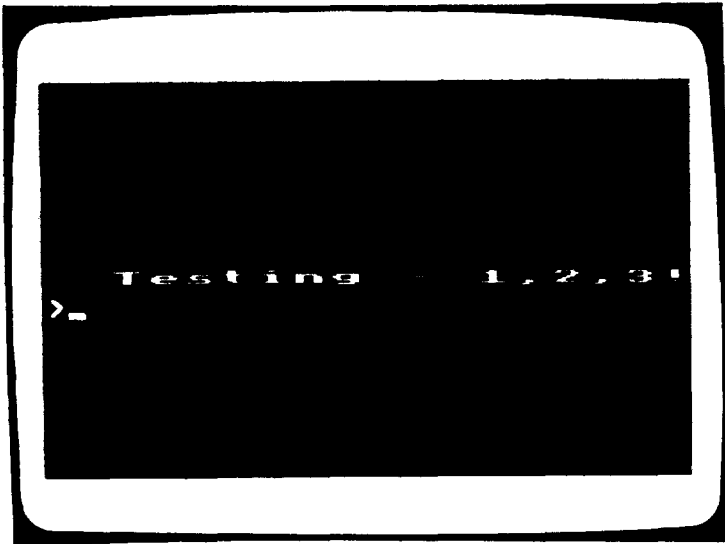


Figure 3.4b Program 2 in BBC BASIC, and its result.

The main reason you need to know about language levels and dialects is so that you realize that you simply cannot copy programs printed in books and magazines *unless* they are stated to apply to your machine. Perhaps you've discovered already that bookshops are crammed to the eyeballs with volumes carrying such titles as *Learn BASIC in 10 minutes* (by Ivor Raquett, Rip-Off Press 1972, £19.99) and *500 programs for your micro* (by O P Timest, Con Publications 50p). The periodical shelves overflow too—loads of glossy weekly, monthly and quarterly magazines with the word 'Computing' in their titles. These give news, reviews, and program listings for many common home machines. AVOID all these unless they contain material you really want that is stated to run on Atari computers.

By the time you have finished this book, you'll know pretty well how to get different effects from your micro's BASIC. You may then be able to transfer programs written for other machines as long as you know the effect those programs and their instructions are aiming for.

In particular note the great variation between dialects in dealing with colour, sound, graphics (perhaps using keywords like MOVE, PLOT, DRAW, and CIRCLE) and machine code (with *, [, ?, PEEK, POKE, CALL, and USR). Programs with such features will cause you great trouble in transfer. Figure 3.5 summarizes this section.

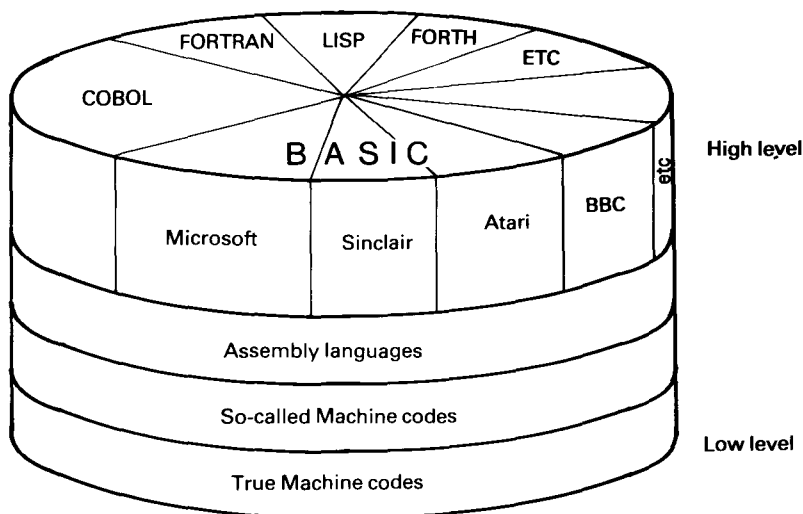


Figure 3.5 Language levels.

INPUT-PROCESS-OUTPUT

As I guess you've gathered by now, the work of any computer—large or small—divides in essence into three types: Data input, data processing, and data output. Some examples follow.

Surely the most obvious use of a computer involving input/process/output is the one inside the simple electronic calculator. Data (numbers and operators) are input and the results of processing them are output on the one-line display. As I've pointed out already, such calculators are dedicated computers—dedicated to the single task of calculating. Our concern in this book is of course general purpose computers like the Ataris, as we've seen, these can act as calculators (with, for instance, `PRINT 14/7(R)`), and do all sorts of other things too. Here are three brief case studies involving general purpose computers programmed to carry out the task in question.

The cash dispenser



Figure 3.6 A Midland Bank AutoBank machine

One common form of automatic cash dispenser in banking works like this.

Input this is of (a) a secret code stored in some magnetic form on the user's card; (b) a second secret code that the user must type in; (c) the details of the user's requirements, also typed in at the time. These inputs are guided by messages output by the computer.

Processing This involves (a) relating the two input codes; (b) checking the user's balance in the central computer record; (c) calculating how to dispense the right amount; (d) calculating the new balance.

Output This involves (a) messages on the screen of the unit; (b) the cash itself; (c) a printed slip giving the details; (d) a record in the central computer of what's happened for the user's statement.

Systems like this are being developed to computerize all kinds of essential transactions, such as in shopping, buying petrol, and getting goods on mail order with Prestel.

Holiday bookings



Figure 3.7 Holiday booking system (Courtesy of Horizon Holidays).

Holiday bookings were one of the first large-scale uses of computers that the general public could see to be of direct value.

Input Here the user gives information in response to the necessary questions, again guided by messages on screen.

Processing This allows the questions to be answered, financial details to be worked out, and hotel and flight records to be updated.

Output This provides (a) on screen, various messages and the answers to questions; (b) the completed tickets from a printer; (c) summaries of the necessary details at company headquarters, the agent's office, and perhaps the credit card company office as well.

It is very likely that voice input and output will soon replace some parts of systems like this. Indeed in certain places you already can phone a computer and hold a kind of conversation about forthcoming airline flights.

Input This consists of pencil marks on a specially printed sheet (see Figure 3.8); a *mark sensor* detects the variation in the brightness of reflected light as shown in Figure 3.9.

Output Output is of printed scores lists; the result of statistical analysis (perhaps in graphical form); and candidates' certificates.

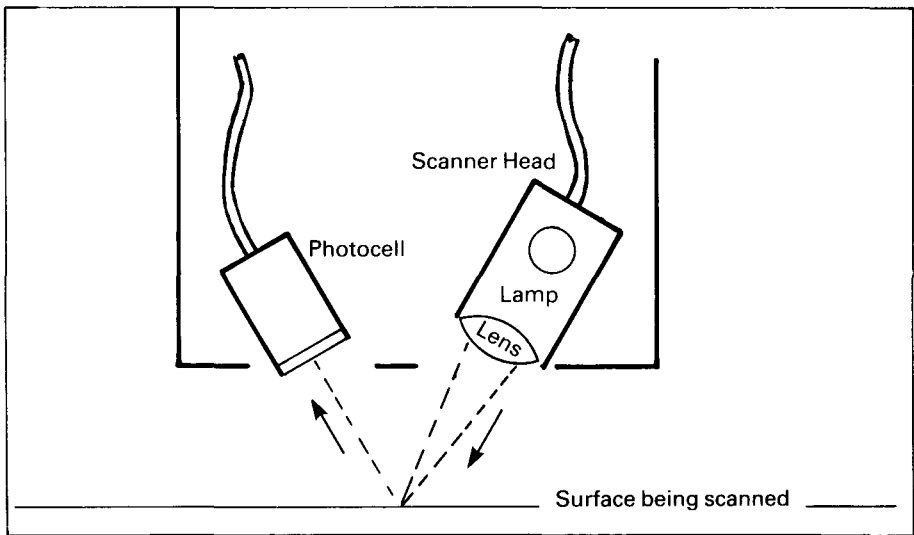


Figure 3.9 A mark sensor reading the answer sheet.

ATARI INPUT-PROCESS-OUTPUT

In each of the cases I described above, data reaches the computer through some kind of *input device*; this may well be a long way from the processor. How the computer then processes the data depends on the program that is stored in its memory. That program will also guide the user to give the input data in the right form by putting suitable messages on the screen, and produces suitable output. In just the same way your Atari must be programmed to do each task. You must tell it to:

- (a) guide the user to give acceptable inputs;
- (b) do the necessary processing;
- (c) *format* (lay out) the output so that it makes sense and is easy to read.

We'll use these keywords in this program:

<i>Input</i>	INPUT	(easy eh?)
	DIM	(which we need to go with INPUT in some cases)
<i>Process</i>	LET	
<i>Output</i>	PRINT	(puts material on screen)
	POSITION	(sets site for printing)
	SETCOLOR	(sets colour)
	PRINT CHR\$(125)	(clears screen)

As you know the last four, I'll introduce two others which don't fall into the above three classes:

<i>Control</i>	GO TO	(which alters the normal order of flow through the program statements—not for frequent use)
<i>Remark</i>	REM	(which leaves a message in the listing to help readers)

Notes appear after the listing—so, off you go, and type it in. This is a long program, by the way—a couple of screens full. Note, therefore, that you can stop the scrolling (upward movement) of the screen listing at any time by pressing CONTROL and 1. You don't need to type in REMs if you want to save time—they are there for information; similarly you can cut all spaces that aren't inside speech marks.

Program 3: Inprout

```

10 SETCOLOR 1,0,14
20 SETCOLOR 2,5,0
30 SETCOLOR 4,0,0
40 PRINT CHR$(125)
41 REM ** Above set up display
50 POSITION 4,4
60 PRINT "Hi! What is your name?"
70 DIM NAME$(12)
80 INPUT NAME$
81 REM ** Input routine
90 POSITION 4,8
100 PRINT "Thank-you, ";NAME$;","
110 POSITION 4,12
120 PRINT "In which year were you born";
130 INPUT YEAR
131 REM ** Number input is simpler
140 LET AGE=1984-YEAR
141 REM ** Processing at last!
150 POSITION 4,16
160 PRINT "So you are about ";AGE;" years old."
170 PRINT "That's about ";AGE-1;" years older than me."
171 REM ** Bit more processing Change data if need be
180 POSITION 4,22
190 PRINT "Please press RETURN to go on...."
200 DIM CONT$(1)
210 INPUT CONT$
220 PRINT CHR$(125)
230 PRINT NAME$;"'s about ";AGE;" years old."
240 GO TO 230
241 REM ** See the control bit?

```

The whole of the next chapter is on the subject of output statements in Atari BASIC so in these notes I'll not say much about them. Rather I'll concentrate on the other new keywords and their use.

Using INPUT

Program 3 uses only one input instruction—it is, surprise, surprise, INPUT. See it in lines 80, 130, and 210. There are others that we shall meet later. When the processor meets INPUT, it carries out the following tasks:

- puts a 'prompt', a ?, at the next print position to tell the user to type something;
- awaits a keyboard entry ending with (R);
- assigns the input data to the named variable.

Data items

As you know, a computer is a data processor. We need to now the vocabulary associated with the various types of data the Atari can process.

Atari BASIC has two types of data—numeric and string. *Numeric data* consists of pure numbers alone; examples are 0, 42, -7, 2.34, and 2E4 (meaning 2×10^4). A *string* can be any combination at all of keyboard characters, for instance, "Shiva", "R2D2", "PI = 3.142", "1984". Yes, strings can contain purely numeric data but if we show them in speech marks ("...") they are strings and *not* numbers.

Both types of data, numeric and string, come in various forms. I show these in the table:

Table 3.1

Form	Numeric	String
Constant	9.87	"The answer is "
Variable	The user's age in days	The user's name
Variable name	YEAR	NAME\$
Expression	AGE-1	(see later)

A *constant* is a value which is set and not expected to change for a time at least. Examples are:

```
LET FACTOR = 2 * 3.142
LET MESSAGE$ = "Press RETURN."
```

A *variable* can be expected to take any value, as in:

```
INPUT RADIUS
INPUT NAME$
```

Here *RADIUS* is a numeric *variable name* and *NAME\$* is a string variable name.
An expression is a combination of constants and/or variables that the computer needs to work out:

```
LET OUTSIDE = FACTOR * RADIUS
```

Numeric expressions can be very complex, but string expressions can use only addition (and that with difficulty). I'll come back to this again later, but note once more the four main arithmetic operators and their BASIC symbols: add (+), subtract (-), times (*), divide (/).

As I say, Program 3 uses three INPUT statements; in truth only two of them involve real assignment.

Numeric input INPUT followed by a numeric *variable*, as in INPUT YEAR (line 130 in the program), tells the processor to display the prompt (the ?, you recall); to wait for a numeric (number) input followed by (R); and to assign the value that was input to the variable named, in this case to YEAR. The program can then process the input value, as in line 140.

Micros differ in this, but in the case of the Ataris, a numeric variable name can be any combination of capital letters and numbers as long as the first is a capital letter. The maximum length in practice is about 100 characters, but I trust you will never try to use names that long! If the first one is *not* a capital letter you get an ERROR report thrown at you.

Table 3.2 sets out what you can and cannot do as far as naming numeric variables is concerned.

Table 3.2

Allowed	Not allowed
A	a
FIVE	5
R2D2	2R2D
YEAROFBIRTH'	YEAR OF BIRTH

Of course it will help you and other people reading the program lines later to see long meaningful variable names like ANNUALOUTPUT and HISCORE. In particular you will find that this is important when you come back at some time in the future to improve a particular program. However, long names like this present obvious typing hazards to the programmer and take up a lot of computer memory. On the whole therefore I shall use very short variable names in the rest of this book. Indeed, to be honest, it is only when you get to writing very long programs that you find you need names of more than a single letter. After all you have 26 possible ones there.

String input INPUT followed by the name of a string variable (one ending with \$), acts just as before, except that *any* mixture of alphabetic, numeric, and other keyboard characters can be accepted by the computer before the RETURN when the line is carried out. (You have probably found out by now, and if you haven't you soon will, that if you enter a non-numeric value in response to a numeric INPUT statement the puter will complain with ERROR 8.)

Thus line 80 in Program 3 is INPUT NAME\$. When the computer reaches this part of the program, it displays the prompt, the ?. Then it waits until it gets a keyboard input followed by (R), and assigns that input value to the string variable NAME\$. The program may then use the value of the variable, whatever it may be—as in lines 100 and 230.

The name of a string variable can be set up to follow exactly the same rules as for a numeric variable, *except* that the final character must be a \$ (say it 'string').

As you have already learned, a string INPUT statement must be preceded (somewhere in the program) by a DIM statement. What DIM does is to tell the micro to reserve the stated number of spaces in memory for the string. Thus DIM NAME\$(12) reserves 12 spaces in memory for the string to be called NAME\$. If the string input turns out to have fewer characters than twelve in this case, the micro doesn't mind. On the other hand, if it has more, then the extra ones are chopped off and lost for ever. I guess that not many people would enter a string of more than twelve characters in response to the question "What is your name?", but if they do, their name is chopped off for life.

Wait Lines 190–210 in Program 3 form a little routine which you will find useful very often. What the routine does is simply to stop anything happening until the user is ready to go on. Then *any* input followed by (R), including (R) by itself, will let the program go on to clear the screen and deal with the final statements. This is of course really a special case of the use of string inputs—but this time the program will not use the assigned value.

In practice it is of course not possible to separate input and output functions. I am thinking in particular of the need for a clear message to go before each INPUT statement, so that the user has no doubt at all what he or she should enter. This is an aspect of what we call 'user-friendliness', the need for every program to be designed to make the user feel unthreatened and at ease. If you check back through Program 3, you will see what I mean—each INPUT follows a message telling the user what to enter. I think I should warn you now, user-friendliness is one of my (many) hobby-horses. I shall return again and again to this theme. . . .

Processing

At last we come to the central part of any computer operation. Look at line 140 of Program 3. We call this an *assignment* statement. Its structure is

LET (variable name) = (expression)

The variable can be numeric or string; the expression can be a constant, the value of another variable, or some combination of these as complex as you wish—first the micro evaluates it. Here are some examples of expressions; I put them in LET statements so that you can get used to this important keyword.

LET FACTOR = 12 (a constant)
LET YEARS = AGE (a variable)
LET MONTHS = YEARS * FACTOR (an expression)
LET OUTSIDE = 2 * 3.142 * RADIUS
LET VOLUME = 4 * 3.142 * RADIUS ^ 3/3 (^ means raise to the power)

By the time we need to use complex expressions like that last one, we need to know the order in which the micro is going to work out a given numeric expression. I'll come back to this in more detail later, but here is the *priority* table we need to know at the moment.

Table 3.3

Priority	Operation	Meaning
highest	(. . .)	brackets
	SQR, SIN, etc	functions
	^	raise to power
	*, /	times, divide
lowest	+, -	add, take away

If in doubt—put in brackets!

We have already met another important method of assignment—the INPUT statement, as in INPUT AGE, or INPUT NAME\$. (There is yet another—READ—that we shall meet soon enough.) The rules for assignment by LET are just the same as for assignment with INPUT. That applies to the naming of variables and to the use of DIM in the case of string assignments. In the latter case, for instance, we could do this:

DIM RESPONSE\$ (10)
LET RESPONSE\$ = "Processing"

One final point needs a mention here. It is that the keyword LET is in fact optional. So you could have SCORE = 15 or TITLE\$ = "Invaders". However, I personally prefer to use LET all the time, and shall do so throughout this book. The reason is that it makes the statement clearer, and, in any case, the keyword LET is essential in some dialects of BASIC.

A fairly common use of LET is to keep score—to count some repeated event during a program. We would first set the counting variable to the necessary initial value, often 0, using LET COUNT = 0, or LET SCORE = 0. Then when the need arises we increment (step up) the value of the variable by 1, with, for instance, LET COUNT = COUNT + 1. Then when required, a PRINT COUNT statement will let the current value appear on screen.

Program 4 shows this in action. Like Program 3 this is fairly trivial, its main function being to show the points I am making at the moment. Perhaps later you will feel that you wish to come back and improve it. . . .

Program 4: Out for the count

```
10 LET GO=0
20 LET N1=INT(RND(0)*10)
30 LET N2=INT(RND(0)*10)
40 DIM OK$(13)
50 LET OK$="That's right."
60 DIM NO$(14)
70 LET NO$="Sorry - wrong!"
80 DIM CONT$(1)
91 REM ** All above is setting up
90 PRINT CHR$(125)
100 POSITION 4,10
110 LET GO=GO+1
120 PRINT "Go number ";GO;"!"
130 POSITION 4,14
140 PRINT "What is ";N1;" times ";N2;
150 INPUT ANS
160 POSITION 4,10
170 IF ANS<>N1*N2 THEN PRINT NO$
180 IF ANS=N1*N2 THEN PRINT OK$
190 POSITION 4,22
200 PRINT "Please press RETURN to go on...."
210 INPUT CONT$
220 IF ANS<>N1*N2 THEN GO TO 90
221 REM ** Return if wrong
229 REM ** Go on if right
230 PRINT CHR$(125)
240 POSITION 4,10
250 PRINT "Well done - you got it in ";GO;"!"
260 POSITION 4,22
270 PRINT "Press RETURN for another one...."
280 INPUT CONT$
290 RUN
291 REM ** Have another go
```

This program introduces a number of new ideas, as well as giving you plenty of chance to work with LET. A major point is that it keeps on going, round and round, so that the only way you can escape from it is to use the BREAK key (at any stage). I'll go through the other points as they appear in the program.

Lines 20–30 What each of these lines does is to select at random a whole number between 0 and 9 inclusive. I can't expect you to understand exactly what's happening at this stage (though I trust you will soon enough)—but just remember the structure $\text{INT}(\text{RND}(0) * N)$ for getting the program to choose a random whole number between zero and one less than N.

Lines 40–70 This program does not really need any string assignment, for reasons that may become clear later, but I put these lines in to show you how it can be done. Remember always to 'dimension' any string you are going to need, to a size as large as it may ever be, before you assign it with LET or INPUT. The assignment does not need to be just before the string first appears, however. That's why I dimension CONT\$ in line 80, but don't use it until quite a lot later in the program. Note, by the way, that once a string is DIMed in a program, you can't reDIM it. That's a good reason for putting all DIMs at the head of the program.

Lines 140–150 I repeat here a trick used in Program 3—putting a semi-colon at the end of the message preceding an input statement forces the INPUT's ? to appear at the end of the message rather than on the next line. I'll deal fully with the semi-colon in the next chapter.

Lines 170–180 Here I use for the first time in this book the most important structure IF...THEN.... I think you'll have no trouble in understanding what's happening here, as soon as I tell you that the symbol < > means 'is not equal to'. That symbol is made up of the 'is less than' sign (<) and the 'is greater than' sign (>). Those two 'inequality' symbols appear on the top row of keys after the zero.

Line 220 Here again is an IF...THEN.... The line also uses GO TO, though I ought to tell you that such a usage is frowned upon by many modern programmers. (The use of GO TO shows a dependence on line numbers, which, though essential in BASIC, are best avoided in what we call structured programs. Later on I'll show you how to get round this problem.) The REMs that follow line 220 point out exactly what the IF...THEN... does. We can translate line 220 as follows. 'IF the answer is not equal to N1 × N2, THEN GO TO line 90; otherwise, go on.'

Line 290 It is quite in order to have RUN as a program statement. What it does, of course, is to tell the program to start again from scratch when it reaches this stage. You will often find, I think, that it is worth ending your programs in this way rather than letting them run out of steam entirely. I think this makes a program more useful, and perhaps more friendly to the user, but the user must know that the BREAK key can stop the program. The command CONT lets the program start again at the next line after it stopped with BREAK.

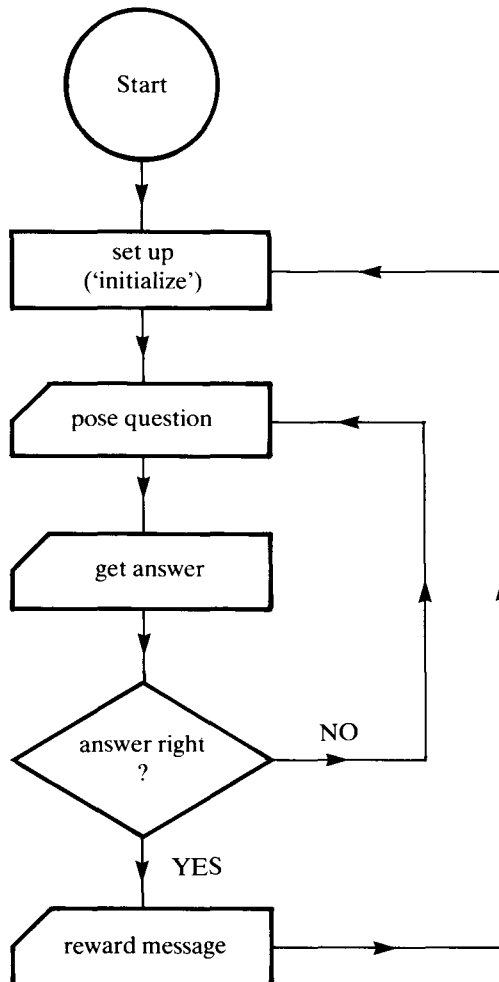


Figure 3.10 Flowchart outlining Program 4.

Finally, I think I ought to say a little bit about the keyword REM. I used it in the last program, as well as in Program 3. REM is short for ‘remark’; REM statements stay in the listing for later use by readers, but the micro ignores them during a run.

The use of IF...THEN... in a program as in line 220 above means that the program doesn’t run straight through from start to end. I think it may help you, therefore, to see a chart of this program to show more clearly what is going on.

SUMMARY

Here’s a list of the keywords that I’ve used in this chapter. I agree that I’m going to go into further detail on a number of them later, but all the same the ideas you’ve met will let you make some sort of programs up of your own. That’s the purpose of the next section!

CHR\$	NEW
CONT	POSITION
DIM	PRINT
GO TO	REM
IF...THEN...	RND
INPUT	RUN
INT	SETCOLOR
LET	THEN
LIST	

You also know about the RETURN and BREAK keys (the latter stopping a program from running, the keyword CONT letting it continue), and you’ve met some arithmetic operators. The arithmetic operators are:

+ (plus) – (take away) * (times) / (divide) ^ (raise to power) (. . .) (brackets)

Then there are the things that posh folk call logical operators. Here’s a full list of those

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

We’ve also talked about the priority list of operations and numeric and string variables. Actually, now I look at all that, it looks as if we’ve done nearly all the important things. I can’t think why the rest of this book is so long. . . .

DO IT YOURSELF

Here are some programming exercises for you to try on your own. Notes on some appear on Page 180. I must tell you, however, that in programming there is never a single correct answer to a problem.

1. Write a program like Program 1 (though don’t use GRAPHICS)—Starring YOU instead of Starring Atari.
2. Make up a poem about computing and get your computer to print it out neatly on screen.
3. Using POSITION, SETCOLOR, and PRINT, devise a program to display a very simple picture on screen. Try using all the strange symbols on your keyboard.
4. Combine the above ideas to get a program to display on screen a certificate like that in Figure 3.11. Of course if you have a printer, certificates like this may be most useful.

```

* * * * *
*
*      SAINT TRINIAN'S SCHOOL
*
*      ## CERTIFICATE ##
*      -----
*
*  Placed LAST in the . . .
*  . . . . .
*
*  Awarded to . . .
*  . . . . .
*
*  HEAD-MISTRESS .....
*  =====
*  DATE .....
*  =====
*
* * * * *

```

Figure 3.11 A computerized certificate.

5. Use the structure of Program 2 for one that suits your own needs.
6. Do the same with Program 3.
7. Study with care the listing of Program 5. Try to predict exactly what each line and the whole program actually does. Then enter the program following the listing exactly, run it, and see if you are right.

Program 5: Can you read BASIC?

```

10 SETCOLOR 4,8,12
20 PRINT "  INPUT  AND  OUTPUT"
30 PRINT "  =====  =====  ====="
40 POSITION 6,6
50 PRINT "Give a number between 1 and 20."
60 INPUT A
70 POSITION 6,12
80 PRINT "Now type any set of ten characters...."
90 DIM SET$(10)
100 INPUT SET$
110 PRINT CHR$(125)
120 LET B=1
130 LET B=B+1
140 IF B<10 THEN PRINT " ";
150 PRINT B;" Hi-di-hi, ";SET$
160 IF B<A THEN GO TO 130
170 GO TO 170

```

Please don't read these notes until you've finished the question!

Lines 20–30 Note layout tricks here.

Lines 140–150 Same applies.

Again you'll need **BREAK** to stop this program. You'll also find, as I didn't have a **PRINT CHR\$(125)** statement at the start of the program, that it's best to press **RESET** before you **RUN**.

8. Study Program 6. I trust that you will see at once that it contains errors! In fact every line except one has at least two mistakes in it. Correct the errors, enter the program, and run it. Make sure that it gives the results you would expect.

Program 6: Finding fault

```
20 PR "What's your name?"
30 INPUT name $
40 POSITON (2,2)
50 PRINT AN EMPTY LINE
60 PRINT "Age, N$?"
70 INPUT AGE$
80 IF B=10 THEN "Wish I was ten!"
90 B " is a nice age!"
100 PRINT: "Bye-bye!N$;","."
```

9. (For advanced programmers!) How did I get a print out of Program 6, if the Atari complains so often about ERRORS?
10. Use Program 5 as the basis for one giving various pleasing patterns on screen. This is your chance to explore the great potential of PRINT with 'inverse' (the effect you get with '█' or '◻') and the special 'graphics blocks' obtained when you hold down the CONTROL (or CTRL) key while pressing a letter key.
11. Practise the use of INPUT (string and numeric) in a program to ask the user various questions, storing the results, and finally printing them out neatly.
12. Devise a BASIC program to accept a number (up to, say, 500) and to print out on screen the multiplication table for that number, up to 12 times. See if you can lay the screen out neatly whatever the input number (positive or negative, whole or fraction). Base your answer on the technique shown in lines 110–170 of Program 5.
13. Develop a neat 'calculator' program which accepts two input numbers and asks how they are to be operated on (+, -, *, /, .). The program should then print out the expression it's got to work out, and give the answer. After that the program should return to the beginning for another go.
14. Devise a program to model as closely as possible (perhaps, however, without the money output!) the action of the cash dispenser system I described early in the chapter.

Note It may well be that you will feel that at least a few of the programs you have developed in this section are worth keeping for later use and polish. Chapter 5 tells you how to save them on cassette.

4 Printing Press

Using the Atari's main output instruction, PRINT, which puts what follows on screen, is not easy to come to grips with once you start digging much further than we've been so far. If you typed in and played with Program 1 (Page 10) you will have got some inkling of how complex PRINT can be.

The reason is that the micro has a large number of display *modes*. Each one is best for a given purpose, and in each case the action of PRINT can differ. First, then, we need to explore the mode thing a little bit. Then I can concentrate on those in which ordinary printing—'text printing'—is of most interest. Before we do that, I think I'll tell you about what's called looping.

LOOPING THE LOOP

I've already said that many modern programmers frown on the GO TO instruction (though I know a book which says it's the most important!). GO TO does have some good points, but I agree with people who say we should avoid it. We'll come back to this point later in the book, but when I've dealt with the concept of *loops* I'll need to use GO TO (or GOTO: you can leave the space out if you like) far less often. Look at this simple counting program. It builds up the ten times table for any input number.

Program 7: Table d'hôte

```
10 PRINT CHR$(125)
20 POSITION 2,2
30 PRINT "TIMES TABLE"
40 POSITION 2,20
50 PRINT "Please give the number you want!"
60 INPUT NUMB
70 POSITION 2,20
80 PRINT "
90 PRINT "
100 POSITION 15,2
110 PRINT ": ";NUMB
120 POSITION 2,8
130 LET T=1
140 PRINT T;" times ";NUMB;" is ";T*NUMB
150 LET T=T+1
160 IF T<11 THEN GO TO 140
170 POSITION 2,20
180 PRINT "Please press RETURN to go on..."
190 DIM CONT$(1)
200 INPUT CONT$
210 RUN
```

Because this is a chapter on PRINT, I've included one or two snazzy PRINT tricks—such as changing the title half way through the program. But now my main concern is the clumsiness of the table-printing routine, the bit in lines 130–160. Not only does this need some thought to write and understand, it uses GO TO and that could make some reviewers scream. Can't have that, can we! (You'll also find the screen layout looks rather horrid, snazzy tricks or not—but we'll be able to overcome that kind of problem soon enough.)

What do lines 130–160 do in Program 6? Here we have a loop: the program goes through the central part (140–150) a certain number of times. Round and round and round—hence the name. Maybe a picture would help—Figure 4.1 shows what's going on.

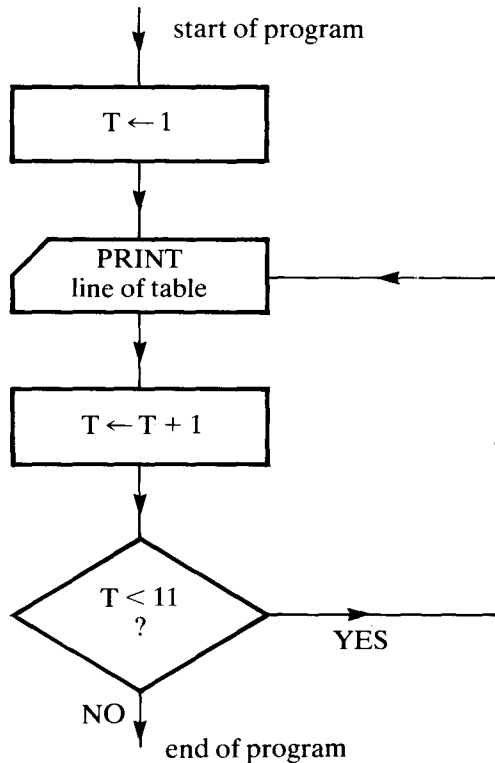


Figure 4.1

First T takes the value 1. The program then puts a line of the table onto the screen and adds 1 to the value of T, making it 2. Next we come to the diamond-shaped box—'is T less than 11?' it means. If T is less than 11, which it is, we zoom round the side back to the PRINT box: a second line of table appears like magic on display. And T then gets a new value again—3 this time. T is still less than 11, so round we go again. And so on. Until at last T reaches the value 11—which is *not* less than 11—so we fall out of the bottom of the loop.

Okay? Clever, I agree. A pretty neat scheme, which lets the computer in effect do lots of actions with only a small number of lines. But BASIC has a posher way, one that is even simpler. This uses what we call a FOR...NEXT structure. Change Program 6 like this:

130 FOR T = 1 TO 10	(this wipes out the old line 130)
150 NEXT T	(this wipes out the old line 150)
160 (R)	(this wipes out line 160 without a new version)

RUN—and you'll find the same action. So you can use FOR...NEXT loops to say how many times you want a routine of the program to repeat. The structure is very simple (and uses no GO TO!). The line

```
FOR COUNTER = STARTING-VALUE TO STOP-VALUE
```

as in

```
FOR T = 1 TO 10
```

tells the micro that the lines that follow need to be done a certain number of times, with the value of the counter going up by one each time. The line

```
NEXT COUNTER
```

as in

```
NEXT T
```

shows the end of the routine within the loop.

Some BASICs have other structures which allow a set of lines to be cycled through again and again. You can in fact mirror these with the Atari (see Appendix 7), but FOR...TO...NEXT has fair power. In fact we can extend *this* a bit, but before I deal with that I would like you to check that Figure 4.1 also shows what happens in the loop section of the new version of Program 6.

It does so exactly. Let me show you. RUN the new program. After the table appears on screen, when the program asks you to press RETURN to go on, press BREAK instead to stop it. (BREAK can always stop a program in action, unless the writer has 'disabled' it—see Appendix 7 again.) Now enter PRINT T (R) to get the value of the counter T at the end of the loop—it is 11, not 10 as you might expect. Bear that in mind if you need to use a loop counter again later in a program.

Anyway, FOR...TO...NEXT can be used with step (increment) values other than +1. The keyword involved is—surprise, surprise—STEP, and this can take any value you like. The STEP value can even be minus, and it needs to be that if the loop's STOP-VALUE is less than its STARTVALUE. But if STEP isn't there, the micro assumes its value to be 1. Here's the full structure then:

```
FOR COUNTER = STARTING-VALUE TO STOP-VALUE STEP  
STEP-VALUE
```

```
(do such and such)
```

```
NEXT COUNTER
```

Here's how this works in full.

When the micro meets the FOR, it:

- (a) puts the specified starting value into the loop counter 'box' in the store;
- (b) puts the specified final value into another box;
- (c) notes in a third box (a 'stack') where the loop starts.

When it meets NEXT, it:

- (a) checks to see if the counter value has reached the final value and, if so, leaves the loop;
- (b) if not, adds one step value to the counter value; and
- (c) goes back to where the stack tells it to.

The next routine, hardly worth calling a full program, shows an obvious use of a minus STEP value. Study it and the notes that follow—there are as usual one or two more new tricks for you.

```

10 PRINT "COUNT-DOWN"
20 FOR COUNT = 10 TO 0 STEP -1
30 POSITION 15, 10
40 PRINT "***"; COUNT; "***"
50 FOR WAIT = 1 TO 450: NEXT WAIT
60 NEXT COUNT
70 POSITION 10, 10
80 PRINT "WE HAVE LIFT-OFF!"

```

There are in fact three bonuses in line 50 alone!

Firstly, line 50, all by itself, is a FOR...TO...NEXT loop. It lies entirely inside the COUNT loop; we say the two loops are "nested".

We often need to nest loops like this, in even quite simple programs. Nothing to worry about there. Except one thing—you must not end an outer loop before you end any ones inside. Figure 4.2 shows what you can and cannot do. The trusty micro with its ERROR feature will of course soon tell you if you make a mistake like this (the error number is 13)

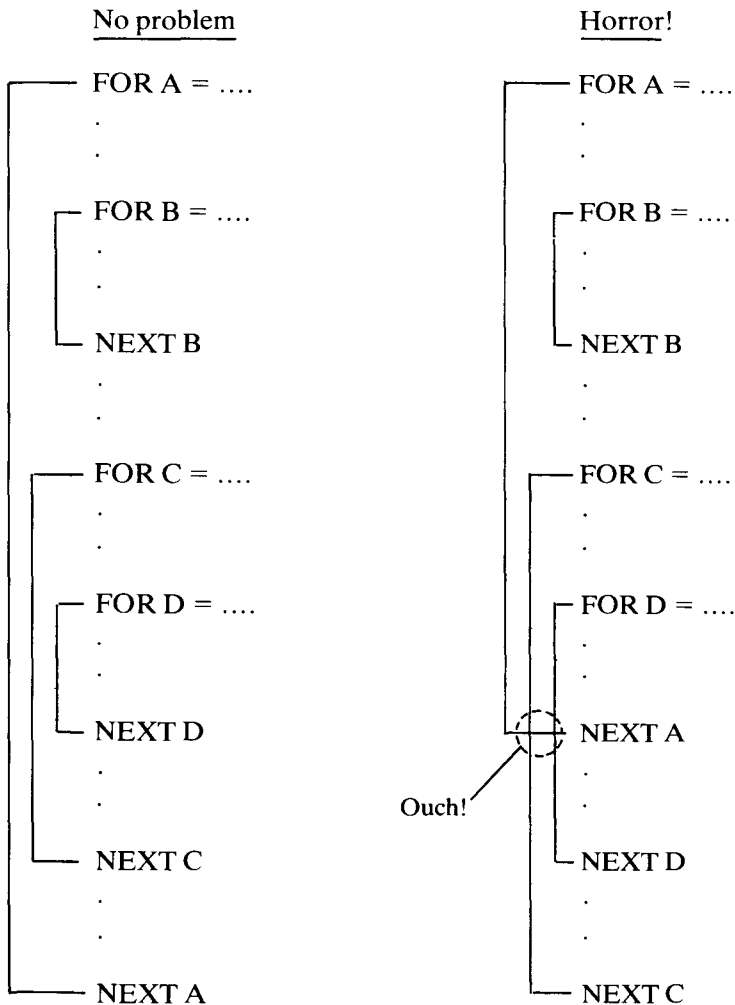


Figure 4.

but it's best not to make the mistake in the first place, isn't it?

Secondly I've fitted both the FOR statement and the NEXT statement on to a single line. The colon (:) between them acts as a separator. This is what we call the Atari's *multi-statement* feature. You can in fact have a dozen or more valid statements in one numbered line if you like, as long as the total length does not exceed about 120 characters (3 screen lines). Separate each with a colon (:). Multi-statement lines need some care and can make your program listings hard to follow. I tend to put a related block of instructions on to one line for convenience—as in 50 above, where the WAIT...FOR NEXT structure has a single function. Beware, however, when there are IFs around—multi-statement lines can then seem to go wrong; I'll say more on that in a little while. *Thirdly* is what line 50 actually tries to do. Its function is to bring a *delay* into the program, in this case a delay of close to one second. That's why I called the loop counter WAIT.

This use of FOR...NEXT, to cause the program to pause for a while, is most useful. I use it a great deal, and you'll see it a lot further on in this book. I admit I haven't timed line 50 exactly, but the delay there is indeed around one second, and I'll leave you to get the exact figure should you so need. However, it's worth remembering for the future that FOR WAIT = 1 TO 500: NEXT WAIT will give a delay of about one second. If you bear that in mind, you will have no problem in devising delays of any length of time.

Timing and loops combine in Program 7. It's a simple one, I agree, but relates to the so-called 'bench mark' programs people often use to compare the speeds of micros. And I use it as such to indicate how *you* can time different Atari functions if you want to (Test 3).

Program 8: Hard times

```
10 DIM CONT$(1):PRINT CHR$(125):GOTO 100
20 PRINT CHR$(125):POSITION 2,10:PRINT "Press RETURN and start timing..."
30 POSITION 2,15:INPUT CONT$:PRINT CHR$(253);CHR$(125):RETURN
40 POSITION 2,10:PRINT CHR$(253);"Please enter time & RETURN.":POSITION 2,15:RETURN
99 REM ** Program start
100 GOSUB 20:REM ** Test 1
110 FOR A=1 TO 2500
130 NEXT A
140 GOSUB 40:INPUT T1
200 GOSUB 20:REM ** Test 2
210 FOR A=1 TO 2500:NEXT A
240 GOSUB 40:INPUT T2
300 GOSUB 20:REM ** Test 3
310 FOR A=1 TO 2500
320 PRINT CHR$(125)
330 NEXT A
340 GOSUB 40:INPUT T3
400 PRINT CHR$(125);"R E S U L T S"
410 POSITION 2,4:PRINT "TEST 1:":LIST 110,130:PRINT " TIME - ";T1
420 POSITION 2,10:PRINT "TEST 2:":LIST 210:PRINT " TIME - ";T2
430 POSITION 2,15:PRINT "TEST 3:":LIST 310,330:PRINT " TIME - ";T3
440 FOR WAIT=1 TO 2 STEP 0:NEXT WAIT
```

Notes

This program has a number of separate sections; note how I have laid them out in the listing to make them easy to follow. Separate program blocks start at lines 10, 100, 200, 300, and 400. Multi-statement lines are common to make the listing even more easy to follow—see lines 420–440 for instance.

To save both of us a lot of trouble I have also used a couple of what's called 'closed subroutines' here. One of them starts with line 20 and finishes at the end of line 30, while the second is in line 40. Each time I need either of those 'closed subroutines' in the main program (which starts at 100) I call it with GOSUB n, where n is the number of the line at which the subroutine starts. I know that's naughty of me—we don't actually get to look at closed subroutines until Chapter 12. All the same, they are of great value, and I don't think it's too hard to follow.

I've used PRINT CHR\$(253) twice in this program—lines 30 and 40. Like CHR\$(125), CHR\$(253) gives a special effect; as you have no doubt worked out by now,

the special effect in this case is to beep the speaker as a signal to the user to do something.

The final nice thing here is the use of LIST as a program statement (instruction). See it in lines 410, 420 and 430, where in each case I ask the computer to list a couple of important lines for the user.

If you have the need to compare the speed of two different programming structures, you will find this program quite useful. All you need to do is to insert each structure in turn as line 320 in Program 7. RUN the program and obtain the timings. (All micros, including the Atari, actually have a microelectronic clock as part of the central processor. Its function is to make sure that all the computer's operations keep in step. Alas, in this case, we cannot access that clock when using the BASIC programming language.)

Next I'd like to give a program to show the use of STEP. Charles Babbage, a British engineer whom people often count as the father of computing, spent much of his life a century and a half ago trying to build a machine to do calculations like these. The effort, which failed, cost him and the British Government huge sums of money. So say 'Poor old Babbage' when you run this program. . . .

Program 9: Cubism

```
10 PRINT CHR$(125); " C U B E   T A B L E   M A K E R "
20 POSITION 2,5:PRINT "What start value";:INPUT START
30 POSITION 2,10:PRINT "What final value";:INPUT END
40 POSITION 2,15:PRINT "What step value";:INPUT STEP
50 FOR WAIT=1 TO 1000:NEXT WAIT
60 PRINT CHR$(125); "  NUMBER      CUBE"
70 PRINT "  =====  "
80 FOR NUMBER=START TO END STEP STEP
90 PRINT "    ";NUMBER;"      ";NUMBER^3
100 NEXT NUMBER
```

Notes

The strange symbol in line 90 (^) stands for what we call 'raise to the power', or more poshly 'exponentiation'. Here we are raising NUMBER (whatever it may be at the time) to the power 3, which means cubing it. For instance, 2^3 , called two cubed, means $2 \times 2 \times 2$. If you haven't found it yet, the ^ symbol is got with SHIFT and the * key.

This program works very nicely, but, as I hope you've found by now, the layout on the screen is absolutely awful. We'll have to do something about that. . . .

Finally, although this is a short program, you can get very long output. You may have the urge, for instance, to obtain a table of cubes of numbers between 0 and 10 in steps of 0.0001. After all, that's the sort of thing (though not to the same accuracy) that people had to put up with in school a decade or more ago. The key-press CONTROL and 1 will stop a scrolling screen output like this when you want, and the same combination of keys will start it again. It's just the same as when you are looking in detail at a listing that takes up more than one screen display.

A LA MODE

I have already said that the Atari micro has a large number of display *modes*. Each one is best for a given purpose and in each case the action of PRINT (and other things) can differ. The next little program is meant to let you explore these modes a little bit. Then I shall look in detail at the 'text' modes, those in which ordinary printing is most useful.

Program 10: A la mode

```
10 DIM CONT$(1)
20 FOR MODE=0 TO 8
21 REM ** XL micros can go to 15 here
30 GRAPHICS MODE
40 FOR WAIT=1 TO 1000:NEXT WAIT
50 POSITION 5,5:PRINT #6;"Mode ";MODE
60 FOR WAIT=1 TO 1000:NEXT WAIT
```

```

70 PRINT "Mode ";MODE
80 FOR WAIT=1 TO 1000:NEXT WAIT
90 NEXT MODE

```

Notes

The main loop of this program is `FOR MODE = 0 TO 8` (line 20) to `NEXT MODE` (line 100). Nested inside this (lines 40, 70, 90) are three delay loops.

What this program does, therefore, is to cycle through the different modes, 0 to 15, and in each case carry out one or two simple `PRINT` operations. Amongst these is the beep, produced by `PRINT CHR$(253)` in line 30, to announce that a new mode has arrived. Study the output of this program with care.

Program 10 shows that the Atari display system has sixteen main modes of action. They are quaintly numbered 0–15; `GRAPHICS` is the BASIC statement that sets each one up. This is what `GRAPHICS n` does, `n` being a number from 0 to 15:

- clears the screen (but see below) using the standard switch-on colours;
- reserves the amount of memory needed by the mode in question (see Table 4.2);
- switches on the cursor (square block) if it was off;
- opens the data channels needed for the mode in question (see below).

Mode 0 is the so called ‘pure text mode’ you get on switch-on or after using the (SYSTEM) RESET button in the array at the right. The others are display screens on to which you can plot lines and such. Chapter 17 deals with that. However the bottom four lines of screen in most of these modes stay in mode 0. That block of lines goes by the name of the ‘text window’.

Table 4.1 shows that there are other modes, however, related to the main ones 0–15. These extra options allow you to avoid the text window of modes 1–15, or to avoid the screen clear action of 0–15, or to avoid both in 1–15. The value of `n` after the `GRAPHICS` keyword is, in these three cases, 16 more, 32 more, and 48 more than the usual.

Table 4.1 Main and other modes

Main mode	No text window	Without screen clear	Neither
0	doesn't apply	32	doesn't apply
1	17	33	49
2	18	34	50
3	19	35	51
4	20	36	52
5	21	37	53
6	22	38	54
7	23	39	55
8	24	40	56
9	25	41	57
10	26	42	58
11	27	43	59
12	28	44	60
13	29	45	61
14	30	46	62
15	31	47	63

In this chapter I'm sticking to text output, so I'll leave the graphics action until later (Chapter 17, as I say). PRINTing in the text windows in modes 1–15 is by what's called Channel 0. This is the channel that deals with the whole screen in mode 0. Thus text window action is just like Mode 0 action and follows the same rules—except you have only four lines on screen to play with instead of 24. Table 4.2 gives a summary of what you need to know about Modes 0–15.

Table 4.2 Structure of main modes

Mode	Action	Memory demand	Number of screen sites	Colours
0	Text	992	24×40	3
1	Big text	672	$20 \times 20 + 4 \times 40$	5
2	Large text	480	$10 \times 20 + 4 \times 40$	5
3	Low resolution graphics	432	$20 \times 40 + 4 \times 40$	4
4	Higher resolution	696	$40 \times 80 + 4 \times 40$	2
5	Higher resolution	1176	$40 \times 80 + 4 \times 40$	4
6	Higher resolution	2184	$80 \times 160 + 4 \times 40$	2
7	Higher resolution	4200	$80 \times 160 + 4 \times 40$	4
8	Highest resolution	8138	$160 \times 320 + 4 \times 40$	3
9	Lower resolution	8138	192×80	1
10	Same	8138	192×80	9
11	Same	8138	192×80	16
12	As Mode 3	1152	$20 \times 40 + 4 \times 40$	5
13	Even lower	660	$10 \times 40 + 4 \times 40$	5
14	Medium	4296	$160 \times 160 + 4 \times 40$	2
15	Medium	8138	$160 \times 160 + 4 \times 40$	4

In the first column I list the mode numbers; and in the second the effect that can be produced. In this chapter I shall explore the three text actions given by modes 0–2. All I'll say now about the graphic modes (3–15) is what 'resolution' means. *Resolution* is a measure of the closeness of points that can be plotted on screen. In the lowest resolution mode, 3, the graphics area consists of only 800 (20×40) points. In the highest, Mode 8, we can have 51200 points (160×320). *Note* that some earlier versions of the Atari microcomputer range cannot support that last mode or the ones after it because they do not have enough memory.

Memory is the subject of the third column, which sets out the number of units of memory that the Atari needs to support each mode. This memory need is supplied by RAM; I ought to say something about all this, so see the Jargon Store section below.

The fourth column of the table gives the number of screen sites into which material can be placed. In each case the first figure gives the number of lines (or rows) while the second shows the number of columns. For modes 1–8 and 12–15, there are two sets of figures; the second, 4×40 , is the number of lines and columns in the text window. If you do without the text window, using Modes 17–31 or 49–63 (see Table 4.1) then the number of lines in the main screen goes up, to use the extra space.

The final column, called 'colours', gives the numbers of colours that can be on screen at the same time.

JARGON STORE

A computer has to be able to store the program instruction and data it is working with at any given time. Also it has to store the 'operating system' instructions, the details of how to carry out all the tasks it can be given. As far as the computer is concerned all these

items are data—electronic representations of binary numbers (0s and 1s). The computer circuits handle strings of binary numbers following certain rules. A binary number, one which can take the value 0 or 1, has the name 'bit' (= binary digit).

A *byte* is a set of eight bits. The value of a byte can range from 0 (0000 0000) to 255 (1111 1111). In the case of most common micros including your Atari, the byte is the same as a word, a set of bits of standard length. (You may also come across a *nibble*, which is half a byte (4 bits). What a charming name!)

We measure the amount of data that a micro can carry in its store in bytes. In most cases the maximum size is 65536. Computer folk call this 64K bytes, or 64K for short. Here the symbol K stands for Kilo; one Kilobyte is 1024 bytes. This strange figure (which happens to be 2 multiplied by itself ten times) is close to 1000, which we call kilo; in the case of computers we use the capital letter to remind us that the figure is a bit bigger than 1000.

A computer's main memory is commonly divided into two parts. The operating systems (including the program that translates BASIC statements) must be permanently fixed. They are fixed in chips called 'read only memory' (ROM). The Atari ROMs are around 24K meaning that they contain 24000 or so bytes of operating instructions which can't be changed. The contents of ROM are truly fixed, even when the power is off.

With 24K of the Atari's 64K maximum taken up by ROM no more than 30K is left for the user's programs and data. We must of course be able to change these, so ROM is no good. Instead we have RAM. RAM is 'read and write memory'—the contents can be got at (read) and also changed (written into). (Some people think that RAM stands for 'random access memory' but that is not so. Both RAM and ROM are random access memory—we can reach any part of them at once without starting from the beginning.)

As RAM chips are fairly costly, micro manufacturers do not automatically provide enough RAM in their boxes to make up the size to 64K. Thus the Ataris are sold in various forms, with less or more chunks of RAM available. Please note, however, that RAM size does not actually tell you how much memory you can use yourself—the computer itself needs quite a few K of RAM for its own seedy purposes, as Table 4.2 reminds you.

After all that, it is time to get back to this chapter's main concern—putting text on screen.

TODAY'S TEXT

Now I shall deal only with the use of PRINT, and the statements that relate to it, in the text modes 0–2. First let me sum up, and in some cases extend, what we know so far of work in Mode 0—the standard text mode. I shall also tell you the short forms of the keywords used, so that you can enter these to save time and typing trouble. In fact most Atari keywords have short forms, the first few characters and a full stop. Note that you don't *have* to use a short form, and indeed the micro will change any short forms to the full keywords, when LISTing. Also note that you can use longer short forms than the shortest, if you see what I mean. For instance, the short form of GRAPHICS is GR., but you can use GRA., GRAP., and so on, as you please.

POSITION (short form POS.) comes with two numbers after it, the column and the row in which the next bit of printing is to appear. Thus

POS.17,11: PRINT "TEST"

puts the word TEST at the centre of the Mode 0 screen.

You will recall that the screen contains 24 lines of 40 character sites. However, note that the first line bears the number 0, and so does the first column—the extreme top left is therefore 0, 0. Figure 4.3 shows this. You may observe my next two points for yourself if you enter, RUN, and then LIST this little program.

```
10 FOR LINE = 0 TO 23: POSITION 0, LINE: PRINT LINE: NEXT LINE
20 FOR WAIT = 1 TO 2000: NEXT WAIT
```

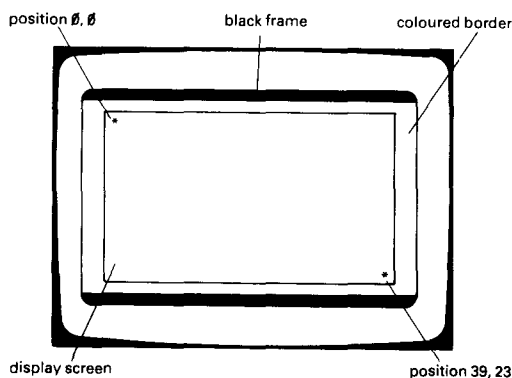


Figure 4.3

Firstly, the first two columns in Mode 0 normally stay empty—but you can access them with POSITION. (The reason for leaving these two columns blank seems to be that with some early Ataris, the left hand side of the display could lie off the TV screen.)

Secondly, if you print down to the bottom line of the display screen the screen contents can start to 'scroll' up to allow new material at the bottom. Thus, when the above program reaches the end of line 10, the number at the top of the column becomes 1 not 0, to allow room for the cursor. Then at the end of line 20 the display scrolls up two more lines to get the usual blank line and READY report in.

You can use a grid like that in Figure 4.4 by all means, and plan your layout of Mode 0 text. Indeed, you are free to copy this one. Take great care when using the bottom line, numero 23, not to wander by accident below it—if you do, you'll lose your top line and later printing won't appear at the sites you expect.

POSITION X, Y, then, warns the micro that the next bit of printing to be done should start (X - 1) places across the screen and on line (Y - 1). The values of X and Y should be within the ranges 0-39 and 0-23 respectively. If you try other values, the micro will shout ERROR at you.

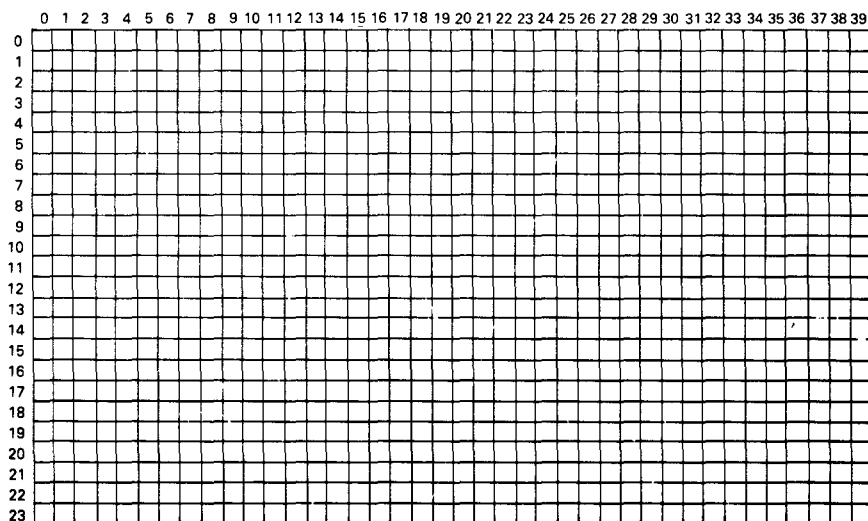


Figure 4.4

SETCOLOR (SE. to its friends) needs 3 numbers after it—the first being 1, 2, or 4 to say whether it's to work on text (foreground), screen (background), or border (see Figure 4.3). The second is the colour number (0-15, as in the list on Page 17). The third is the brightness value (0-15 again but in steps of 2, from dark to light).

Some people claim that the Atari offers 128 or 256 different colours. That's surely stretching the truth a great deal. After all, even the 16 colours listed do not differ hugely and you can't get nice primaries unless there's something greatly wrong with your telly! Anyway, that's what we have to play with, eight shades each of about 16 'colours'. And in Mode 0, SETCOLOR's the tool with which we play with them.

You've had a fair bit of practice with SETCOLOR already, so I'll now make only two points, one a repeat, and the other new. The former is that in Mode 0, the ink colour has to be the same as that of the screen area. All SETCOLOR 1... can do, therefore, is to affect the brightness of text. Still, that does give us the chance of flashing on screen text that was printed invisibly. Like this:

```
10 SETCOLOR 1,0,0:SETCOLOR 2,0,0:SETCOLOR 4,0,0: REM All black
20 POSITION 17,11:PRINT "TEST": REM Invisible printing
30 FOR W = 1 TO 1000: NEXT W: REM delay
40 SE.1,0,14: REM Let there be light
```

My second point is to introduce to you the concept of colour 'registers'. There are five of these (you can guess what their numbers are), each a special site in the micro's store set aside to carry colour and brightness data. The first number that follows a SETCOLOR instruction is in fact the register concerned. So in Mode 0 the registers are as follows:

Table 4.3 Mode 0 colour registers

Register	Usage	Start up value
0	not used	—
1	text	pale blue
2	screen	blue
3	not used	—
4	border	black

Now I can repeat the first point in a new way. In Mode 0, register 1 carries only brightness data; the colour of register 1 (text) is the same as that of register 2 (screen).

PRINT (PR. for short) is the actual output instruction. It needs a section all to itself.

PRINTED MATTER

Please refer again to Program 3 on Page 28; this program includes all the features of PRINT in the list that follows, except the first, the simplest. PRINT can be used as follows:

1. On its own: PRINT (R). This cause a screen line to be left blank. Try it as a direct command now! We can therefore skip several lines on a screen if we want by a structure such as this PR.: PR.:PR. (in this case leaving 3 lines).
2. With a message (a string constant) as in line 60. This makes the message alone appear on screen. After PRINT, the message must appear in speech marks ("..."), once PRINTED, the speech marks do not appear.
3. With a numeric or string variable name, as inside lines 160 and 100 respectively. This causes the micro to display the current value of that variable.

4. With some mixture of numeric and string variables and constants (as in line 160). Each 'print item' needs to be separated from the others by a semi-colon (;) as here, or in fact, by one or more commas(.). A semi-colon when between two print items causes them to appear on screen without a space between; commas separate the displayed items into zones (columns) across the screen—I'll come back to that again in a moment. Note that commas and semi-colons can appear at the end of a PRINT statement as well as in the middle—see line 120, for instance—and commas can even appear straight after PRINT. (A semi-colon at the beginning of a PRINT statement, is, however, meaningless, and may cause an ERROR.)
5. After POSITION: the next PRINT item to appear will start at the line/column given by POSITION. The example in Program 3 line 90 makes the 'T' of the message in line 100 appear in the fifth place of line 9. (Recall that the first line and the first column on screen each carry number 0.)
6. With CHR\$(...) to cause some special effect. CHR\$(125)—as in line 40—clears the screen. We have also met PRINT CHR\$(253) to buzz the speaker. Use the 'clear screen' facility of PRINT CHR\$(125) often to make room for more material, or to hide earlier material—and to stop clutter. (I've also used PRINT to clear part of the screen in one or two other programs—can you remember the system?)

The careful use of PRINT is an essential part of screen layout (formatting); I think that all programs you write need very good screen design, to maximize communication and therefore value to their users.

If you use PRINT... alone it will cause the print item(s) that follow to appear at the next available screen position. Often that's okay—but by no means always. To lay out material displayed on the screen, in time and space, needs various additions to the simple PRINT statement.

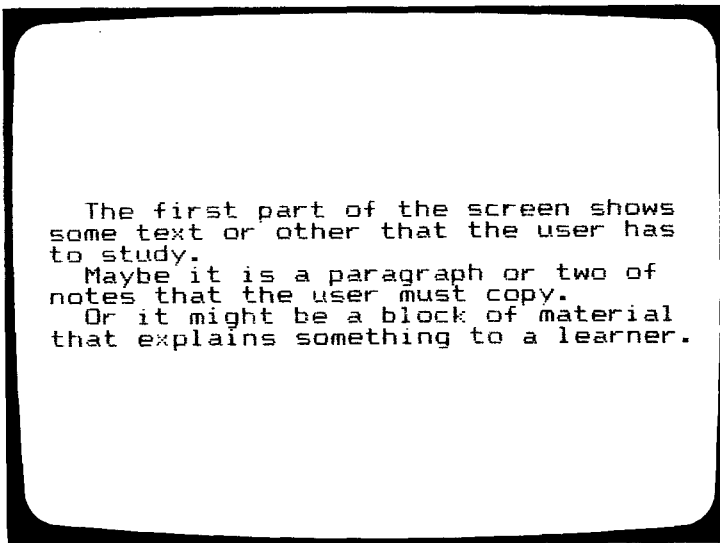


Figure 4.5

Say that in a program you've got a screen looking like the one in Figure 4.5 and now you want to display the message "Press RETURN to go on." If you just use PRINT "Press..." as the next line in the program, the display that results will be like that in Figure 4.6. For a start that's not very pretty. But much more important, the last message is hard to see. Many users wouldn't even notice the extra line, and might sit there forever thinking what to do. As I've already said, you could first use PRINT CHR\$(125) to empty the screen—but perhaps you want the original text to remain when new stuff is added. What we need is a set of 'steps' within the 'frame' like those in Figures 4.7 and 4.8.

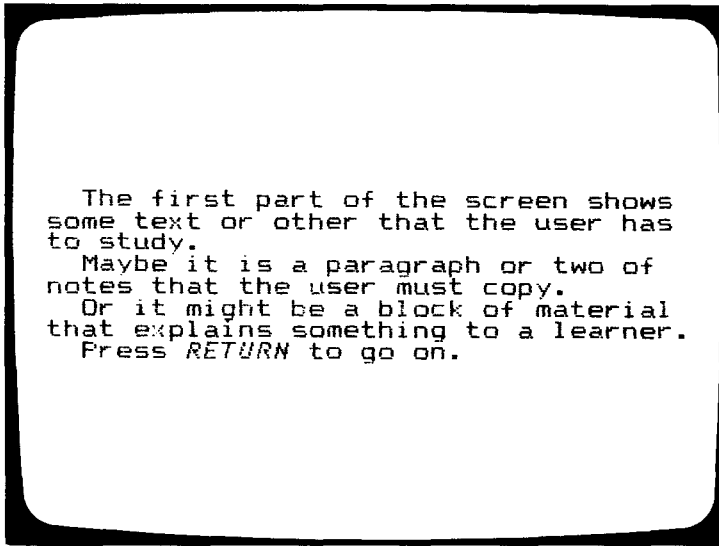


Figure 4.6

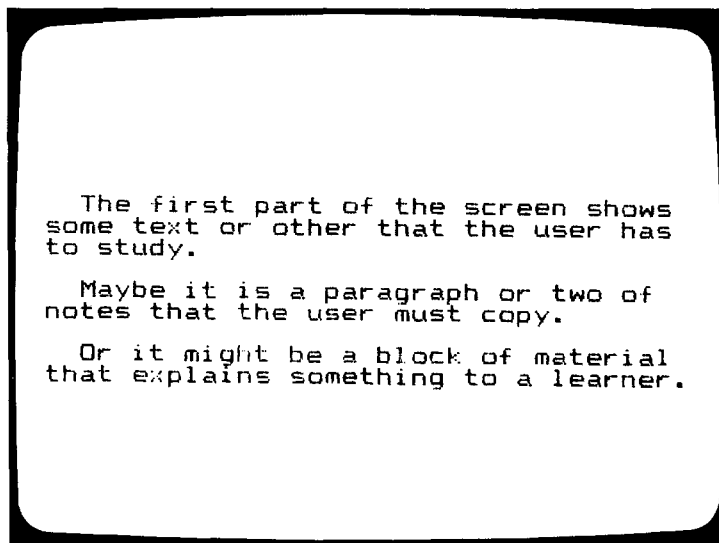


Figure 4.7

Firstly I've put a blank line between each of the first paragraphs, not (wastefully) by using PRINT "(40 spaces)", or "PRINT (R)", but with PR.:PR."next paragraph".

Secondly I use POSITION to move the "Press RETURN..." message to the foot of the 'page'. What I did in fact was

POS.4,21: PR. "Press RETURN to go on."

This starts the message appearing 22 lines down from the top and 4 spaces across from the left (bearing in mind that the top row is 0 and so on). This gives us the bottom line in Figure 4.8.

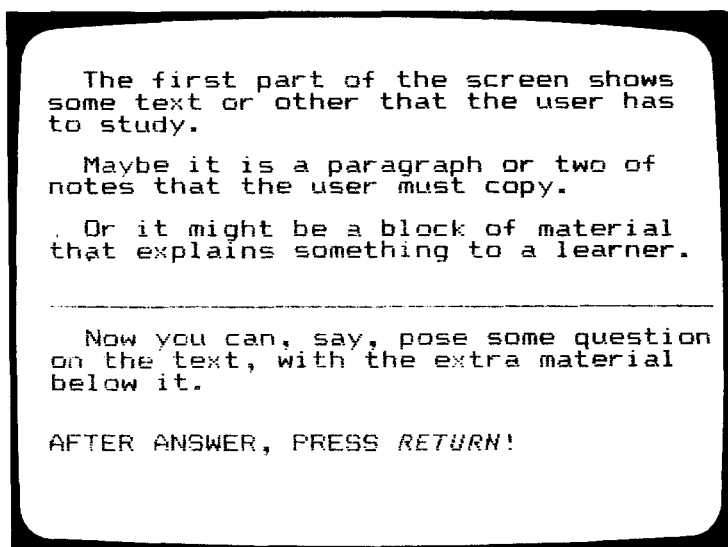


Figure 4.8

After the user has pressed (R) to go on, he or she should no longer see that outdated message. I used

```
POSITION 2,21: PRINT E$
```

```
POSITION 2,22: PRINT E$
```

To get rid of it and the old prompt. Here E\$ is "(30 spaces)". Then I ruled off the old text from the new with

```
POSITION 0,11: PRINT "(40 underlines)": POSITION 4,13:
PRINT "(next paragraph)".
```

Thirdly I have used a bit of *inverse* material in the print items. Inverse is a very simple and neat way of picking material out on the screen. Use it too inside REM statements to make those stand out clearly in a long listing.

It is easy enough to obtain inverse characters. Just before you want them, press the key at the bottom right of the keyboard, marked with the strange flag-like symbol (or the 2001 style Atari symbol on older micros). Press the same key again to revert to normal style. Use inverse often (but not *too* often) to help catch the eye of the user.

TAB

I expect you'll guess that this relates to the 'tab' function of a posh typewriter—it is just the same sort of thing, to set the next print position so many places across the page or screen. I still can't work out whether it's a good thing or not, but the Atari computers in fact have two quite separate tab systems.

The first is the simple one that we have already mentioned—the use of commas(,) in PRINT statements. This is fairly simple, and on the whole I would suggest you use it rather than the other. And what is the other? It is to involve the TAB key at the left hand side of the keyboard.

If you press that key now a few times, you will see that the cursor skips across the screen to pre-set positions. In just the same way a typewriter with the tab feature will let its print-head skip across the paper to pre-set positions when you press the right key. To cause the print position to skip to a 'TAB', use PRINT CHR\$(127). Try it—try

```
PRINT CHR$(127); "something or other".
```

That's okay once in a while, but if you wish to use the TAB feature a lot that's going to give you a great deal of typing to do. The Atari has a short way, one that uses the ESC key at the top left hand corner of the keyboard. The short-hand system I am now going to tell you about applies to all those CHR\$ control codes we've mentioned. What you do is first press the ESC key, then let it go, and then press whatever's next. *This must be done inside speech marks.*

Let's try it. Enter PR," , press ESC (no effect on screen), press TAB, then type a key or two then close the ". The sequence ESC followed by TAB, which I'll write ESC/TAB, make a sort of right-pointing arrow appear on screen. When you press RETURN, the characters after that arrow appear to have shifted to the right of the usual position. Try something a little more complex:

```
PRINT "► 1 ►► 2 ►►► 3" (R)
```

this makes a '1' appear in the sixth site of the line, '2' well over towards the right and '3' at the start of the next line. Now try this:

```
PRINT ,1,,2,,,3 (R)
```

Again the digits 1, 2 and 3 are separated out over two lines, but differently.

What a comma does between two print items is to separate their starting point by ten character positions. We say that the Mode 0 display screen, in full 40 sites wide, breaks down into four zones. Each comma in a PRINT statement moves the print position to the start of the next zone.

So the comma system is simpler than TAB (or its equivalent CHR\$(127)). However even using commas has a problem—and that arises from the fact we have already discussed, that for historical reasons the Atari print lines start two spaces to the right of the display area. That means, therefore, that the user of commas as print separators is confused—the actual line length is not 40, but 38. The next program shows this and other points I have made in this section.

Program 11: TAB confusion

```
10 POKE 82,0
20 DIM MARKER$(38)
30 LET MARKER$=".....!.....!.....!.....!"
40 PRINT "j"
41 REM ** That symbol is ESC/CLEAR, an arrow pointing up and left
50 PRINT MARKER$
60 PRINT 1,2,3,4,5,6,7,8,9,10,11,12,13
70 PRINT MARKER$
80 PRINT " 1 2 3 4 5 6 7 8 9 10 11 12 13"
81 REM ** Before each number there is the right-pointing arrow got with ESC/TAB
90 PRINT MARKER$
100 POKE 82,2
110 PRINT
120 PRINT MARKER$
130 PRINT 1,2,3,4,5,6,7,8,9,10,11,12,13
140 PRINT MARKER$
150 PRINT " 1 2 3 4 5 6 7 8 9 10 11 12 13"
151 REM ** Before each number there is the right-pointing arrow got with ESC/TAB
160 PRINT MARKER$
170 PRINT
180 GOTO 200
```

Notes

If you enter, RUN, and study the program and its output with care, you will learn a lot about how the two Atari TAB systems work. As I have said before, I would suggest you stick to the one using commas rather than the TAB key (►). The latter is complex, and therefore in advanced use can be of great value. (For instance, you can change, create, and delete tab sites within a program as you wish.) The program also makes one or two other points.

POKE 82,0 (line 10) tells the Atari to stop messing around with a left margin two spaces from the edge of the screen, but rather to start printing right at the edge. In line 130 I use the same structure to go back to the normal system—POKE 82,2 to put the left margin two spaces in. (You may also care to note that POKE 83... deals in the same kind of way with the right margin, the normal value being 39.) Try, for interest,

POKE 82,9; POKE 83,29: LIST(R)

this will give you a nice narrow listing down the centre of the screen. Press the (SYSTEM) RESET key to get back to normal.

In line 40 I use a second of those funny ESC functions. PRINT “␣” is an alternative for PRINT CHR\$(125)—in other words it will clear the screen. You get this symbol in your listing, and therefore in the computer’s memory by using (inside the speech marks, of course) ESC followed by SHIFT and CLEAR. The CLEAR key is on the top row near the right; in direct mode, as you may have found, SHIFT and CLEAR will empty your screen very quickly. Finally, note that the vertical line I use inside MARKER\$ is got with SHIFT and the = key.

Let’s summarize PRINT in Mode 0 again. Then we’ll have some more programs.

PRINT WHAT?

The Atari print instruction can be used as direct command or as a program statement. Used alone, it causes a line to be skipped; otherwise it must be followed by one or more print items:

1. A number (numeric constant), as in PRINT 5, causing the number to be displayed.
2. A string constant, as in PRINT “X”—causing the string to appear.
3. A numeric expression, as in PRINT 4 + SQR4, causing the result to be printed.
4. A string expression (for examples see later on), causing the result to appear.
5. A numeric variable, as in PRINT AGE—printing value of that variable, if it exists; otherwise printing 0.
6. A string variable, as in PRINT ANS\$, doing the same for a string (but in this case the “otherwise” gives nothing).

Here numeric and string expressions can be as complex as you like, but each term must be of the same type (in other words, numeric or string). So you can’t have PRINT “the total is” + 4. However, as we shall find out later, you can change a numeric expression to a string, and you can change a string expression into a number, if that has any meaning.

Individual print items in a mixed set must be separated from each other using, surprisingly, *separators*. These are:

1. Semi-colon (;) as in PRINT “answer”; ANS—giving no separation on screen.
2. Comma (,) as in PRINT X,Y—giving separation into zones.

Good *format* (screen layout) may also come using POSITION and/or “TAB”:

POSITION 5,5; “Hello ►► Dolly!”

Get the “►” symbol in a string constant using ESC/TAB.

Program 12: Tom Tiddler’s ground

```
10 SETCOLOR 1,0,14:SETCOLOR 2,13,2:SETCOLOR 4,13,0
20 PRINT CHR$(125)
30 FOR A=1 TO 100
40 POSITION RND(0)*39,RND(0)*23
50 PRINT CHR$(253);"$";
60 NEXT A
70 POKE 755,1:POSITION 0,0
71 REM ** Makes cursor vanish
80 GOTO 80
81 REM ** Stops READY message
```

My apologies to British readers—the Atari is a US micro, and does not readily offer the £ sign. Otherwise, there is nothing special of note in this program, other than the use of POKE 755,1 (in line 70) to hide the otherwise intrusive cursor and RND(0) in line 40. I've dealt with RND(0) already, and you should not find it too hard to follow what's going on here.

Program 13: Simple Simon

```
10 PRINT CHR$(125)
20 DIM NAME$(15)
30 POSITION 2,2:PRINT "Hallo! My name is Simon."
40 FOR W=1 TO 500:NEXT W
50 POSITION 16,8:PRINT "  ":POSITION 2,8:PRINT "What is yours?";INPUT NAME$:IF
LEN(NAME$)<3 THEN GO TO 50
60 SETCOLOR 1,0,14:SETCOLOR 2,8,0:SETCOLOR 4,8,2:PRINT CHR$(125)
70 FOR LINE=3 TO 12
80 POSITION LINE+10-LEN(NAME$)/2,LINE*2-4:PRINT NAME$
90 NEXT LINE
100 FOR W=1 TO 500:NEXT W
110 SETCOLOR 2,4,0:POSITION 2,22:POKE 752,1:PRINT "Such a pretty name...."
120 FOR HALT=0 TO 1 STEP 0:NEXT HALT
```

While this is not exactly a world-shattering program, it is quite pleasant, and uses a few nice tricks. You may particularly find it worth putting into some programs of your own. The tricks include these:

- Line 50 a 'mug-trap'. The program will not proceed past here unless the user enters a string of three or more characters. The structure involves LEN(NAME\$), a 'string function' that gives the length of the string given in the brackets. Can you work out what the first two statements of the line are for?
- Lines 60 & 110 SETCOLOR used for a change of scene within a program.
- Line 80 Very complex numeric expressions for the X,Y values of POSITION. This line, with line 40 of the program before, shows that you can do all kinds of things with POSITION.
- Line 110 POKE 72,1: much the same as POKE 755,1—for getting the cursor out of the way.
- Line 120 A variation of the use of FOR...NEXT to give a delay. In this case the delay is forever. You'll need to press BREAK or RESET to get out of this.

Program 14: Favourite colour

```
10 DIM MYCOLOR$(4):LET MYCOLOR$="BLUE"
20 DIM MESSAGE$(20):LET MESSAGE$="What colour do you like best"
30 DIM ANSWER$(8)
40 PRINT CHR$(125):FOR GO=0 TO 1 STEP 0
50 PRINT MESSAGE$;:INPUT ANSWER$
60 IF ANSWER$=MYCOLOR$ THEN LET GO=2
70 NEXT GO
80 SETCOLOR 1,0,0:SETCOLOR 2,8,8:SETCOLOR 4,8,10
90 FOR W=1 TO 500:NEXT W
100 PRINT :PRINT :PRINT MYCOLOR$;"'s my favourite too.":POKE 752,1:PRINT :PRINT
"Glad we agree...."
110 FOR H=0 TO 1 STEP 0:NEXT H
```

Yet another use of the FOR...NEXT loop appears here—in this case to model the important BASIC loop structure REPEAT...UNTIL that, alas, the Atari does not offer. See it in lines 40–60. FOR GO=0 TO 1 STEP 0...NEXT GO makes the program go round and round the GO loop forever (as we saw in the last line of the previous program, and again in the last line of this one). This time, however, we can break out, not just by leaving the program for good, but by giving the correct answer to the question. That is done in line 60.: IF...THEN LET GO=2. What I've done is to make

the value of GO greater than the end value of the loop (that being 1). So when the computer sees NEXT GO again, it assumes that the loop is finished and gets on with the rest of the program.

ALL UNDER CONTROL

You may have found out that pressing CONTROL and a letter key at the same time gives a strange symbol. We call these symbols 'graphics blocks'.

If you want to build up pictures using the normal keyboard characters, you'll find it fairly hard. It is not easy in most cases to use an alphabetic character as part of a picture design. You may one day find out how to change characters to your own design, but in the meantime let's look at the ones Atari gives us with use of the CONTROL key. First enter Program 15.

Program 15: Pattern generator

```
10 PRINT CHR$(125)
20 POSITION 2,2:PRINT "Enter a string of ten characters!"
30 DIM SUB$(10):DIM CONT$(1):DIM E$(30):LET E$=""

40 FOR RPT=0 TO 1 STEP 0
50 POSITION 2,8:PRINT "          ":POSITION 2,12:PRINT E$;E$;E$;E$;E$
$
60 POSITION 2,8:INPUT SUB$
70 POSITION 2,12:PRINT "Press Y & RETURN if you are happy withthis. Otherwise p
ress RETURN and try again."
80 POSITION 2,16:INPUT CONT$
90 IF CONT$="Y" OR CONT$="y" THEN LET RPT=2
100 NEXT RPT
110 POKE 755,0:POKE 82,0:REM ** Clears cursor & changes left margin
120 SETCOLOR 1,0,14:SETCOLOR 2,13,0:SETCOLOR 4,13,0:PRINT CHR$(125)
130 FOR PRINT=1 TO 96:PRINT SUB$;:NEXT PRINT
140 POSITION 0,0:PRINT SUB$(1,1);
150 FOR H=0 TO 1 STEP 0:NEXT H
```

Notes

I would like to draw your attention to the great effort I have made to lay out the opening screen really neatly. Material is well separated, and stuff no longer needed is erased.

The RPT loop between line 40 and 100 is similar to the GO loop in the last program—the program goes round and round the loop, forever, or until some condition is met. In this case the condition is that the user is happy with the input string.

Note OR in line 90. I think that what this means is quite clear—anyway the structure allows the user to be either in upper case mode or in lower case mode. After all, in a program like this, one can get extra characters by using lower case.

The last two lines are a (not quite successful) attempt to stop any break up of the final pattern by the ending of the program. We do in fact get 21 lines of pattern, but it's too complex getting the full 24 allowed.

Again the program closes with a loop forever instruction; escape with BREAK or RESET. The latter is better.

This program is quite simple to use. On running, it invites you to enter a string of ten characters, and you get the chance to correct the entry *after* pressing RETURN as well as before. Note that if you enter more than ten characters the DIM statements at the start of the program will make it ignore the excess. However, I should really have put in a statement to check that the length of the input SUB\$ is not less than 10. I'll leave that to you—after all, the program doesn't work properly if SUB\$ is shorter than it should be.

Many people would be quite happy to enter keyboard characters to make up SUB\$. After all there are quite a few nice ones for this purpose—the ones above the number keys, and some of the other symbols in particular. A few of the numbers and some of the letters (either upper or lower case) can also produce good patterns.

However, I really brought the program in here to let you explore the use of

CONTROL with other keyboard keys. Holding CONTROL down while pressing many main keys produces those ‘graphics blocks’ I mentioned. Try it in the input section of the above program, as often as you wish.

29 keys work with CONTROL in this way—all the letter keys and those giving comma, full stop and semi-colon, as in Figure 4.9.

This shows you the different shapes you can get by using CONTROL with one of the 29 keys concerned. 29 pretty shapes for you to get, then! But in fact there are 29 more—that’s why I also show the ‘inverse’ key in the sketch. As I briefly noted before, using this key is rather like using the CAPS key—press it to get the effect you want, do your typing, and press again to get back to normal.

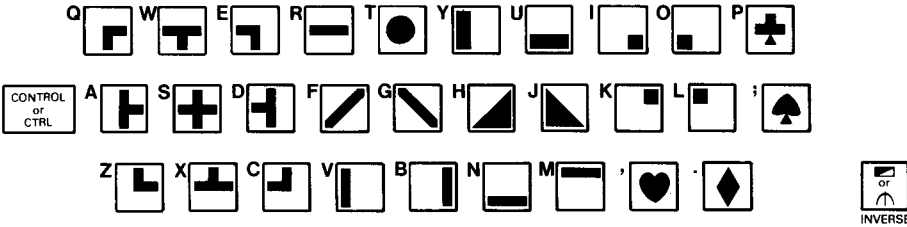


Figure 4.9

In the case of the INVERSE key, pressing it will make what you type thereafter ‘black on white’ instead of ‘white on black’ (the actual colours depend on the colours you’re using at the time, of course). And pressing it again will send you back to normal. This inverse feature also applies to the graphics blocks got with CONTROL. It took me long enough to draw Figure 4.9, so I’m not going to have a bash at doing the same thing for the inverse graphics blocks—I leave it to you to explore this.

Table 4.4 Under control?

Key combination	Effect	Inside strings	Effect
CONTROL and T etc (Figure 4.9)	blob etc	CONTROL and T	same
CONTROL and 1	stop/start scroll	ESC/CONTROL and 1	cuts out/in keyboard
CONTROL and 2	buzz buzzer	ESC/CONTROL and 2	same
CONTROL and 3	confusion!	not accepted	—
CONTROL and <	clear screen	ESC/CONTROL and <	same
Used in editing		Used in advanced program strings	
CONTROL and -	cursor up	ESC/CONTROL and -	same
CONTROL and =	cursor down	ESC/CONTROL and =	same
CONTROL and +	cursor left	ESC/CONTROL and +	same
CONTROL and *	cursor right	ESC/CONTROL and *	same
CONTROL and TAB	clear TAB site	ESC/CONTROL and TAB	same
(SHIFT and TAB	set TAB site	ESC/SHIFT and TAB	same)
CONTROL and DELETE	delete next character	ESC/CONTROL and DELETE	same
CONTROL and CAPS (CAPS alone to get back)	to graphics mode	—	—

You'd be a rare genius if you could keep in mind which letter key each of those graphics blocks is on. (If you're anything like me, you can't even find the J key!) Here's a tip if you feel artistic. Get a bottle of white typing correction fluid and very neatly mark on the keys the graphics blocks they give. Put them on the top left corners if you're right handed and top right if not. The designs will wear off before you want to sell the machine!

While I'm on the subject of the CONTROL key, it is worth giving you a list of all the different things that this can do. Some of them we'll come back to in due course (those concerned with what I call 'editing'), but the full list is in Table 4.4.

Note

Some of the above key combinations give interesting characters on screen—but you cannot get at these for making patterns or pictures of your own.

YOUR OWN THING

So far we have learned how to PRINT any of the following types of character on screen:

- (a) numbers
- (b) upper case letters
- (c) lower case letters
- (d) keyboard symbols
- (e) any of the above in inverse form
- (f) graphics blocks

Any of those can be part of string constants (in other words, inside speech marks) or as REMs. The instructions for printing each of them must of course be held in the micro's memory. That is indeed the case, and it is why you can also print any of them by using a structure of the form `PRINT CHR$(...)`. What that instruction does is to tell the computer to print the character whose 'code' is such and such. Thus `PRINT CHR$(65)` will put A on screen, as 65 is the code for 'A'. If you look at Appendix 5, you will find details of all the Atari characters and their codes.

In the notes on Page 45 I mentioned that computers like the Atari, which have a 'word' of eight bits (binary digits), can have 256 different words. These range in value from 0 (in binary, 0000 0000) to 255 (1111 1111). Some micros waste many of those 256 different possibilities but the designers of Atari BASIC have not done that. In fact every single one of the 256 character codes means something. Nearly all are so-called 'printing' characters—`PRINT CHR$(...)` will make something appear on screen. However, a few are non-printing—they produce some special effect. We have already met a couple of those: `CHR$(125)` to clear the screen and `CHR$(253)` to 'ring the bell'.

But what if you aren't happy with that huge choice of shapes? It may well be that you want to design something which cannot make use of those two hundred-plus designs. In that case, Atari lets you design your *own* characters!

I had better say right away, that you'd best stick to the ones that Atari gives you in the normal way. Getting your own characters is highly complex, and it's too advanced to describe in detail here. All I'll do, therefore, is to leave you with Program 16, which at least lets you dip your toe into this stormy water.

This program uses another assignment structure of Atari BASIC that we haven't yet met, the statements `READ` and `DATA`. `DATA` lets a program hold various data items in memory (line 10 of the next Program) for `READ` to pull out when required (line 40). Well, I'll come back to `READ...DATA` in due course. And the program also uses `PEEK`, which digs out the contents of a particular memory site, in this case site number 742.

Don't worry about all those details, but enter and run this program if you like. What it does is to replace the 'space' symbol with a dot.

Program 16: Going dotted

```
10 DATA 0,0,0,24,24,0,0,0
20 LET A=256*(PEEK(742)-4)
30 FOR B=A TO (A+7)
40 READ C
50 POKE B,C
60 NEXT B
70 POKE 756,INT(A/256)
```

Okay? Well, I said the system is complex. As you see every space on the screen has become a dot, but all the other characters have disappeared because with the Atari, you can't define one single character but must deal with a whole set of them. None of the others have been redefined, so they do not appear here.

Truth is, Atari BASIC can't really handle such complex things as this. We need to descend down towards machine code to be able to cope, and that is beyond the scope of this book. The same applies to the handling of what we call 'sprites', the Atari's well-known player/missile graphics.

To help you get over your disappointment, let me try a little bit of animation—movement of objects on screen. Program 17 will get you going.

Program 17: Fly-past

```
10 POKE 755,0
20 FOR GO=0 TO 8
30 FOR SITE=0 TO 30
40 SETCOLOR 2,8,14:SETCOLOR 1,0,60:PRINT CHR$(125)
50 POSITION SITE,GO+2:PRINT ">";CHR$(253)
60 NEXT SITE
70 NEXT GO
```

The concept of animation is the only new thing in that program. And it's line 50 that does that. Look at it with care. Each time through the SITE loop, the string '>' is printed one place to the right of its previous one. The space that starts that string wipes out the previous 'bird' and a new bird appears in the new place.

I admit that the bird flies rather slowly—it is in fact the 'cheep' sound that slows the loop down just a bit too much. When we've studied how to program sound with the Atari micros (see Chapter 7), you'll be able to come back and improve this if you wish.

Take a look now at the first program in Appendix 2 (at the end of the book). Flit-out is a more sophisticated version of Fly-past but not so much harder that you won't be able to see what's going on. When you've tried those two programs, you'll be able to do things of your own with much more meaning.

MODUS OPERANDI

Although this chapter started with a look at non-zero modes in the Atari, I haven't mentioned them for several pages. However most of what I've said of late applies to the posher text Modes 1 and 2. Let's now look at those two modes in more detail.

If you check back on Table 4.2 (Page 44) you'll see that I call these two modes the 'big text' and 'large text' modes respectively. And then if you look in the fourth column of that same table, you'll see why—Mode 1 offers 20×20 screen sites while Mode 2 has 10×20 . In each case there is a 'text window' at the foot of the screen, four lines of the usual Mode 0 forty screen sites. Program 18 collates various ideas about Modes 1 and 2 that we've met in earlier ones.

Program 18: One, two and away!

```
10 DIM CONT$(1)
20 FOR MODE=1 TO 2:GRAPHICS MODE:PRINT "Mode ";MODE
30 PRINT #6;"MODE ";MODE
40 PRINT #6;"mode ";MODE
```

```

50 PRINT #6;"MODE ";MODE
60 PRINT #6;"mode ";MODE
70 PRINT #6;"XXXXX";MODE
71 REM ** CTRL blocks used there:MODE .
80 PRINT #6;"XXXXX";MODE
81 REM ** CTRL blocks are MODE .
90 FOR W=1 TO 1000:NEXT W
100 PRINT #6;CHR$(253);CHR$(125);CHR$(35);CHR$(135)
110 FOR W=1 TO 1000:NEXT W
120 PRINT CHR$(253);CHR$(125);CHR$(35);CHR$(135)
130 PRINT "Ready to go on";
140 INPUT CONT$:NEXT MODE
150 PRINT "That's all!"

```

What does this program show us? It shows that PRINT can access the screen through either of two 'channels'. The normal PRINT statements, as in line 120, go to the text window at the foot of the screen. To print in the main screen area we need to send the instructions through Channel 6. This is done by using PRINT#6;..., as in most of the lines of this program.

Printing through Channel 6 in Modes 1 and 2 differs in various ways from normal printing (Mode 0). The most obvious is that the characters are larger in Mode 1 than before, and larger still in Mode 2. In fact Mode 1 text characters are the same height as before but twice as wide, while those in Mode 2 are twice as high *and* twice as wide. Figures 4.10 and 4.11 will give you the printing grids for Modes 1 and 2 (compare with Figure 4.4).

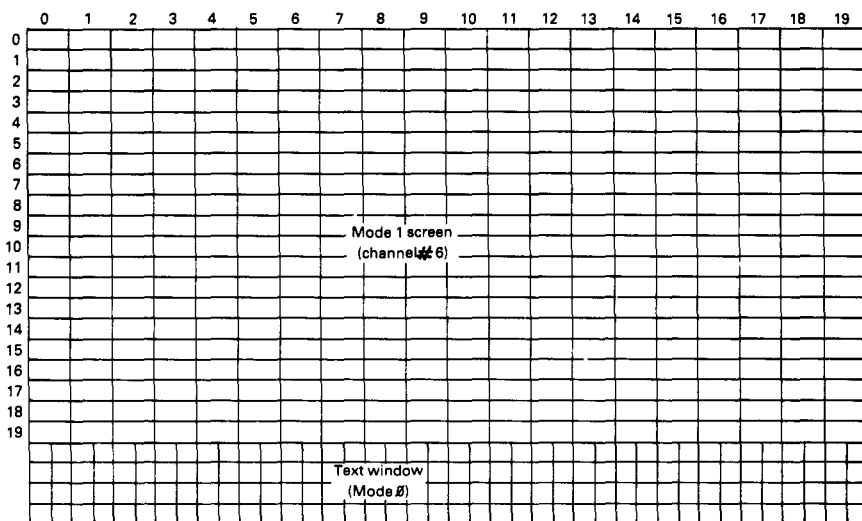


Figure 4.10 Mode 1 screen layout.

The numbers round the edges of the two figures describe the columns and rows for Channel 6 printing using POSITION—POSITION in Modes 1 and 2 works on the main screen and not on the text window. Now that you have a smaller number of screen sites to play with, you must take even more care not to try to POSITION outside the edges.

As I've noted before, you can omit the Modes 1 and 2 text window if you wish by using Modes 17 and 18 instead. (Adding 16 to the normal Mode numbers gives the effect.) Figures 4.12 and 4.13 shows the grids concerned. POSITION now works over the whole display area, and *all* PRINT statements must lead through channel 6. *Any* attempt to use Mode 0 printing will cause the computer to escape from a full-screen large-text Mode. That means you mustn't use any PRINT statements without #6, nor must there be any error in your program as ERROR reports appear in Mode 0. Try this, for instance—enter as a direct command GR.17 (to give Mode 1 without text window) and press RETURN. You will fail to enter Mode 17 this way—the READY message that the computer must give has to be in Mode 0.

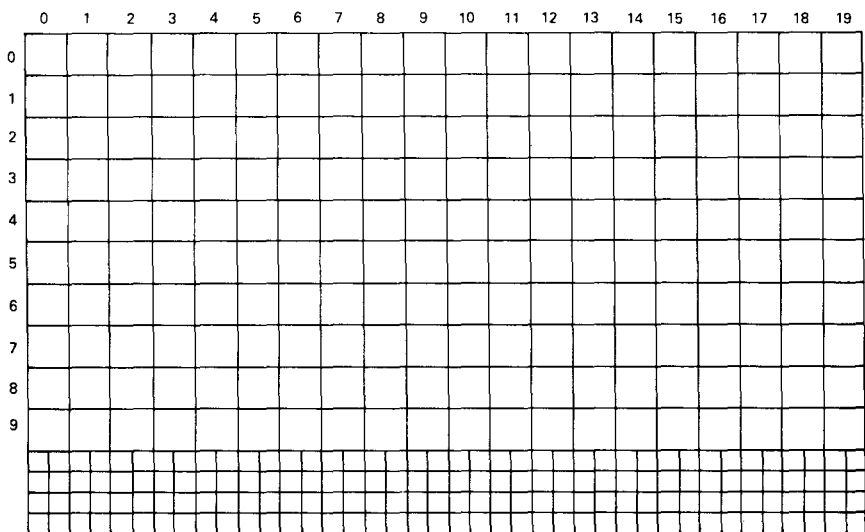


Figure 4.11 Mode 2 screen layout.

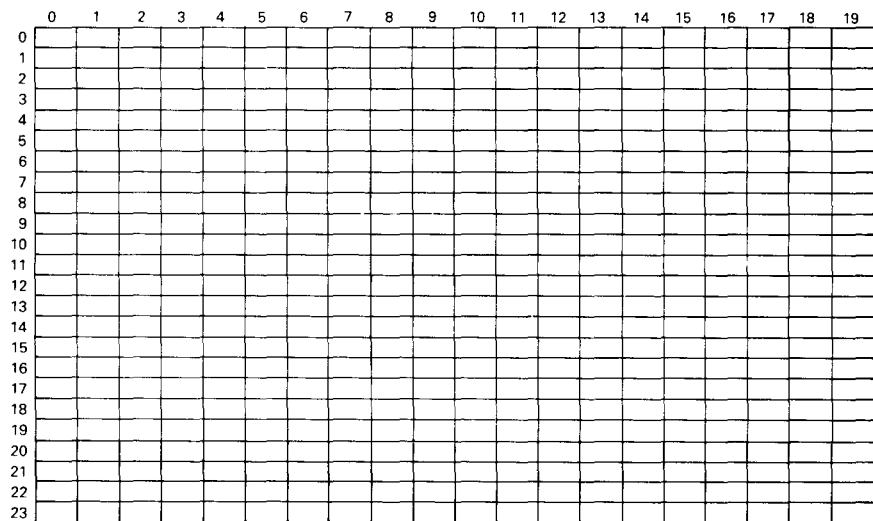


Figure 4.12 Mode 17 screen layout.

Maybe I should say a little about that strange symbol that we use for channel numbers in Modes 1 and 2. The symbol $\#$ is called 'hash' (and is got using SHIFT and 3). The symbol is often used in North America to mean 'number'.

One final note on POSITION in Modes 1 and 2 before we leave that subject. It is that there is no attempt by the micro to keep you away from the left hand edge of the screen this time. You will recall that in Mode 0, printing of all kinds normally starts two sites in from the edge. This does not happen in the other Modes.

Colour control in Modes 1 and 2 differs from that in Mode 0. Surprise, surprise! May I remind you about the concept of 'registers' that I mentioned before? Look at Table 4.3 (Page 47), which showed the use of the five Atari colour registers in Mode 0. Table 4.5 now does the same for Modes 1 and 2; you'll be pleased to know that both Modes have the same system!

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0																				
1																				
2																				
3																				
4																				
5																				
6																				
7																				
8																				
9																				
10																				
11																				

Figure 4.13 Mode 18 screen layout.

Table 4.5 Modes 1 and 2 colour registers

Register	Usage	Start-up value
0	“upper case”	orange
1	“lower case and window text”	pale green
2	“inverse upper case and window paper”	dark blue
3	“inverse lower case”	red
4	screen and border	black

SETCOLOR again is able to affect the contents of these colour registers. Recall that the first of the three numbers after the SETCOLOR keyword is the register concerned. Thus SETCOLOR 4,0,14 will give you a white screen (and white border) with the text window sitting sadly near the bottom. SE.2,0,14 will now bring the text window white as well. I put the usage of registers 0–3 in that table in speech marks to relate them to the effects got with lines 30–60 of Program 17. In Channel 6 in Modes 1 and 2, the contents of string constants (in other words, anything in between speech marks) appear in capitals in a colour that depends on the character style. Capital letters come out in orange, lower case letters in green, inverse capital letters are blue, and inverse small letters are red. Those four colours are the start-up values of the colour registers 0–3, and of course you can use SETCOLOR in the usual way to change the effects.

So you can’t get lower case letters on screen in capitals Modes 1 and 2. Well you can, but the technique is too advanced for us now. The same applies to the graphics blocks we met in Mode 0 using CONTROL. Lines 70 and 80 of the above program used normal and inverse graphics blocks, but they didn’t appear on screen as you would have expected—other characters, not related to those used, came up instead.

And using CHR\$(...) is fraught with problems too. Compare the effects of lines 100 and 200 in Program 18, for instance. You’ll find that CHR\$(253) gives us a buzz in Mode

0, but not through Channel 6. Enter this line 1000, for instance, and run it with GO.1000 in Mode 1 or 2.

```
1000 FOR A=0 TO 255: PRINT#6; CHR$(A); "□": FOR B=1 TO 100:  
NEXT B: NEXT A
```

This shows you that there are no lower case or graphics characters in the Mode 1/2 character set. To get them you would have to redefine existing ones as mentioned earlier. However, do peep at Question 5 in the last section of this chapter. . . .

The best way to get on top of large-scale printing in Modes 1 and 2 is to try some of the programs earlier in this chapter in those Modes. I'll make that part of the 'questions' section at the end of this chapter.

SUMMARY

The keywords used or extended in this chapter are (with their short forms):

DATA (D.)	POKE
FOR (F.)...TO...STEP...NEXT	POSITION (POS.)
GOSUB (GOS.)...RETURN	PRINT (PR. or ?)
GRAPHICS (GR.)	READ...DATA
LEN	RETURN (RET.)
NEXT (N.)	SETCOLOR (SE.)
OR	STEP
PEEK	TO

We've glimpsed the many modes other than Mode 0, but in the main looked at getting stuff nicely on screen in Modes 0–2. The 'stuff' has included some inverse material and graphics blocks, and I've perhaps whetted your appetite about ESC and CONTROL and about the hidden depths of DIY character design.

Apart from all that, by far the most important matter discussed is looping with FOR...TO...STEP...NEXT. Uses of this range from getting delays and mug-trapping to quite complex nestings.

A dozen programs came up *en route* to help (I hope). If you write a dozen more using these ideas, I'll be well pleased and so, I trust, will you. The next section should give you some ideas.

DO IT YOURSELF

1. Try at least a few of the programs from this chapter in Modes 1 and 2, and, where appropriate, without text windows in Modes 17 and 18.
2. Devise a program to print on screen a picture, map or design. Explore fully the good and bad things about Modes 0–2, and get used to POSITION, SETCOLOR and the graphics blocks. Try to keep as low as you can the number of keystrokes.
3. Extend Program 17 (Fly past) as far as you wish. How about random motion (as in Flit-out) and a nice landscape below (graphics blocks)?
4. Try variations on Program 12 (Tom Tiddler's ground). Star-fields perhaps?
5. Try POKE 756,204 at the start of a GR.0 program to print out the whole of the Atari character set. Similarly try POKE 756,206 in GR.1 or 2, then the same with 226. (Not all these POKES will work with older Ataris.) *Someone* has redefined character sets for you. Practise what you learn in text mode programs of your own. Did you find the £ sign?!

5 World Record

Doesn't time pass quickly when you're happy? I just noticed that we've worked through and made up loads of programs for your Atari, but haven't had all the details about how to save them for wet evenings. This subject then is the main topic of the chapter—saving your precious programs on cassette for use in the future.

I've mentioned the need before—if a micro could do everything it would need many millions of programs (sets of stored instructions). Even the biggest computer in the world hasn't got enough memory to store all the programs it could use. That failing's even more obvious with a micro. Hence the concept of backing store—a system which can put programs into a form able to be stored away from the machine for use as and when needed.

As with most modern micros, the Ataris have as their basic back-up system magnetic tape on normal audio cassettes. (No need to spend money on costly 'data cassettes' by the way.) For rich folk planning really big things with their micro, there is also what's called a disc system. Discs have a number of advantages, and are super to use—but they *are* costly and are best left till later.

Unlike most modern micros, on the other hand, the Atari cannot make use of a normal audio cassette recorder for its back-up system. (Well, it *could*, but you'd need to be a magic master of the soldering iron to be able to set it up.) It's a pity, but you'll have to spend more than a little sum on a special smart Atari tape recorder. There it is in Figure 5.1. I said it was smart. Looks good around the room. (Indeed it looks a bit like a disc drive, so you can impress your friends even more with it.)

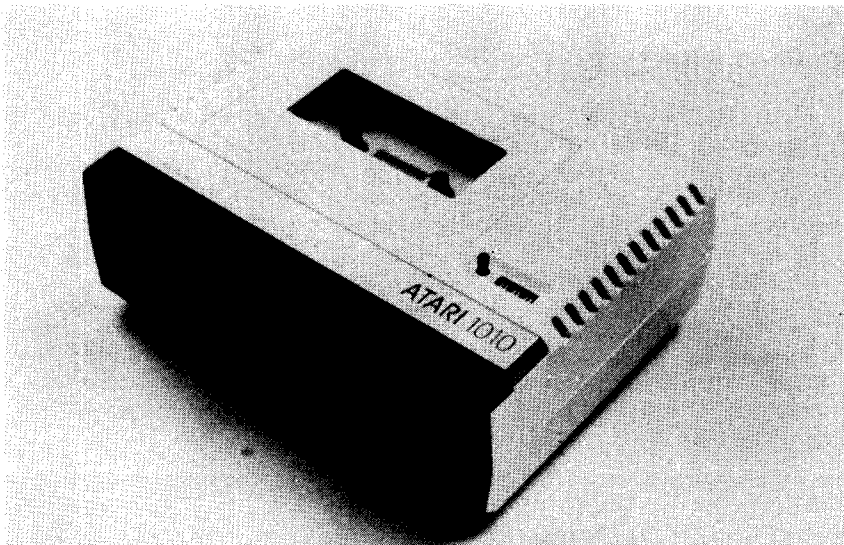


Figure 5.1 Atari tape recorder.

THE ATARI CASSETTE RECORDER

To set this up brings you one step nearer having to buy a mains adaptor as well—the recorder has its own posh power plug. And a thick cable on the end of that, and a second thick cable to join it to the micro. That's now five cables in use with your computer. Organization needed. Fear not—I'll come to that later in this chapter.

Join the recorder to the computer using the thick cable with a 'D-plug' at each end. It doesn't matter which end you use to make each connection. One end goes into the socket marked PERIPHERAL at the back of the computer, and the other goes into one of the two marked I/O CONNECTORS at the back of the tape recorder. (The other I/O CONNECTOR at the back of the tape recorder is for use if you later wish to add some other 'peripheral' device to your micro system.) Look after the cassette recorder:

Don't put cups of coffee on it.

If you do, don't knock them over.

If you do, Heaven help you.

Clean the heads occasionally, say every month or so (oftener if you use the system more than a few times a week). Use a special head-cleaner cassette. Or, if you are feeling truly loving, a cotton bud dipped in meths smoothed over the exposed parts when the empty machine's in PLAY mode. Don't spill the meths.

Keep the lid closed except when you put a cassette in or take one out.

Treat the control keys with respect. They don't seem very robust to me (but I admit I don't *know* they'll break).

Switch off the supply when the device is not in use.

Don't let the leads become too kinky or tangled.

In use the data recorder (for that's what they call it) has two tasks. Firstly you need it to record ('save') on cassette your masterpieces, or the programs you like from this book and elsewhere, for the future. Secondly use it to reload the data back into the micro's main store; this process is 'loading'. Spend thirty minutes or so practising with a short unimportant program entered on the keyboard. Then if anything goes wrong—as it may while you're learning—there's no harm done.

SAVING YOUR GEM

During any saving process, the micro sends a stream of data to the recorder. That stream is a coded copy of the program data in the main store. If the system is set up in the right way, the recorder will lay down on the surface of the tape a pattern of magnetization that represents that coded data. Then, as long as you look after your cassettes with care, that copy will be there forever. You can copy it back as often as you wish.

For good or ill the Atari system has three different commands and methods for saving program data. The simplest is CSAVE—the simplest, and as it happens, the least slow—but also the least useful. All you need to do is to type CSAVE (R) and the saving process will start. (CSAVE stands for 'cassette save', of course.)

The second method uses SAVE "C:name", where again the C stands for 'cassette' and where 'name' is the (optional) name you give your program. Saving this way takes longer than with CSAVE—so it's more of a bind—but on the other hand you now have the valuable option of having the program's name in the tape record. Soon enough you'll have dozens or hundreds of programs on cassette. Naming them like this will save you a lot of trouble. So, to save this way, type SAVE "C:name" (R) and off you go.

The third is a special version of the LIST command we've met before. The structure is LIST "C:name" (R)—easy too, but special purpose, so I'll come back to it in a while.

I suggest you use SAVE "C:name" (R) for getting copies of programs you want to keep in your software library. CSAVE is best only for making temporary copies of a program you're working on, while you get a cup of coffee or snatch a few hours' sleep. You can use a 'rubbish' tape for this purpose.

So—step by step—here's how to save a program in your Atari's memory to cassette tape for the greater good of mankind in future.

1. Connect the recorder to the micro; connect it to the power supply and switch on. The lamp on the front should glow to tell you that the device has got electrons coursing through its veins.
2. Press the EJECT (EJ) key to open the lid, and put in a cassette, tape side towards you. Close the lid by hand.
3. If the cassette tape is not at its start, press the REWIND key to put it there. The spools will turn until the cassette is ready. Press the STOP key. Press the counter button to reset the counter to 000.
4. Design a simple but meaningful name for the program. With a rare flash of genius I'll call it TEST here. Type SAVE "C:TEST", check this on screen, then press the RETURN key. The speaker will buzz twice and tell you to set up the recorder. So do it.
5. Press the RECORD and PLAY keys down together on the recorder. They should stay down (this takes a bit of strength—and a bit of practice if you are not used to audio tape recorders). Tell the micro you've done all that by pressing RETURN.
6. The micro tells the recorder to start working, and the spools will turn. The speaker gives out a dreary high-pitched whine, then a few bursts of sci-fi garble (that's the data) then more whine. The screen says READY, to show you the process is over, and the tape stops moving. Clever, that.
7. Make a second copy—repeat steps 4–6. Just possibly something was wrong. Note the counter setting.

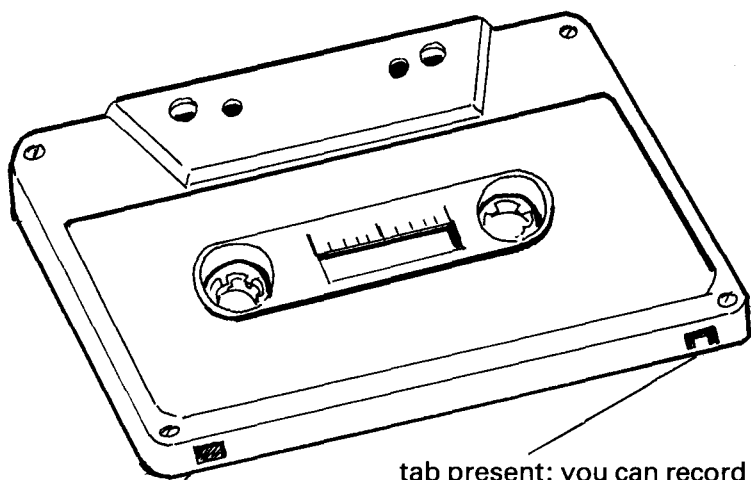
Except for the actual command used for saving, all the above applies for CSAVE, SAVE "C:" and LIST "C:". But as I said, possibly something was wrong. What could the problems be?

1. Recorder lamp not on and keys not working—no power to recorder. So check.
2. Recorder lamp not on, but keys working—lamp failure. Make a note to get it replaced. The lamp is useful.
3. Counter sticks on a setting and clicks without turning. Press the reset button more firmly.
4. Spools and counter don't move as they should. Check that the lid is fully down and replace the cassette if the problem remains.
5. RECORD key won't go down: The record-protect tab is missing from the cassette. Cover the hole with a piece of sticky tape. (See Figure 5.2.)
6. ERROR 133—the micro can't find the cassette recorder. Check the link between the two.
7. ERROR 138—the micro isn't happy with the recorder status. Check the link and that RECORD and PLAY are both down.
8. ERROR 139—the micro isn't happy with the recorder action. Check that the lid is fully down.
9. ERROR 163—the micro found something wrong but doesn't know what. Start from scratch again.
10. ERROR 165—the micro doesn't accept your clever name for the program. Program names must not be longer than about 120 characters. I guess you won't find this ERROR!

That's a giant list of problems! Saving programs is nearly always quite trouble-free! That's more than I can say for most micros.'

All the same you'll want to check that your save has met with success. Well—tough—you can't. There's no way with the Ataris to carry out what folk call *verifying*—letting the machine check that the copy on cassette is exactly the same as that in store. Saving is trouble-free, but it is just possible there's a fault in the coating on the tape. 'Drop-out' the jargon-ready users call it.

That's why I said you should make two copies. Indeed if it's really crucial that you have a separate record you should make two saves on each of two new cassettes. But that's only for masterpieces that took you many hours of work. Don't be put off, I say again.



tab present: you can record on top side

tab missing: you can't record on under side

The only way to verify that a tape copy is OK is to load it into a second micro while leaving the original program in the first one. Not many of us can afford that. So—take a deep breath, recall my motto ('don't be put off'), switch the Atari off and on again, and try to load the cassette copy back. Well I did *say* practise all this with a little unimportant program. . . .

LOAD OFF YOUR MIND

Loading means putting a copy of a program recorded on cassette into the micro's main store. We have three styles of saving, so guess how many loading systems there are! Wrong: Four in fact.

CLOAD (= load from cassette) is for use with a program put on tape with **CSAVE** (fast, simple, nameless).

LOAD "C:name" goes with **SAVE** "C:name".

RUN "C:name" does as well, but after the load is finished, the micro will start the program off without you typing **RUN**.

ENTER "C:name" is the other half of **LIST** "C:name". I'll return to it later.

I advised you to save with **SAVE** "C:name", so you'll need to load with **LOAD** "C:name" or **RUN** "C:name" as you wish. Both commands (as does **CLOAD**) include a **NEW**, wiping out the contents of the main store first. Here are the steps (whatever loading system you use):

1. Follow steps 1 and 2 in the saving procedure. Make sure that the cassette is the right way up (the same side should be on top as was on top before).
2. Follow step 3 above.
3. Type **LOAD** "C:TEST", check on screen, then press **RETURN**. This time the speaker will buzz once only.
4. Push the **PLAY** key down and press **RETURN** to tell the micro you've done so.
5. The Atari follows step 6 above and says **READY** when it's done.
6. Check that the program is in and correct with **LIST**, and then **RUN**.

There's no reason to expect it won't have worked, but I'll go through the fault-finding bit again.

Faults 1–4, and 6–8 are as before (except don't press the RECORD key down in 7) on top of those 8 we have:

9. All seemed well, but the tape runs through both copies without READY at the end: the micro hasn't found a program with a name you told it to load. Check that 'name' in LOAD "C:name" or just use LOAD "C:".
10. The same: you had RECORD pressed as well as PLAY. Tough—you've wiped out the record you wanted.
11. READY after the first set of sci-fi garbled sounds, but LIST gives nothing: you typed SAVE rather than LOAD (a common mistake!). Try again.
12. ERROR 2, the program being loaded needs more computer memory than your computer can offer it. (That won't happen with our TEST but it could happen with cassettes you buy, borrow or steal.) If you are rich, get more memory. If not, shrug.
13. ERROR 19: same applies.
14. ERROR 21: the program wasn't saved with SAVE "C:name". Try CLOAD or ENTER "C:".
15. ERROR 143: the micro found a fault in the cassette record. The magnetic surface may be damaged or the plastic under it could be stretched, or the 'recording' started on the clear leader part of the tape.

Sounds bad again, doesn't it? But fear not. These lists of possible LOAD/SAVE problems are really for reference—I bet you don't ever meet many of them. On the other hand of course, you may meet others (as when trying to run an Atari 600XL program on an Atari 400 micro). But I've done my best. . . .

MERGENCY

The SAVE/LOAD systems using LIST "C:" and ENTER "C:" differ from the others in being able to be used with parts of programs as well as whole ones. The main value of this is that it lets you 'merge' two programs into one.

What do you know about the standard LIST command?

1. LIST (R) displays on screen in number order the lines of a program.
2. LIST n shows the line n alone (if it exists).
3. LIST 0,n gives a display of the program from the start-up line to line n if it exists.
4. LIST n,3E4 gives the statements from line n (if it exists) to the end (unless you happen to have some with higher numbers than 30 000, in which case use LIST n,32767).
5. LIST n1,n2 gives the program from line n1 to line n2 if they exist.

Now, that standard LIST instruction sends the output to the screen. Its full structure is really LIST "S:" such and such, where "S:" sends the data specified to the screen. In the same kind of way LIST "C:" sends the data specified to the cassette recorder, with a name after the colon if you like. (And of value too is LIST "P:" such and such, to get a paper copy from a printer, if attached.)

The value of saving with LIST "C:" then is that you can select which section of a program to put on tape this way. You may find, for instance, that you've devised a really super reward for success in a game. Then you can save the game in the normal way (in other words with SAVE "C:name") and save the reward alone, for later use with other programs, using LIST. . . .

By that method you will be able to build up a collection of program sections (we call them subroutines). To use this subroutine library you need to be able to load back the right ones into the store, merging them by name into a new program you are writing. That's when ENTER... comes in. Unlike the other LOAD commands, this does *not* first clear the memory (despite what the *Atari Manual* says). All you need to do is to make

sure that none of the main program line numbers are the same as those of the subroutine—otherwise the new ones will overlap and wipe out the old.

You will therefore need to keep good records of subroutine line numbers in your paper files ('documentation'). Like this:

<i>Name</i>	<i>Line numbers</i>	<i>Action</i>
SR1	5100–5199	Slow screen clear with trills
SR7	5700–5799	Random colour-sound reward sequence
SR82	14200–14299	INPUT test—whole numbers from 1 to X only

And so on.

I'll say more about documentation later, but with a library like this you'll be able to build up complex programs with complex effects very quickly. All you need to do to add subroutines to the main program in store is to find the right cassette, use the tape counter to get the right place (with REWIND and ADVANCE on the recorder), and type commands like ENTER "C:SR42" (R). Perhaps Figure 5.3 will make this clear. Of course you will need to find out how to get the main program to 'call' each subroutine when required. Chapter 12 will tell you about that, but we have met the system in Program 8; it uses GOSUB and RETURN.

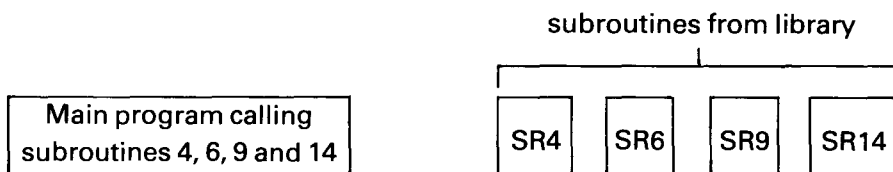


Figure 5.3 Building a program.

YOUR TAPE LIBRARY

Whether you are using your Atari at home, at work, or in a busy and complex classroom, you will need to keep important programs on audio cassettes (or discs). I would suggest most sincerely that right from the start you get into the habit of storing and documenting these cassettes in such a way that will keep you, and other people involved, completely satisfied even when your library contains several hundred cassettes. Several hundred? Wow! That's not at all unlikely if you really get into computing!

Major programs for the Atari 400 and 600XL may be stored on C10 or C12 cassettes; you will likely need C15 or C20 cassettes for big programs for the micros with larger memories. At first, when you are devising only very short programs, it is wise to have no more than about four on one cassette, and as I have said, it is best to save each one twice. This is because Atari programs on cassette are fairly long, and it is time-consuming to find the right one out of a number on a tape, even if you use the tape counter properly. Apart from anything else, there is nothing more satisfying than having a whole shelf full of neatly arrayed and numbered cassettes!

From the start then, number your cassettes, including a code for type if you wish (G for games, H for household, and so on). At the same time, record the main details of each program, with the same number, in an exercise book, card index, or loose-leaf file. Here are the details you need to include in this *file documentation*. The items marked with two stars are, in my view, quite essential.

- ** Cassette number and counter reading for each recorded copy of each version.
- ** Program name (*exactly* the same as that recorded on the cassette, with special attention paid to upper and lower case letters and special symbols if used).
- ** Author name(s).
- ** Date (of each version number if applicable).
- ** Computer, memory, any special requirements such as disc, or printer.

- ** Source of program.
- ** Objective(s) of program—what it tries to do.
- * List of variables.
- * Types of input if not clear from the messages on screen or printer.
- * Unacceptable inputs and other limitations.
- * Line by line listing of the program.
- * Ideas for possible future development.

Typical data cards of this nature are shown in Figure 5.4. Should you record listings typed in from magazines and books and/or purchase cassettes of commercially available Atari programs, document and store these in the same way with any user material. The same applies to cartridges.

If you *have* to record a large number of programs on one side of a tape, make the first one an index. Program 19a is suitable for this purpose.

TABLES	M17
AVC Software May 1984 £3.00 Atari 600 XL + printer	
Tables testing game; four levels	
6 (level 1) to 10 (level 4) + Remedial	
Tables 2-12	

ATARIVADER	G5 100-150 X2
Abdul Smith Sep 83 (version 4) Atari 800XL + joystick	
Version of Space Invaders with 18 levels of play 1/2 players	
(Developed from listing in Daily Mirror)	
(Veronica has version with printout of league tables)	

Figure 5.4 Typical program record cards.

Program 19a: A very simple index

```

10 SETCOLOR 1,5,0:SETCOLOR 2,5,14:SETCOLOR 4,5,14:PRINT CHR$(125);CHR$(125)
11 REM ** Set up; clear screen; buzz
20 PRINT:PRINT "*****"
21 REM ** Even an index can be pretty
30 PRINT:PRINT " This is Cassette 47. Side 2...."

```

```

40 POSITION 14,9:PRINT "I N D E X"
50 POSITION 9,12:PRINT "Bank accounts"
60 POSITION 9,14:PRINT "Bank statements"
70 POSITION 9,16:PRINT "Money invaders"
71 REM ** etc
80 POSITION 2,21:PRINT "*****"
90 FOR W=0 TO 1 STEP 0:NEXT W

```

Let me repeat the comments made in one of the REMs in that listing—even index programs should be nicely laid out! See how I have tried to follow that rule with this program.

If you are using disc rather than cassette, you have the possibility of very fast access to a large number of programs. (Each 'floppy disc' can carry up to 64 separate programs.) As we can use the various loading (and saving) commands within a program as statements, we have the possibility of extending Program 19a as in 19b. This gives a neatly automatic loading of named programs from disc.

Program 19b: Simple index with automatic loading

```

90 REM ** Delete this from 19a
100 DIM PROG$(10):PRINT :PRINT "Type name of program needed";:INPUT PROG$
110 SETCOLOR 1,5,14:SETCOLOR 2,5,0:SETCOLOR 4,5,0:PRINT CHR$(125);CHR$(253)
120 POSITION 5,12:PRINT "SEARCHING for: ";PROG$;""
130 LOAD PROG$

```

This process cannot be used with cassettes, and in any event would not be worthwhile as it would be incredibly slow. However, there is another approach that I would like to bring to your notice that is very helpful if you have a large number of small programs to deal with.

If you wish to make full use of your Atari's fairly large memory to store a number of fairly short programs all together, you may adapt the index to let the user select the individual one wanted as in Program 20. This approach is more professional (and causes less uncertainty) than that which expects him or her to type GO TO 1000, GO TO 2000, or whatever.

Program 20: Simple index with user-selection

```

10 DIM CH$(1):SETCOLOR 1,9,14:SETCOLOR 2,9,4:SETCOLOR 4,9,0:PRINT CHR$(253);CHR$(125)
20 PRINT :PRINT "<=><=><=><=><=><=><=><=><=>"
30 PRINT :PRINT "This is Cassette 7, Side 1"
40 PRINT :PRINT "1. Donald Duck"
50 PRINT "2. Mickey Mouse"
60 PRINT "3. PLATO"
70 REM ** etc
120 POSITION 2,18:PRINT "<=><=><=><=><=><=><=><=><=>"
130 POSITION 2,20:PRINT "Please press number (& RETURN) ";:INPUT CH$:IF LEN(CH$)<>1 OR CH$<"1" OR CH$<"8" THEN GO TO 130
131 REM ** Mugtraps there
140 PRINT CHR$(125):GOSUB VAL(CH$)*1000
150 RUN
1800 REM ** Program 1 starts here
1990 RETURN
2000 REM ** Program 2 starts here
2990 RETURN
3000 REM ** etc

```

An interesting feature appears in this program in lines 130–140. The former includes 'mug-traps' (programmed devices designed to prevent a user-error from passing on)—it accepts a string only if it contains a whole number between 1 and 8 inclusive. Then the next line, 140, converts the input string into a pure number using VAL. That number, multiplied by 1000, forms the address to which the program is sent with GOSUB. (We call this a 'computed address'.)

LOOKING AFTER YOUR EQUIPMENT

While I surely do not recommend it, dropping your Atari or otherwise treating it roughly should not harm it or you. Unless you drop it on your toes. Rough treatment, may, however, affect the plugs and/or the leads—there are lots of them!—and perhaps cause an internal break or poor contact. See the next section for tips on a permanent arrangement which should reduce this major problem.

Because of charge effects, TV (and monitor) screens become dusty very quickly. This reduces the picture contrast and sparkle and it is made worse with fingerprints. I suggest you avoid getting into the habit of touching the screen. (A particular hazard arises from touching the screen and the computer at the same time—much easier to do than you might think. This can transmit high voltage electric charge from the screen to the micro and damage some chips.)

Clean the screen each week with a cloth dampened in warm, slightly soapy water. Take the opportunity to give a quick wipe to the computer bodywork at the same time. Do not, however, let the water enter the gaps in the keyboard or the socket. (Hot coffee is even worse if spilled on the keyboard!)

Other than this, neither the computer nor the TV set will require any special care: both should give you good service for years.

The cassette recorder and the cassettes do need a little more looking after, however. This equipment has two main enemies—dust and magnetic fields. I recommend that you take note of the following and make sure your friends do the same.

1. Do not leave the cassette recorder cover open.
2. At regular intervals clean the head and demagnetize the system as I described on Page 63 (use a special head-cleaning cassette or a cotton bud soaked in methylated spirits). This can be done when you clean the rest of the equipment.
3. Keep each cassette in its (correct) box when it is not in use—certainly do not leave it in the cassette recorder.
4. Never put cassettes on top of the TV set or close behind it—video equipment contains very strong magnetic fields.
5. Do not leave cassettes in a hot place, such as on a window sill or a shelf over a radiator.
6. Once a year run through each cassette on fast forward and fast rewind.

If your cassettes are going to be used by a number of people, it's a good idea not to let the originals ('masters') into circulation. Instead make 'back-up' copies for daily use and keep the masters in a safe place. You can make a copy by loading from the master into the micro and saving on the new cassette. Alternatively use a good audio cassette duplication system.

These precautions are widely reckoned not to break copyright; however it is *quite* wrong to pass copies of purchased software outside your home/department, even if you don't charge for it.

STAKING YOUR CLAIM

Using a computer can become a marvellous hobby, totally engrossing, totally detrimental to the use of legs and to your social life (that's a serious warning really). But as with all hobbies, it needs a bit of effort to fix things up right. You can't hog the home TV set forever. You can't work for long on a bean-bag. So once you're sure you're going to enjoy computing (if you're not sure yet), *organize*. You'll need these items.

1. A table (at least 2 m × 1 m) and a comfortable upright chair in a quiet corner of a quiet room, with power nearby and storage space at hand. Alternatively, if you want to be really swish, why not look at some of the specially designed computer furniture you can get now? I'm very happy to use, for instance, a combined workdesk/trolley with shelf and TV stand from Selmor Engineering of London E1.

2. A TV set just for the purpose. A black and white portable is OK, but consider a guaranteed second-hand colour set from a local shop. Good ones are available for around £50, and you can secretly watch the midnight movie while you are thinking about your precious program.
3. A permanent arrangement of everything on and around the table, allowing you to work quickly and for a long time without strain. I've sketched a suitable layout for your Atari corner (assuming you are right-handed) in Figure 5.5. Make sure it's appealing to visitors when you proudly show off your work—but not too appealing for stray infants poking around in your absence. Or parents. Or burglars.

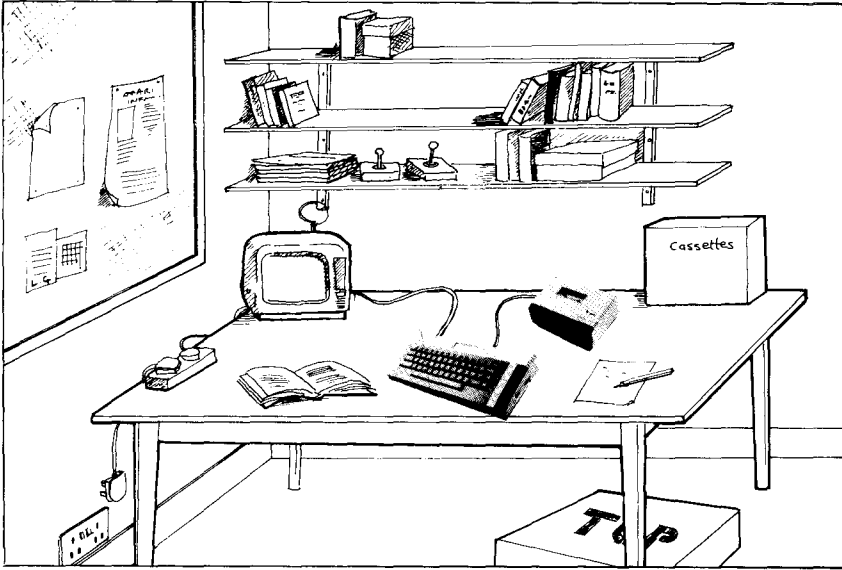


Figure 5.5 Atari corner.

SUMMARY

In this chapter I've concentrated on how to get your precious programs from the micro onto cassette for storage and how to get them back again. We've also looked at organizing yourself, your work area, and your software library.

The keywords used in the chapter are as follows (with the smallest abbreviation, if any, in brackets):

CLOAD	CSAVE (CS.)
ENTER (E.)	LOAD (LO.)
RUN (with cassette program)	SAVE (S.)
VAL	

I couldn't think of anywhere else to put this advice. If you have any load/save problems, try the command CPRINT (R) before your load and save commands.

DO IT YOURSELF

Of course, a chapter like this cannot end with specific programming exercises. All the same it includes some important material about organization. So I suggest you get yourself together, in your corner, with your nice Atari computer set up, and a smart software library.

6 The Master Plan

This book is *not* a programming text, as you'll know if you are unusual enough to have read the Introduction. After all, the last chapter was about much more than programming itself, it dealt with the value of organization in general. In this chapter I shall get a little closer to programming and discuss the planning of programs.

Now I'm *not* going to spend a lot of ink on telling you what good programming is (even if I were immodest enough to claim to know). When you've finished the book, and perhaps started to think seriously of programming for work, real fun or profit, then you will have some idea of what makes a good program, but it will be up to *you* to find out all the details.

All the same, there are good habits and bad habits. And if you get into good habits early on, you'll be glad later. And then your work, fun or profit-making programming will be less work, more fun, and—who knows?—more profitable.

PLANNING

The essence of good habits in programming is *planning*. I think I must go further and say that good programmers plan their planning. The worst thing to do when you get a program idea—trivial or world-shattering—is to sit down at the keyboard and start typing. If you do that, **YOU WILL GET INTO AN AWFUL MESS**. And even if, by some miracle, the result seems to work first time, it will not be very efficient; it may have **BUGS** in it (dreaded hidden errors); and it will not impress the folk who are to look at it and use it.

Efficiency in programming means having a product which works correctly in all circumstances, works at the right speed, uses as little memory as possible, and looks good to everyone involved. So your planning must aim for these criteria. It will still be a miracle if the thing is perfect first time you try, but getting it close to perfection will be a lot simpler.

The essence of program planning is an approach called *top-down development*. This means breaking that initial idea down into smaller and smaller chunks until at last each one is an easy-to-handle unit. Actually we use the words *module* or *routine* for those final units.

It's a bit like having a bar of chocolate at a party. You've got to break it up (down?) into the right number of pieces of the right size before it can be processed (eaten!). Here the processing I'm thinking of is producing the idea in the form of a program the computer can run, *coding* it as we say.

Coding is simple when you have the idea in fragments in front of you. Each fragment—concept or module—can then transfer to a chunk of the program. We call such a chunk a routine, or a subroutine, or a procedure. The sketch in Figure 6.1 shows what top-down programming entails. Do you see what I mean? The original broad concept breaks down into a lot of small chunks.

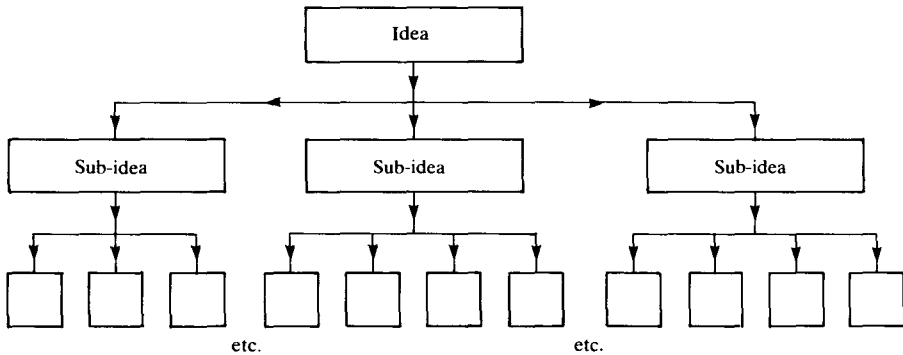


Figure 6.1

Of course, at the moment, the programs you are writing are pretty short ones. I wouldn't dream of saying that you've got to work out a detailed set of modules for them. Even now, however, it is surely important to view your program not as a whole but as a number of separate parts. Each part has its own aim, each part is programmed (coded) separately, and each part is tested separately. Then the whole thing made of those parts added together is likely to be neat, efficient, *and* bug-free. If you get into the habit of thinking like that now, by the time your masterpieces are lengthy (novels rather than short stories), your work will be based on good habits and involve far fewer tears.

I think I've already mentioned that *user-friendliness* is a hobby horse of mine. Here again I must say something about it. Yes, user-friendliness is a crucial part of program planning. As well as working correctly and efficiently, the program must also appeal to the user in other ways. I try to plan from the outset to make my work user-friendly—so that that person, whoever he or she may be, is always at ease and never wonders what to do. The most important thing here concerns planning the screen layout and the messages on it. The messages should be short but clear, well laid out and not cramped—and spelled and punctuated properly!

Get used to using large and small letters in your messages, as well as highlighting methods such as inverse, special symbols, underlines, and framing tricks. Text on screen is *dreary* if all it consists of is capital letters.

You may be sure that I shall come back in much more detail on all this later in the book. In particular, Chapter 11 deals with the matter in much more depth.

7 Full of Sound and Fury

As you can guess from my title, the main topic we'll study in this chapter is making your Atari sound forth. The 'fury' I also mention implies the feeling you'll get when you come up against some of Atari's strange quirks of sound control. Indeed I should say that sound with these micros is pretty complex. As it would take a book half as long as this one to cover it fully, you can guess that I'm not going to go very far!

Early Ataris have their own speakers built in—small, but quite good ones. Those also, and the more recent ones only, can send sound output direct to the TV speaker. This is a very good idea, met with strangely few micros—TV speakers are better than the weedy ones you can get in a micro case and volume control is of course easy and good. (On the other hand, most video monitors don't offer sound, so if you use a monitor with your machine, you've got a problem—or blessed silence.) To help you get on top of sound coding, let me first set out a little background science.

THEORY

Think of a single pure note produced by vibration in some musical instrument or other source. A guitar will do fine. We can describe the note in terms of five factors.

1. *Loudness* (volume)—is it strong or faint? What we call loudness relates to what scientists term amplitude. The vibrations are more violent in the case of a loud sound (high amplitude) than in that of a soft one (low amplitude).
2. *Pitch*—is it high or low? Sounds from the left-hand keys of a piano (or a man's throat) are lower pitched than those from the right (or a female voice). Pitch relates to what scientists call the frequency of vibration: to obtain a low pitch there are fewer vibrations in a second—the frequency is low.
3. *Quality* (or timbre)—is the sound pure or rough?—the difference between the sounds of say a trumpet and a guitar playing the same note is a difference in quality. (What's really happening in those cases is that there's a different set of tones, of different amplitudes and frequencies, in each sound.)
4. *Attack*—how does the sound start, build up, continue, fade away, and stop? You may have heard tapes played backwards of the sound of a piano playing—the result does not sound like a piano at all because the attack has changed.
5. *Duration*—how long does the sound last?

ATARI SOUND

As I said, the sound facility of the Atari micros is very complex. Almost uniquely among home computers, they allow the user to program all but the fourth of the above factors of

a sound. (Most micros, if they have sound at all, allow control only of pitch (frequency) and duration.)

Partly this is because the Atari has four-channel sound, allowing four separately programmed notes to be 'played' together—giving chords as well as good control over quality.

To control all these factors from instant to instant gives the Atari the potential of a fairly feeble synthesizer. No, it is *not* a synthesizer, so you have to do a lot of work to get anything like real music. Don't be ambitious, at first anyway, stick to simple beeps and warbles, simple sequences of notes and sounds.

The keyword we use for coding the speaker output is SOUND. Follow the keyword with four numeric values separated by commas:

SOUND C,P,T,A

Here

C is Channel: 0, 1, 2, or 3

P is Pitch: 255 (low) to 0 (high)

T is Timbre (or something like it!): allowed values are 0–15

A is Amplitude: 0 (peaceful) to 15 (loud)

I use the symbols C, P, T, A in an attempt to make it a little bit less hard to recall what each numerical value does. The mnemonic I use is 'chapter' for C,P,T,A.

If you check back to the list I have of the theoretical sound factors, you'll see that I haven't mentioned either 'attack' or 'duration'. With your Atari micro there's nothing you can do to work on attack—unless you're an extremely clever programmer, anyway. Duration, on the other hand—the time for which a given note lasts—is not programmed as part of the SOUND statement (which is a pity in many ways). As soon as you tell the micro to send a note to the speaker using SOUND, it will do so—and the note will go on and on and on forever. You have to stop it, in other words. The only ways you can stop the speaker sounding after SOUND are:

- (a) by sending a new SOUND command to the channel concerned;
- (b) by sending SOUND C,0,0,0 (to switch the sound channel off without putting a new sound in);
- (c) using END (often the last statement in a program);
- (d) by using RUN or (this being rather more drastic!) NEW;
- (e) by pressing the (SYSTEM) RESET button.

Clearly only the first couple can be used within programs. Type in and run Program 21. I designed it to get you moving with SOUND.

Program 21: Sound as a bell?

```
10 PRINT CHR$(253);CHR$(125):REM ** Sound the old way!
20 PRINT :PRINT "C H A N N E L S"
30 PRINT :PRINT "Channel 0"
40 SOUND 0,121,10,8
50 FOR W=1 TO 25:NEXT W
60 PRINT :PRINT "Channel 1"
70 SOUND 1,91,10,8
80 FOR W=1 TO 25:NEXT W
90 PRINT :PRINT "Channel 2"
100 SOUND 2,72,10,8
110 FOR W=1 TO 25:NEXT W
120 PRINT :PRINT "Channel 3"
130 SOUND 3,60,10,8
140 FOR W=1 TO 1000:NEXT W
150 SOUND 2,0,0,0:SOUND 3,0,0,0
160 PRINT :PRINT "P I T C H E S":PRINT
170 FOR P=0 TO 255
180 PRINT P;" ";
190 SOUND 0,P,10,15:FOR W=1 TO 25:NEXT W
```



```

200 SOUND 1,255-P,10,15:FOR W=1 TO 25:NEXT W
210 NEXT P
220 FOR W=1 TO 1000:NEXT W
230 PRINT :PRINT :PRINT "T I N B R E"
240 FOR T=0 TO 15 STEP 4
250 SOUND 0,100,T,8:FOR W=1 TO 25:NEXT W:PRINT T
260 SOUND 1,100,T+1,8:FOR W=1 TO 25:NEXT W:PRINT T+1
270 SOUND 2,100,T+2,8:FOR W=1 TO 25:NEXT W:PRINT T+2
280 SOUND 3,100,T+3,8:FOR W=1 TO 25:NEXT W:PRINT T+3
290 NEXT T
300 FOR W=1 TO 1000:NEXT W
310 PRINT :PRINT "A N P L I T U D E"
320 FOR A=0 TO 15
330 PRINT A
340 SOUND 0,100,10,A:FOR W=1 TO 25:NEXT W
350 NEXT A
360 FOR W=1 TO 1000:NEXT W

```

The best thing you can do now is to explore SOUND as much as you can. Use Program 21, and the comments that follow, as a basis for this work. Use plenty of direct SOUND commands, writing points down as you go. Switch off the speaker at any stage if the noise gets on your nerves, using END (R)—this is better than (SYSTEM) RESET, as you don't lose the screen contents. Think of the neighbours—keep the TV volume control low.

CHANNEL

The SOUND keyword takes four numbers after it—SOUND C,P,T,A, as I've said—and the first of these tells the micro which of the four separate sound channels you wish to use. Lines 20–130 of Program 21 build up a four-note chord, with each SOUND statement sending details of a pure note along one sound channel to the TV speaker. Atari advise that when using more than one channel you shouldn't let the sum of the loudness values exceed 32. (The fourth number in SOUND controls loudness.) The reason for this is the strange speaker vibrations that could appear. All the same, I've had no problem going to the limit (which is 60), so I guess you can try it if you need.

Anyway, there's not much more I can say about channel control—oh, except that in a program (when you can't use END unless it's truly the end of the program), switch a channel off with SOUND C,0,0,0. This instruction is the only way to turn a *single* channel off outside a program as well as in one of course—END turns off *all* sound. Here's the quickest way to turn off three channels (or four likewise):

```
FOR C = 1 TO 3: SOUND C,0,0,0: NEXT C (turns off 1–3; leaves 0)
```

PITCH

The second number after SOUND has the job of telling the micro which pitch (frequency) to send down the channel; I call it P. The value of P can range from 0 to 255 in whole number steps. There are two problems with it.

Firstly, how people perceive sound is very complex; indeed much remains to be learned. Secondly, how the cuddly Atari deals with pitch is very complex as well.

In lines 170–210 of Program 21, I set up a loop which uses two channels to trip lightly up and down the P range. One up, one down. At the same time. If you listen to that section

of the program you'll hear more than a rising and a falling scale. Sometimes the notes combine to produce a third one. It's not the micro's fault, nor the fault of the speaker—these so-called secondary tones are from the realms of physics and physiology. Try this:

SOUND 1,10,10,10

Then add

SOUND 2,11,10,10

Now you get, not two notes as you may expect, but three—and shifts in what you hear as you move your head. This is all very interesting, I think, but I must leave *you* to explore it.

I said that Atari pitch control is complex. In what ways? The first problem is that high values of P give low notes, and vice versa. That makes it harder to deal with any kind of music as you have to sort of compose upside-down. The second problem is that a given number of steps between Atari P values does not give a constant pitch 'interval'. Thus, for instance, you can't double (or halve) P to go up an octave. A pity. You can see both points in Figure 7.1. (I hope you have enough idea of how to read music to be able to handle this.)

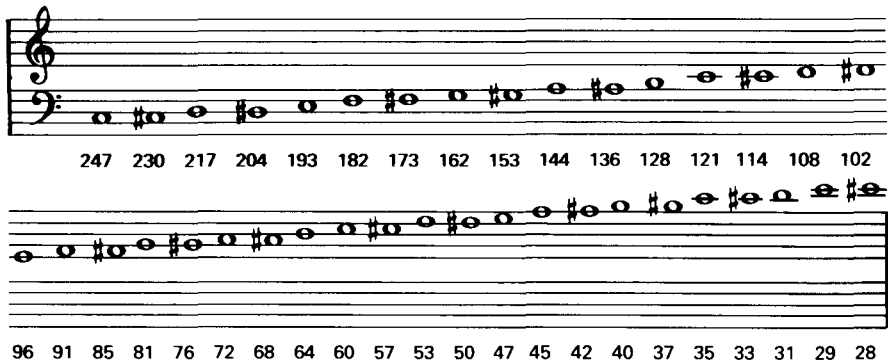


Figure 7.1 Scaling the Atari mountain.

Look at that figure and I hope you see what I mean—P goes down as the note goes higher, and the steps aren't even. If you listen again to the 'pitch' section of Program 21, you'll hear my point—jumps between high notes are very clear, while the low ones glide gently into each other. As I say, these points mean that you are going to have quite a lot of work to do when writing music for your Atari!

TIMBRE

Well—this is more complex still. . . . I'll try to set out what the values of T in SOUND C,P,T,A does.

- (a) T can take any value from 0 to 15. Only whole numbers officially, but sometimes decimals seem to make a difference.
- (b) Use either 10 or 14 to give a pure tone. I prefer 10—it's not so hard to remember, especially as I tend to use the same number for my standard value of A.
- (c) Even (not odd) values of T introduce some level of 'distortion' to the pure tone. That's a gross over-simplification. Here's a table which tries to cover the effects;

Table 7.1

T	Effect	Interference
0	Mix of 4 and 8	Sometimes
2	Same as 6	Sometimes
4	Pulsing pitched white noise	Sometimes
6	Pulsing pitched grey noise	Sometimes
8	Pitched white noise	No
10	Pure tone	No
12	Buzzes	Very common
14	Pure tone	No

I use some code words and phrases in that table! ‘White noise’ truly means a mix of all frequencies (itches) at equal levels. It’s rather like the sound you make called Shhh! The so-called Shhhs here relate to some extent to the value of P, pitch, so I call them pitched white noises. The P-effect is even more marked when T = 6, hence my term ‘grey noise’.

The third column of this table gives some indication of the chance of getting some of those strange perception effects I mentioned before. Sometimes a single distorted note will sound like two, or even three, separate notes (secondary tones appear, in other words). In other cases, no sound at all comes out. Try SOUND 1,9,4,15 for instance—not a squeak!

- (d) Odd values of T switch the channel off if it’s on. But if the channel is off already, you get a click. That can be useful.

To come to any kind of grip with Atari sound distortion—the T-factor—you’ll have to explore the thing yourself. Lines 230–300 in Program 21 aren’t really much help now. Try Program 22 instead. Enter it. RUN, BREAK to stop at any point of interest, play with direct SOUND commands, then use CONT. to get back to the program where you left off.

Program 22: Time for T

```

10 FOR P=0 TO 255
20 FOR T=0 TO 15
30 SOUND 1,P,T,10
40 PRINT "Pitch ";P,"Timbre ";T
50 FOR W=1 TO 50:NEXT W
60 NEXT T
70 NEXT P

```

AMPLITUDE

The control of sound amplitude (loudness or volume) in your Atari, is, by a strange design fault, very easy. The last number after the SOUND keyword does the work now, A I called it. A can have any value from 0–15, and the larger the value, the louder the sound. Simple, n’est-ce pas? The last lines of Program 21 say it all!

INTO PRACTICE

Let’s try some programs using SOUND ideas. They’ll help you gain some more feel for this complex but important aspect of Atari coding.

For a start we'll add a bit of sonic sparkle to Tom Tiddler's ground (Program 12)—and at the same time POKE some colour into his cheeks. (As I'll explain in Appendix 4, there are times when POKE can do better than SETCOLOR.) Program 23 is the result.

Program 23: Tiddler on the roof

```
20 PRINT CHR$(125)
30 FOR A=1 TO 100
35 POKE 710,RND(0)*255:POKE 712,RND(0)*255
36 REM ** Random screen/border colour
40 POSITION RND(0)*39,RND(0)*23
41 REM ** Random site
45 SOUND 1,RND(0)*255,10,10:FOR W=1 TO 25:NEXT W
46 REM ** Random beep
50 POKE 756,204:PRINT CHR$(0);
51 REM ** Pound sign at last! Xls only....
60 NEXT A
65 SOUND 0,247,10,10:SOUND 1,121,10,10:SOUND 2,60,10,10:SOUND 3,29,10,10
70 POKE 755,1:POSITION 0,0
80 FOR W=0 TO 1 STEP 0:NEXT W
```

The next program takes these ideas a bit further (and also uses GET, which I'll come back to later). Try pressing keys. . . .

Program 24: Key in the door

```
10 PRINT CHR$(125):OPEN #1,4,0,"K":REM ** 'Opens' the keyboard for direct input
20 FOR GO=0 TO 1 STEP 0:REM ** Repeat for ever
30 GET #1,K:REM ** Makes K take the code of the key pressed
40 SOUND RND(0)*3,K,10,10:REM ** Plays a note of pitch K through random channel
50 POKE 710,K:POKE 712,255-K:REM ** See last program
60 POSITION RND(0)*39,RND(0)*23:PRINT CHR$(K)
70 NEXT GO
```

Audio-visual effects always make nice wallpaper. So the next program is another one of the same kind of thing. It's also here to give you a taste of drawing lines in graphics mode. I've put comments in REMs, but when you run this pleasant little program, you'll make your own, I expect—I bet colour control in graphics modes ain't easy.' You'd be right.

Program 25: Concrete cat's cradle

```
10 GRAPHICS 11
11 REM ** Use Mode 7 if no 11
20 FOR GO=1 TO 2 STEP 0
21 REM ** Slight change for a change
30 SOUND RND(0)*3,RND(0)*255,RND(0)*15,RND(0)*15
40 COLOR RND(0)*4
41 REM ** Graphics color setter
50 PLOT RND(0)*70,RND(0)*190
51 REM ** Chooses random point on Graphics screen
60 DRAWTO RND(0)*70,RND(0)*190
61 REM ** Draws to random point on Graphics screen
70 FOR W=1 TO 25:NEXT W
80 NEXT GO
```

The name, by the way, is meant to describe this smashing combo of concrete music and the cat's cradle design of lines. Well, I had to call it something. I keep on using the random number thing, you'll notice—RND. RND(0), recall, gives a random number (truly a semi-random one) from 0 to 0.999999999. Multiply it by a number X and you get a 'random' number between 0 and (almost) X. Where you need a random *whole* number (integer) use INT (RND(0) * X). We don't in fact need INTs in the cases I'm using, so I've left them out.

The next program is the obvious development of Number 24—a sort of Atari synthesizer. Look, Atari sound is quite fun, but no way are you going to be able to get

decent music out of it without very great hard work. If computer music's your scene, then this program (and the next) will give you a start, but I tell you now, you go it alone from then on.

In Cynth, Program 26, the centre row of letter keys acts as a simple eight-note 'organ'. A is C, S is D, and so on (if you see what I mean), up to the K key which gives C again, an octave higher. If you want a rest, musically speaking, press the space bar. Line 35, by the way is optional—it gives you a piano rather than an organ. Roughly, anyway.

Program 26: Cynth

```

10 OPEN #1,4,0,"K"
20 FOR 60=0 TO 1 STEP 0
30 GET #1,K
35 REM : FOR C=0 TO 3: SOUND C,0,0,0:NEXT C
40 IF K=65 THEN SOUND 0,121,10,0
50 IF K=83 THEN SOUND 0,100,10,0
60 IF K=68 THEN SOUND 1,96,10,0
70 IF K=70 THEN SOUND 1,91,10,0
80 IF K=71 THEN SOUND 2,81,10,0
90 IF K=72 THEN SOUND 2,72,10,0
100 IF K=74 THEN SOUND 3,64,10,0
110 IF K=75 THEN SOUND 3,60,10,0
120 IF K=32 THEN FOR C=0 TO 3: SOUND C,0,0,0:NEXT C
130 NEXT 60

```

Although Program 26, as it is—or as extended by you—is very good for letting you mess about at the keyboard, it doesn't allow you to code tunes that you can record on tape for the computer to play later. Even in a simple game program, after all, you may wish to pop in the odd trill or fanfare, perhaps as a reward for beating the high score of sixty thousand.

For this you need to write a series of SOUND statements. Something like that in the next masterpiece (where again, begging your pardon, I use subroutines to make life easier).

Program 27: Call the RSPCA

```

10 LET Q=1000:LET C=1100
11 REM ** Quaver & crotchet routine addresses
20 FOR A=1 TO 2
30 SOUND 1,96,10,10:GOSUB C
40 SOUND 1,100,10,10:GOSUB C
50 SOUND 1,121,10,10:GOSUB C
60 SOUND 1,0,0,0:GOSUB C
61 REM ** Rest for one crotchet
70 NEXT A
80 FOR B=1 TO 3
90 SOUND 1,81,10,10:GOSUB C
100 SOUND 1,60,10,10:GOSUB C:SOUND 1,0,0,0
101 REM ** How to separate repeated notes
110 SOUND 1,60,10,10:GOSUB C
120 SOUND 1,64,10,10:GOSUB Q
130 SOUND 1,72,10,10:GOSUB Q
140 SOUND 1,64,10,10:GOSUB Q
150 SOUND 1,60,10,10:GOSUB C
160 SOUND 1,81,10,10:GOSUB Q:SOUND 1,0,0,0
170 SOUND 1,81,10,10:GOSUB C
180 SOUND 1,0,0,0:GOSUB C
190 NEXT B
200 SOUND 1,91,10,10:GOSUB Q
210 SOUND 1,96,10,10:GOSUB C
220 SOUND 1,100,10,10:GOSUB C
230 SOUND 1,121,10,10:GOSUB C
990 END
1000 FOR W=1 TO 100:NEXT W
1001 REM ** Gives quaver delay
1010 RETURN

```

```

1100 FOR W=1 TO 200:NEXT W
1101 REM ** Gives crotchet delay
1110 RETURN

```

Writing programs to music like this is tedious, especially when you use more than one channel to get chords. Here's how to go about it in essence.

1. Choose a short, easy tune that you know!
2. Play it out on a piano and write down the order of notes by name. You should be able to remember the durations and rests.
3. Then program. . . . For each note enter SOUND C,P,10,A, where C, P and A stand for Channel (0–3), Pitch from Figure 7.1, and Amplitude (5–15, say). Follow with a subroutine call to give the time you want the note to play for: GOSUB C,Q, or whatever, as in Program 27. For each rest, use SOUND C,0,0,0 (or FOR C = 0 TO 3: SOUND C,0,0,0: NEXT C if you are into multi-channel work).
4. RUN to test—and then be prepared for many hours of polishing.

Your work in coding tunes as above is almost doubled by the fact that the Atari SOUND statement includes no time control. A note once SOUNDED plays on till stopped (by any of the methods I've listed before). Still, that *does* give us the means of getting screen action while notes play forth. Not all micros can cope so well with this double task! Program 28 shows what I mean. Try to catch it. . . .

Program 28: Falling star

```

10 SETCOLOR 1,0,14:SETCOLOR 2,0,0:SETCOLOR 4,0,0:POKE 755,1
20 PRINT CHR$(125)
30 FOR A=0 TO 22
40 POSITION 10+A,A:PRINT "*"
50 SOUND 1,20+A*5,10,A/2+1
60 FOR W=1 TO 10:NEXT W
70 POSITION 10+A,A:PRINT "."
80 NEXT A
90 FOR A=100 TO 250 STEP 5
100 POKE 710,A:POKE 712,A
110 SOUND 1,A,8,25-A/10:SOUND 2,250-A,4,25-A/10
120 FOR W=1 TO 5:NEXT W
130 NEXT A
140 FOR W=0 TO 1 STEP 0:NEXT W

```

That program includes two styles of sound effect (and the one you are more used to on vision, using POKE 710/2). First is the 'fall' of lines 30/50/60/100, and second is the 'explosion' given by 110 and 130–150. No doubt you'll be able to use versions of both for your own needs. I'll close with a few more effects you may wish to explore.

Rocket-launch

```

FOR A = 250 TO 100 STEP -1
SOUND 1,A,8,A/10-10
SOUND 2,225,4,A/10-10
FOR W = 1 TO 25: NEXT W
NEXT A

```

Car crash

```

FOR A = 0 TO 15 STEP 0.1
SOUND 1,100,8,A
FOR W = 1 TO 5: NEXT W
NEXT A
FOR A = 15 TO 0 STEP -0.2
SOUND 1,60,8,A
NEXT A

```

Siren

```
FOR A = 0 TO 1 STEP 0
FOR B = 200 TO 75 STEP -1
SOUND 1,B,10,10
NEXT B
FOR B = 75 TO 200
SOUND 1,B,10,10
NEXT B
NEXT A
```

SUMMARY

I've spent time in this chapter on the following keywords (listed here, as usual, with any useful short forms):

COLOR (C.)	OPEN (O.)
DRAWTO (DR.)	PLOT (PL.)
END	RND
GET	SOUND (SO.)

Of course I've spent most of my time on the use of SOUND to produce beautiful effects from the TV speaker. The SOUND statement takes four numbers after it, these being Channel (0–3), Pitch (255–0), Timbre (0–15) and Amplitude (0–15). In particular, I've put before you methods of using sound to produce random 'tunes', some kind of 'music', and special audio effects.

DO IT YOURSELF

The best way for you to explore the power of Atari sound is to try your own ideas on the basis of the ones you've met in this chapter. Programs 23, 26, 27, and 28 are most useful in this regard. In each case there is plenty for you to try in order to extend what I've said. With Cynth, for instance, you could add sharps and flats, explore the use of the keys at left and right of the keyboard to change octaves, and try making keys on the top row act as 'stops' to affect timbre perhaps. And then you can add pretty screen colour flash routines as has been done in a couple of programs in this chapter.

8 Just a Plot

Near the end of the last chapter I brought you face-to-face with COLOR and as you guessed, its use is complex. It is complex, so get stuck straight into Program 29, and then we'll take it from there. I'll also use the opportunity of going thoroughly into the RND and looping tricks used—so I'll separate the notes not concerned with graphics colour effects.

Program 29: Tour of Hanoi

```
10 GRAPHICS 3
20 FOR W=1 TO 50:NEXT W
30 SETCOLOR 4,4,10:COLOR 3:PLOT 1,0:DRAWTO 1,20:SOUND 1,50,10,10
40 FOR W=1 TO 400:NEXT W
50 PLOT 20,0:DRAWTO 20,20:SOUND 2,100,10,10
60 FOR W=1 TO 400:NEXT W
70 PLOT 30,0:DRAWTO 30,20:SOUND 3,150,10,10
80 FOR W=1 TO 400:NEXT W
90 COLOR 2:PLOT 17,0:DRAWTO 12,20
100 PLOT 18,0:DRAWTO 13,20
110 PLOT 22,0:DRAWTO 27,20
120 PLOT 23,0:DRAWTO 28,20
130 SOUND 0,200,10,10
140 FOR W=1 TO 600:NEXT W
150 FOR C=1 TO 3:SOUND C,0,0,0:NEXT C
160 FOR GO=1 TO 200
170 SETCOLOR 1,RND(0)*15,RND(0)*15
180 SOUND 0,(250-60)-RND(0)*(50),10,10
181 REM ** Sort this out, despite perils of upside-down SOUNDING! Very nice effect....
190 NEXT GO
200 FOR GO=100 TO 250
210 SETCOLOR 4,RND(0)*15,RND(0)*(15-60/17)
220 SOUND 0,60,0,25-60/10
230 SOUND 1,60+5,4,25-60/10
240 SOUND 2,60-5,0,25-60/10
250 SOUND 3,255,0,25-60/10
251 REM ** Nice sound effect here too
260 NEXT GO
270 FOR C=0 TO 4:SETCOLOR C,0,0:NEXT C
280 FOR W=0 TO 1 STEP 0:NEXT W
```

I use RND quite a lot there for special effects, so here's a thorough account.

RND(0) gives a number, roughly at random, between 0 and 0.999... The zero in the brackets has no meaning, but one of the quirks of Atari BASIC is that there *must* be a number in those brackets. Thus RND(57.839 * 17.1) has just the same effect. A zero is easiest! Test the effect of RND with this command:

FOR A = 1 TO 15: ? RND(0): N.A. (1)

(Recall ? is short for PRINT, and N. for NEXT.)

Then if you want, you can try different numbers in the brackets. Even A will work here!

Of course, each time you use a RND command like that, you'll get a different set of numbers. After all, the result is supposed to be random. (In fact, however, it isn't truly random, but it is rare that one needs to worry about that.) Most uses of RND are to produce a random whole number between zero, or one, and some limit. To display a random whole number between zero and twelve, for instance, use

```
PRINT INT(RND(0) * 12)
```

The function INT chops off the decimal fraction part of the number it works on. Try it:

```
FOR A = 1 TO 15: ? INT(RND(0) * 12): N.A. (2)
```

You get a series of numbers between zero and eleven inclusive. *Not* from 0–12, as the highest value possible is INT(0.999... * 12), which is eleven. This will do the trick, then:

```
FOR A = 1 TO 15: ? INT(RND(0) * 13): N.A. (3)
```

Or this, if you want a number from one to twelve inclusive:

```
FOR A = 1 TO 15: ? INT(RND(0) * 12) + 1: N.A. (4)
```

Here's a routine, on the same lines, that models fifteen throws of a die:

```
FOR A = 1 TO 15: ?.INT(RND(0) * 6) + 1: N.A. (5)
```

Now let's see what I've done in Program 29 with RND.

Lines 170 and 210: SE.C, RND(0) * 15, RND(0) * 15

The first number of a SETCOLOR instruction refers to the colour register (and I'll come back to that in the main text soon). It's the other two numbers that matter now. They set the colour and brightness in turn (see Page 46); each needs to take a whole number value between zero and fifteen inclusive. Really then I should have used INT(RND(0) * 16)—compare command (3) above. In fact you do *not* need to use INT here—if SETCOLOR finds a non-whole number, it rounds it off to the nearest whole number. So if (RND(0) * 15) gives the value 14.8, SETCOLOR will take the value to be 15.

Thus the use of RND within SETCOLOR statements has much more logic than usual (except for that silly need to put in (0)!).

Line 180: SO.0,(250 - GO) - RND(0) * 50,10,10

What I wanted in this GO loop (lines 160–190) is a series of random beeps gradually rising in pitch. In this case the first SOUND number and the last two are no problem. (Recall SOUND C,P,T,A in the last ChaPTA!) I'm using channel 0, and looking for pure fairly loud tones. So it's the second SOUND number we're concerned with now, the one dealing with pitch. Straight away I remind you of a major problem with Atari SOUND coding—the higher the pitch number the lower the pitch. (That's what I meant about upside down SOUNDing in line 181!)

First, then, the RND(0) * 50 bit. This gives a random number in the range 0–49.999... I take the value obtained away from (250 - GO), getting a number in the range 249 to 199.000 000 001 when GO = 1, and in the range 50 to 0.000 000 001 when GO gets to 200. SOUND, like SETCOLOR, turns a number with a decimal part into the nearest whole number, so I don't need an INT here.

Got it? Okay, it is complex, but this is the sort of effect you want to be able to put into your programs, isn't it? After that I guess it's not too hard for you to follow what's going on in lines 220–250 as GO in the second loop runs from 100–250. After all, these lines have no RNDs to cause extra brain-hurting. We get a nice explosion effect. Some of this mental gym work *does* hurt my brain. I like it! It took me about ten minutes to get line 180 right, for instance. Great fun!

The main part of Program 29 was to give you a chance to play with COLOR, PLOT, and DRAWTO in Mode 3. PLOT plots a point in a graphics mode, DRAWTO draws a line, and COLOR affects the colour of the point or the line. No problem? Well. . . .

First thing is that what you need to do in the different graphics modes (in other words, Modes 3 to 15 and their derivatives) depends on the mode. Secondly the way COLOR

and SETCOLOR relate is not straight forward. Well, did you expect it to be? 'The last shall be first', so I'll look at work in Mode 3 before going through the door to the upper modes.

MODE 3

Using PLOT (PL.) and DRAWTO (DR.) to mark points and draw lines in the graphics window (the main part of the screen) is dead easy. Fully clear your Atari by switching off then on, and try these commands in turn.

GR.3	Screen goes all black except for the blue text window, in which READY appears.
C.2	No effect seen.
PL.10,15	Yellow square appears ten character positions across and fifteen lines down.
DR.15,10	'Line' of yellow squares goes from there to 15,10.
DR.25,10	Horizontal yellow line.
DR.35,35	Yet another line, and ERROR 141 ('gone off limits').

Try your own. But then, you really know all the ideas here, don't you? Program 25, Concrete Cat's Cradle exposed them on Page 79.

To summarize—in Mode 3, you can use PLOT x, y to make a square appear at point x, y in the graphics window. The limiting values are 0, 0 (top left); 39, 0 (top right); 0, 19 (bottom left); 39, 19 (bottom right). DRAWTO x, y gives a 'straight line' of squares from the last point used to x, y . If you use DRAWTO *without* having visited any graphics point before, the starting point of the line is not fixed. Often it is 6, 3, but the actual site depends on what has happened.

Figure 8.1 shows the Mode 3 screen. It's very much the same as that of Mode 1 in practice, *except* that the bottom four lines only are for text now, the rest being the graphics window.

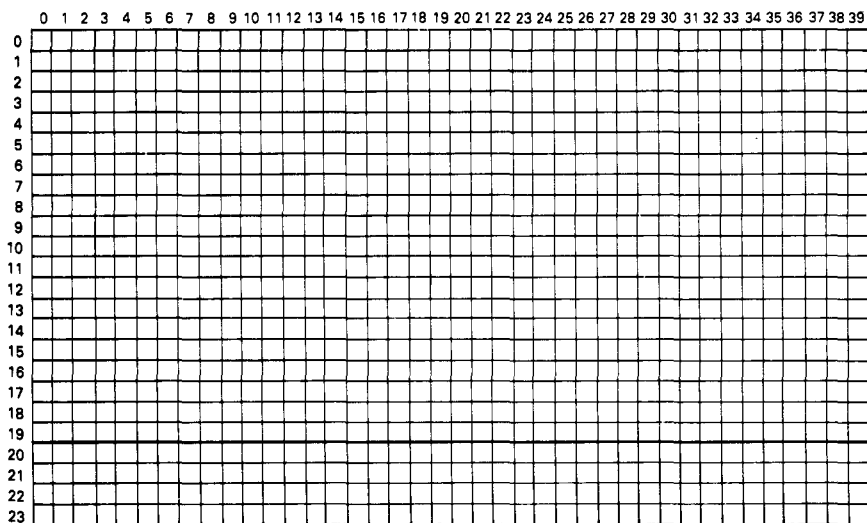


Figure 8.1 Mode 3 screen layout.

As a little aside, why don't you try the effect of PRINT# 6 in Mode 3? You recall that in Modes 1 and 2, this lets us put coloured text above the Mode 0 window. Ah, no text now—Mode 3 is pure graphics! Apart from the text window. Mode 19 (3 + 16) has no text window at all, of course.

Still, the use in GR.3 of such commands as PR.#6; “Shiva for eva!” does show one thing—that the squares that appear in the graphics window can have different colours. Still, you guessed that from Program 29 (whose PLOT and DRAWTO statements should now be clear).

What then of colour in Mode 3? Let’s start with the new version of the table I did showing the colour registers in the text modes. Table 8.1 deals with Mode 3’s colour registers, there are five as before.

Table 8.1: Mode 3 colour registers

Register	Usage	Start-up value
Ø	plotting	orange
1	plotting	yellow
2	plotting	blue (as text window)
3	not used	—
4	background	black

SETCOLOR, you recall, changes the values of the colour number held in these storage boxes. SE.R,C,B puts colour number C, brightness B, into register R.

At the moment we should have a black ‘paper’ for the graphics window. That’s handled by register 4, so mess around with SE.4... for a while. SE.4,9,4, for instance, will make all the screen the text window colour. (I like that effect—I’m not keen myself on a text window that differs in colour from the main area, though it does have some uses.)

It’s not just hat-makers who regret the fact that few people wear hats in these modern times. I do too. If I thought you’d be wearing a hat now, I could say ‘hold on to your hat’. That’s because it’s now time to bring the keyword COLOR on to the stage.

COLOR lets you choose which register controls the PLOT and DRAWTO squares that follow its use. As Table 8.1 says, you can plot and draw in any of three colours. Select from your palette with COLOR. Here’s the bit where you need to hold on to your hat. . . .

- To get the colour in Register Ø use COLOR 1.
- To get the colour in Register 1 use COLOR 2.
- To get the colour in Register 2 use COLOR 3.
- To get the colour in Register 4 use COLOR Ø. (Register 3’s not used.)

No comment. . . .

I’ll go through all that again. Say you have the urge to draw a red line from top left to bottom right of a purple Mode 3 graphics window. The steps are, right from switching on, as follows:

1. GR.3—to enter the right mode
2. SE.4,6,8—screen goes purple except for frame (still black) and text window (still blue—true)
3. SE.Ø,3,6—put a darkish red colour value into Register Ø
4. COL.1—say you’re going to use Register Ø
5. PL.Ø,Ø—plot the starting point of the line
6. DR.39,19—draw the rest of the staircase, I mean line

Bonuses:

7. SE.2,6,8—makes text window match rest of screen
8. SE.0,13,10—changes Register 3, and therefore the line, from red to yellow
9. SE.1,0,0—darkens the text in the window
10. PR. “End of practice”—adds a sort of title in the text window

Got it now? You can have up to four colours on the screen, and I include that of the text window. The values in the colour registers 0–2 and 4 control these—and *you* control those values with SETCOLOR Register, Colour, Brightness. Use COLOR 1, 2 or 3 in advance of any PLOT and DRAWTO work. The statement tells the micro that you wish to use the colour in Register 0, 1 or 2 respectively. Plot a point *x* columns along and *y* rows down with PLOT *x*, *y*. Draw a line between the last point visited and *x*, *y* using DRAWTO *X*, *Y*. Stay within the screen boundaries to avoid ERROR.

Some other points arise from what I’ve said in the last few paragraphs.

1. Once you have put a value into the colour register with SE. (or some other way), it stays there until you change it, or until you switch off the micro. Colour effects in Mode 3 therefore depend on what has been done before you enter the mode. In practice it is thus wise to set your colour register values each time you change modes.
2. I implied that you can PLOT and DRAWTO in background colour, telling the micro you want to do so with COLOR 0 (= access colour register 4). What’s the point of that, you ask! Well, by plotting in background colour you erase what was there before. Try this, for instance, as direct commands in GR.3:

```
C.1          (C. = COLOR)
PL.0,0
DR.39.19
C.0
DR.0,0
```

Okay, I’ve been lazy: you can do it better—but you get the idea. And my laziness also makes the point that DRAWTO does not give a line starting with the starting point, but starting *from* the starting point. See?

3. The lines drawn with DRAWTO in Mode 3 are hardly what one would call fine straight lines. The word staircase does in fact describe them very well. How do we get round that problem? The answer is to go to a different graphics mode, and thus increase what we call the ‘resolution’ of graphics work. *Resolution* gives a measure of how close points can be plotted and how fine the lines drawn are.

After you have played as much as you wish with the material in this section, we can go on to extend my last point.

MODES 5 AND 7

First the good news. Pretty well all I said in the last section about Mode 3 applies to Modes 5 and 7 as well. The bad news isn’t really bad—it follows from the fact that these modes offer better and better resolution, so you can plot smaller and smaller points and draw finer and finer lines. So the screen graphics area is no longer 20×40 .

What I need, therefore, are layout grids of the two modes. Here they are, in Figures 8.2 (Mode 5) and 8.3 (Mode 7). As before, add 16 to the Mode number to get a text-window-less version, with the bottom four lines replaced with, respectively, eight rows of 80 points and sixteen of 160.

The best way to see the changes you need to make is with Program 30. I have made the program up out of the ten steps used above to explore Mode 3. Put as a program you can go through as often as you like, and make what step-by-step changes you wish in any of Modes 3, 5, and 7. I’ve used GOSUB/RETURN again, you’ll note—clearly better than other methods.

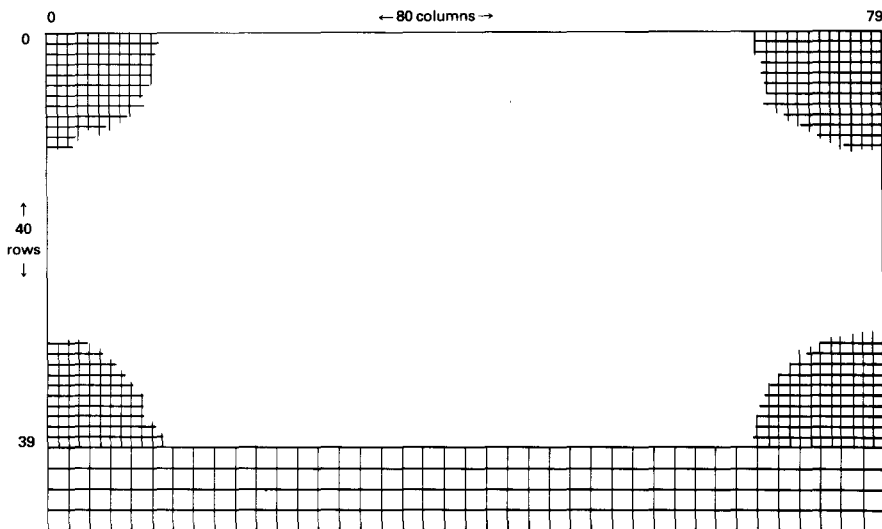


Figure 8.2 Mode 5 screen layout.

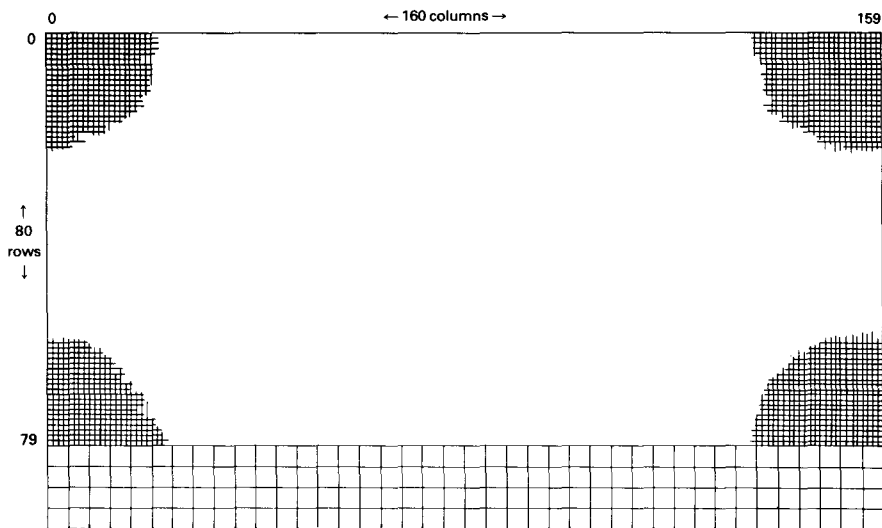


Figure 8.3 Mode 7 screen layout.

of Modes 3, 5, and 7. I've used GOSUB/RETURN again, you'll note—clearly better than other methods.

Program 30: Mode explore

```

10 DIM GOON$(1):LET GOON=1000
20 GRAPHICS 3
21 REM ** Or 5 or 7
30 SETCOLOR 4,6,8
31 REM ** Purple screen
40 LIST 30,31:GOSUB GOON
50 SETCOLOR 0,3,6
51 REM ** Put dark red in Reg.0
60 PRINT :LIST 50,51:GOSUB GOON
70 COLOR 1
71 REM ** Prepare Reg.0 for use
80 PRINT :LIST 70,71:GOSUB GOON

```

```

90 PLOT 0,0
91 REM ** Plot line's starting point
100 PRINT :LIST 90,91:GOSUB 600N
110 DRAWTO 39,19
111 REM ** Draw line
120 PRINT :LIST 110,111:GOSUB 600N
130 SETCOLOR 2,6,8
131 REM ** Match window to screen
140 PRINT :LIST 130,131:GOSUB 600N
150 SETCOLOR 0,13,10
151 REM ** Change Reg.0 colour
160 PRINT :LIST 150,151:GOSUB 600N
170 SETCOLOR 1,0,0
171 REM ** Darken text in window
180 PRINT :LIST 170,171:GOSUB 600N
190 PRINT :PRINT :PRINT , "    That's it!"
200 FOR W=0 TO 1 STEP 0:NEXT W
990 STOP
991 REM ** Protects sub-routine: good habit tho' not strictly needed in this ca
se!
999 REM ** Sub-routine here
1000 PRINT ,,"More";
1010 INPUT 600N$
1020 RETURN

```

Really, I don't think I need to say more, do I? *You* can explore as you will. That *was* an easy section. Will the other main graphics modes be as simple to pick up? Well, yes. . . .

MODES 4 AND 6

First the good news. There's nothing new to practise here. Mode 4 is set up like Mode 5 and 6 is like 7. And of course the corresponding higher siblings relate in the same ways. They differ from the ones explored in the last section by allowing only two colours on the screen at once rather than four.

Why use Mode 4 (or 6) rather than 5 (or 7) then? The reason is that, as the Table on Page 44 shows, doubling the colour power leads to a near doubling of the memory taken up by the micro for its own purposes. So we'll have to use the even modes when we're in a long program and need to do some plotting without pushing storage needs too high. You won't have to do that for a while unless you are using an Atari 400. Here are the colour registers' data for Modes 4 and 6.

Table 8.2: Modes 4 and 6 colour registers

Register	Usage	Start-up value
0	plotting	orange
1	text brightness	—
2	window colour	blue
3	not used	—
4	background	black

As before, use SETCOLOR to change the contents of a register, and COLOR 1 (to get Register 0) or COLOR 0 (to get Register 4) to select your plotting colour. (Recall we plot in background colour, COLOR 0, to unplot something.)

Really, then, Modes 4 and 6 are simpler than 5 and 7. Fair enough, they have lower numbers.

MODE 8

Well, after all that, we'll be able to take Mode 8 in our strides. (Though, note that if your Atari is a lot less than spanking new, you can't get Mode 8—not enough memory.)

As with the even number modes we've just looked at, 8 lets you have only two colours on screen. For a change though, the colours are those of (a) the border (Register 4 as you'll guess), and (b) the rest. 'The rest' includes the text window—so at last we have a mode in which text window and plotting patch *have* to be the same colour. Register 2 therefore handles the colour of all this. (We're used to Register 2 for the text window paper colour, after all.)

Table 8.3: Mode 8 colour registers

Register	Usage	Start-up value
0	not used	—
1	text and plotting brightness	—
2	screen colour	blue
3	not used	—
4	border	black

As before, SETCOLOR etc, etc. I'm sure you get the use of SETCOLOR by now. Alas, I can't say the same about COLOR—in Mode 8 there's something a bit perverse.

As usual, you dig out and dust off Register 4 for use with plotting in background colour. *But* this doesn't affect text printing as you might expect (which is good). A bigger *but* is that you use COLOR 1 and *not* COLOR 2 to get back to normal after using Register 4.

Perverse, OK, but no real problems there. All I need to say now about Mode 8 is its resolution. I'm not going to drive Shiva's artist potty with all the little squares, so Figure 8.4 gives the data more simply. I suggest you get hold of some graph paper for your rough work of plot plotting.

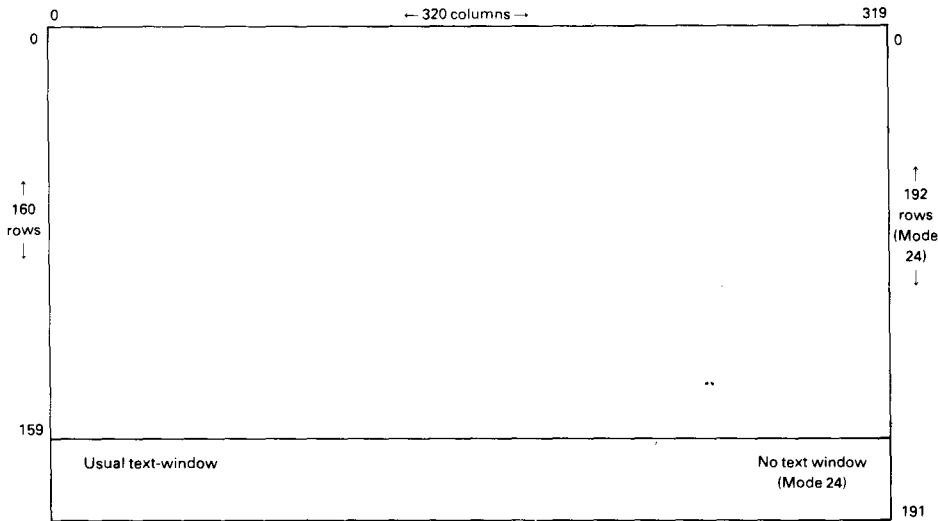


Figure 8.4 Mode 8 screen layout.

Mode 8, then, gives us the chance to plot tiny points, just big enough to see, and to draw fine lines with the smallest staircase effect. What you lose is colour, but colour is not *needed* in many Mode 8 uses. (There are, by the way, advanced methods of getting more colours in a Mode 8 display, but they are beyond the range of us in this book. Still, there'll be some more on graphics in Chapter 17 where I'll bring you the joys of Modes 9–15.) Here's a nice Mode 8 program to finish off with.

Program 31: Leonardo

```
10 GRAPHICS 24:SETCOLOR 1,0,0:SETCOLOR 2,10,10:SETCOLOR 4,10,4:COLOR 1:REM ** S
et up screen
20 OPEN #1,4,0,"K:":REM ** Prepare keyboard for RETURN-less inputs
30 LET X=159:LET Y=85:REM ** Start at screen centre
40 FOR GO=0 TO 1 STEP 0:REM ** Set up REPEAT/UNTIL loop
50 PLOT X,Y:SOUND 1,Y,10,10
60 GET #1,K:REM ** Look for key press
70 IF K=43 AND X>5 THEN LET X=X-1
80 IF K=42 AND X<314 THEN LET X=X+1
90 IF K=45 AND Y>5 THEN LET Y=Y-1
100 IF K=61 AND Y<186 THEN LET Y=Y+1
101 REM ** All above fixes next plot point
110 IF K=83 THEN LET GO=2
111 REM ** Detects S for STOP
120 NEXT GO
130 REM ** YOU can put what you like in here for the end routine: flashy sound
& screen maybe?
150 FOR W=1 TO 2 STEP 0:NEXT W
151 REM ** Keeps us in Mode 24
```

SUMMARY

In this chapter I've set out how to plot points (called pixels = picture cells by the *élite*) and to draw lines in the graphics Modes 3–8. Here are the keywords used, with their short forms:

- COLOR (C.)
- DRAWTO (DR.)
- GRAPHICS (GR.)
- PLOT (PL.)
- POSITION (POS.)
- SETCOLOR (SE.)

The point/line bit is not a problem as long as you recall the number of columns and rows in the Mode you are in. See the table on Page 44.

A little more tricky is colour control. We use five registers, the colour and brightness data in which we can change with SETCOLOR. Then you access the register you want with COLOR, in most cases get at Register X with COLOR X + 1, unless X is 4 in which case circle round back to COLOR 0.

Here's a table of the register functions in *all* the modes we've met. A dash means 'not used'.

Table 8.4

Modes	Registers				
	0	1	2	3	4
0/16	—	text ink	text paper	—	border
1/17	upper case text	lower case text	inverse upper case text	inverse lower case text	text screen

Table 8.4 contd.

Modes	Registers				
	0	1	2	3	4
2/18	upper case text	lower case text	inverse upper case text	inverse lower case text	text screen
3/19	plotting	plotting	plotting	—	plotting screen
4/20	plotting	—	—	—	plotting screen
5/21	plotting	plotting	plotting	—	plotting screen
6/22	plotting	—	—	—	plotting screen
7/23	plotting	plotting	plotting	—	plotting screen
8/24	—	(ink brightness)	screen	—	border

DO IT YOURSELF

By far the best exercise you can give yourself with all the material covered in this chapter is to write lots of BASIC programs to draw all kinds of pictures on screen. Start of course with simple ones, and build up by letting your ambition match your ability. You may prefer to avoid using a text window (though it's very nice for titles)—in that case make the last line of your program the first one you write, for instance, `1000 FOR W = 0 TO 1 STEP 0: NEXT W`. This will prevent your super design from jumping back to Mode 0 as soon as it stops.

Having done that as much as you wish, try to devise something of your own on the lines of Program 29, or extend Program 31 (Leonardo) into, say, Mode 7, with features that let the user change the colour of the 'ink' used and to add a title at the end.

9 Taking Command

In this chapter I'd like to discuss in some detail a rag bag of Atari features that should make your 'program development' quicker and more splendid. I'll first revise and extend the idea of commands.

COMMANDING HEIGHT

You'll recall, because you use them at your keyboard all the time, that commands are instructions for the micro that you enter without a line number—the micro obeys at once, but doesn't keep them in store for later use. We've used commands a lot in recent chapters, not just for the obvious things like getting a program into the micro off tape (LOAD...) or getting the machine to carry it out (RUN) but also to allow quick checks of what happens if. . . .

As far as I know, we can use all Atari keywords either as direct commands or as program statements (instructions with line numbers in a stored program). (Don't you think the use of LIST within Program 30 was neat? Not all micros can do that!) In this section I'll look at the most useful commands other than the ones which involve the cassette/disc interface. Chapter 5 deals with those.

BREAK

This is fairly drastic, but using the BREAK key *is* a command. It tells the micro to stop whatever it's doing and let you take control again. I agree it isn't a command you type in and follow with RETURN, but it would not really be of value if that were the case.

Drastic BREAK may be, but it isn't as drastic as the BREAK feature on some micros. Really the Atari BREAK key has the effect of ESCAPE in other machines—simply causing the current instruction to 'abort'. There is no loss of screen display, nor are the contents of the storage sites used at the time changed. When you press the BREAK key you get a message on screen on the lines of "STOPPED...". Really that is a good message—the BREAK key simply stops what is going on at the time. If you use BREAK within a program instruction, the keyword CONT(R) will allow work to resume pretty well where it left off. Not *quite* where it left off, though—the micro will restart at the line after the one it was on when stopped. In fact I find it hard to tell the difference between the use of the BREAK by the person running the program and the effect of a STOP statement within the program.

Rather more drastic than BREAK is using the (SYSTEM) RESET button. This has the effect of BREAK plus GR.Ø—and with the latter of course comes a clear-screen effect. Even so, the contents of memory are not disturbed. Again CONT will let you continue from the line after that in which the interrupt occurred.

I must say I like the Atari's 'soft' breaks given by **BREAK** and **RESET**. Their use is very kind to programmers busily working on software development.

CLEAR SCREEN

Again some would query whether this is a command or not. All the same I think it useful to put it here. You get the screen to clear when out of program mode by pressing **SHIFT** and **CLEAR** or **CONTROL** and **CLEAR**. We have already looked at ways of getting this effect within a program. Very useful. Very important.

LIST

LIST displays on screen, in line number order, the current program instructions. It does so in Mode 0 only—therefore in the text-window in modes which offer that feature. (If you think about it, you'll see that you cannot give commands to Atari when it is in a mode without text window.) **LIST** has several forms, as you already know.

LIST displays the whole program.

LIST n displays only line number *n* (if it exists).

LIST n,m shows the program section between line *n* and line *m* (inclusive if they exist).

To show the whole program up to, or from line *n*, you have to fiddle as I noted on Page 66. Thus to get the program listed from line *n* onward, use **LIST n,3E4** or **LIST n,32767** if you have very large line numbers. In the same kind of way, get the first part of a program using **LIST 0,n**.

All those last ideas are because when you have a program of more than a couple of dozen lines **LIST** alone becomes rather cumbersome. It's not easy to find the bit you need to look at nor is it easy to edit. (I'm coming to 'editing' in a minute.)

As you know, if your program has more than 21 lines it can't all fit on to the screen at once. The list then 'scrolls' through until the last line appears. **LIST** such and such gets over that. You may prefer instead to stop the scrolling action. That is done by pressing **CONTROL** and **1**. The same key combination lets the scrolling start again, but at any stage, of course, you can use **BREAK** to halt the listing for good. (**CONT** does not allow listing to resume.) Note that it's not wise to press **BREAK** or **RESET** during a scrolling listing.

NEW

This is not new to us (ho, ho). Use it to remove the current program from memory and to put all variable values to zero. Because of that last feature, **NEW** is really more drastic than **RESET**. Even so, it does not return you to the Atari's switch-on state—the colour registers used in graphics control, for instance, remain with any values you gave them since you started. Use of **BYE** is the only way to lead to the virgin switch-on state apart from really switching off and on again.

Don't get into the habit, by the way of switching the micro off and on again to clear the store. This is the most drastic command of all. I admit that the practice shouldn't do any harm if you have a standard model, but it could cause damage to equipment you may later add on, and bad habits started now are hard to remove.

Some folk don't bother with **NEW**, but avoid their mistake! If you have an old program and start entering a new one without using that command, and the new program doesn't use exactly the same line numbers as the old one, you are going to get a crazy mixed-up program—a combination of instructions from two programs in one. That surely won't work the way you want it to!

PRINT

The user of **PRINT** as a direct command has a number of values. (The ‘abbreviation’ “?”, is most useful in this context, by the way.) I think there are three main uses of the command **PRINT**—to do little sums, to be of help when ‘tracing’ *bugs* in your program, and for trying the layout of **PRINT** statements.

I don’t really think that any of those three uses need much further comment from me here, but all the same, being a wordy person, I’ll say a little about each.

Quite often when coding you’ll have the need to do little sums. You may wish to work out, for instance, the centre of the screen in a given mode so that you can use the right values with **PLOT**. The command **PRINT** helps then, **PRINT** being followed by a numeric expression (‘the sum’) and (R). Thus **PRINT 2 + 2 (R)** gives an answer 4 on screen to save you working it out.

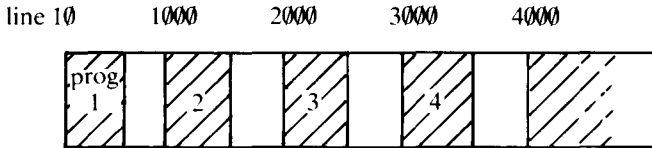
The second use of the **PRINT** command is for tracing bugs. I’ll give you more on this later, but for the moment let me just give you the basic tip. It is that if you find the program not working properly, use **BREAK** and then **PRINT** to give the current value of variables that seem to cause problems. For instance, if the program works fine up to line 200, but has gone wrong by the time it gets to 250, put **STOP** after each line between those two. Then when you run the program, each time it stops, use **PRINT** this, that and the other in order to find the values of the variables concerned. Then **CONT** will let the program go on to the next **STOP**.

The third use of **PRINT** as a direct command is the most obvious one. If you wish to lay your program title neatly out in the centre of the screen, use **POS.X,Y: PRINT “title”** as a direct command with different values of X and Y, until you find the ones that suit you. Then you can put the posh version straight into the program.

RUN

As you know, this tells the micro to start carrying out the program currently in memory from the statement with the lowest line number. It also sets all variable values to zero. You may not want that; use **GO TO n**(first line number) instead if so. The command **GO TO** such and such is also useful in other cases:

1. When you want to start the program running from a different point than the first line, to test how one part alone works, for instance.
2. When you have several separate programs in memory, and want to use any but the first. Figure 9.1 shows what I mean.



Get Program 1 with **GOTO 10** (or **RUN**)

Program 2 with **GOTO 10000**

etc.

Figure 9.1

Some micros let you use **RUN n** in such cases. The Ataris do not. If for no other reason than these, I hope that current distress about the use of **GO TO** in BASIC does not lead to our losing it!

POKE

We use POKE... to place ('poke') a given numeric value into some site in the Atari's store. We've met a number of uses of this already, and I'm not going to say much more about it here. See Appendix 4 if you want more now. All the same it is worth pointing out at this stage that POKE..., being often used to change the way the micro behaves, may well be used quite often as a direct command.

SETCOLOR

Of all the other keywords we have so far met, I find I use SETCOLOR most as a command. The form in which I use it tends to be SETCOLOR 1,0,0—to make the text stand out better on the screen. But sometimes, when I get fed up with the usual weedy blue background, I go for a change with SE.2, something else.

THE EDITOR'S CHAIR

Of late I've been feeling rather guilty at not having told you about how to 'edit' Atari program statements. After all, I've faced you with thirty weedy programs of my own, and all being well you've tried a good number of your own. 'Editing' is often a far simpler and nicer way to correct errors in program lines than simply typing them out again. Check through this section, therefore, best practising on a program in memory—and get used to the techniques.

The full set of editing facilities—ways to change program lines directly—on the Atari is fairly versatile. That makes it a bit clumsy if you are not careful.

The simplest form of editing concerns the complete removal of one or more lines of a program. That's easy! You just enter each line number in question followed by (R). The line now has nothing in it, so *is* nothing—and when next you LIST you'll see it's vanished. But *do* be sure you want to remove the line before doing this! (If you're *not* sure, and you want to see how a program works without a given line, insert REM after the line number. REMember that REM keeps the material that follows in the listing, but the computer ignores it. I'll tell you how to 'insert' in a moment.)

The same kind of technique allows you to replace an existing line with a new version. Thus if line 510 is PRINT ANS\$ and you want it to be INPUT ANS\$ just enter 510 INPUT ANS\$(R)—and this will now appear in future listings rather than the old form.

Fine so far—but it's a bit of a drag to go through all that if the old and new lines are lengthy, or if small changes are needed to many lines. Then we can call on the Atari's full range of 'screen editing' facilities. These use the four *cursor-control* keys—the set of four left of RETURN with arrows in different directions marked on them. The cursor, as I guess you know by now, is the solid square that tells you where the next print item will appear on the screen.

Let's give ourselves something to edit. Enter a couple of fairly complex program lines. Something like these:

```
10 PRINT "This line is wrong."  
20 PRINT "This line needs copying:"
```

How can we correct line 10 to make it right? We may want it to read

```
10 PRINT: PRINT "Now this line is right."
```

We *could* do it by just entering the new version, but let's try screen editing. Keeping the CONTROL key down, press the up-cursor key four times. The cursor dutifully moves up screen to lie over the 1 of line 10. We can now edit line 10. Press the left-cursor key: the cursor appears at the far end of the line. Press the right-cursor key: the cursor goes back over the 1. These two last tricks show that you can edit any single displayed line, whether it's a whole statement or not. That adds power to Atari editing.

It's not going to be easy for me to keep on saying hold the CONTROL key down and press such and such. So during all the editing bit, please remember to hold CONTROL down when using a cursor-control or editing key. (That habit comes quickly.) Now hold the right-cursor key down for a second or two. The cursor skips along line 10 as long as the key stays down. By this means, therefore, we can quickly move the cursor to any point in the line.

Now I'll go through the steps required to change the first version of line 10 to the one we want. Follow through this sequence of key-presses. I'm starting again.

Action (C is CONTROL and)	Result (I show cursor site by =)
1. start	10 PRINT "This line is wrong."
2. C↑↑↑↑	10 PRINT "This line is wrong."
3. C→→→→→→→→→→	10 PRINT " This line is wrong."
4. C INSERT,×4	10 PRINT = "This line is wrong."
5. :PR.	10 PRINT :PR. " This line is wrong."
6. C→	10 PRINT :PR. " <u>T</u> his line is wrong."
7. C INSERT,×4	10 PRINT :PR. " = This line is wrong."
8. Now t	10 PRINT :PR. "Now th <u>i</u> s line is wrong."
9. C→,×12	10 PRINT :PR. "Now this line is <u>w</u> rong."
10. right	10 PRINT :PR. "Now this line is right <u>=</u> "
11. (R)	10 PRINT :PR. "Now this line is right."

There! You've now used a sort of 'word-processor'. In fact, the way word-processors work is rather like that, sometimes even simpler. Complex? Maybe. Easier to do than to describe. Let me summarize.

The cursor controls can move the block cursor anywhere on the screen. When you press CONTROL and INSERT a few times, that number of spaces feeds out from the cursor in front of the next character. These spaces are what you need to insert new characters typed in in the usual way. If you don't make space using CONTROL and INSERT in this way, new characters typed will replace those in the original positions.

You can remove text in the same kind of way. Pressing CONTROL and DELETE a few times makes the cursor appear to 'eat up' that number of characters after it, the rest of the line moving to the left to make up. Try it—put the cursor back at the front of line 10, use the right-cursor key to put it in the correct place on the line, and use CONTROL and DELETE to remove the ":PR." you just added.

Here are some other points about editing you can check through now if you wish, but refer to later otherwise.

1. The BACKSPACE (BACK S) key, without CONTROL, moves the cursor to the left. It wipes out characters over which it passes, leaving spaces instead. You'll rarely need this, as the simple overwrite feature is quite enough.
2. You can use the TAB key while editing in the normal way to move the cursor a number of positions right at a time.
3. SHIFT and DELETE wipes out the whole program line or command you are on.
4. If you don't press RETURN after editing, but move the cursor elsewhere on screen, the changes you made will not count. ALWAYS press RETURN after editing, even if you are not at the end of the line—the whole program line, new version, will replace the old one in the store.
5. Of course if you realise that your editing has gone wrong, *don't* press RETURN—once RETURN is pressed, there's no way you can get the old version back without re-editing.
6. I mentioned a few pages back the value of direct commands to explore the use of keywords. Don't forget that you can edit direct commands in the same way as program statements. All you need to do is to move the cursor on to the command line, edit it as before, and press RETURN. The new version of the command will work at once. This is very useful for finding out, for instance, how SOUND works as you change the values of the numbers after it. Of course you can't do this if scrolling has moved the command in question off screen.

- When you press RETURN after editing a line the cursor will jump down to the next program statement, command, or message on screen. Sometimes this leads to strange effects. Don't worry in particular, if your cursor lands on the READY message, and when you press RETURN you get ERROR number 6. This is just one of the strange effects of Atari editing—it does no harm at all.

SQUASHING PROGRAMS

The Atari offers two main features for reducing programming time and memory. I've mentioned both before: the first is that keywords can be used in short form; the second is that more than one statement can appear on a single program line. The former saves programming time and reduces typing errors; the second reduces memory take-up and can make listings easier to follow.

The micro will accept abbreviated *keywords* when you are entering commands or instructions. In each case, enter one or two letters plus a full stop instead of the full word. Thus you can use PR. for PRINT, L. for LIST, and SE. for SETCOLOR. When lines with abbreviated keywords are listed, those appear in the usual full form and take up the usual amount of memory. The only exception to this is when you use the question mark, ?, for PRINT.

This is obviously a useful technique for the longer common keywords. However, it's not worth the trouble of thinking about it in less common cases. Thinking *is* necessary, as, with several dozen keywords on tap the abbreviation used cannot be just the first letter.

Here's a list of the keywords we have so far met, and their abbreviated form. Those marked with a dagger (†) are either not abbreviations or not worth using. You may think the latter about some of the others too!

AND	AND †	LIST	L.
BYE	B. †	LOAD	LO. †
CLOAD	CLOA. †	NEW	NEW †
CHR\$	CHR\$ †	NEXT	N.
COLOR	C.	OPEN	O.
CONT	CON. †	OR	OR †
CSAVE	CS.	PLOT	PL.
DIM	D. †	POKE	POK. †
DRAWTO	DR.	POSITION	POS.
END	END †	PRINT	PR. (or ?)
ENTER	E.	REM	. (true!)
FOR	F. †	RETURN	RET.
GET	GE. †	RND	RND †
GOSUB	GOS.	RUN	RN. †
GO TO	GO.	SAVE	S.
GRAPHICS	GR.	SETCOLOR	SE.
IF	IF †	SOUND	SO.
INPUT	I.	STEP	STEP †
INT	INT †	STOP	STO. †
LEN	LEN †	THEN	THEN †
LET	(nothing!)	TO	TO †

To confuse the matter further (but to be complete) note also that you can use longer abbreviations than those shown if you want—IN. or INP. (or even INPU.) for INPUT is an example.

Multiple statements on a line is the posh term for structures like the following—I've already often used this kind in fact:

```
100 FOR WAIT = 1 TO 1000: NEXT WAIT: REM delay
```

This contains three statements (FOR..., NEXT..., and REM...). There's nothing wrong with putting them on three lines, but they *can* go on one, with a colon (:) to show the

boundary between them. And on one line, the micro needs less memory for storing them, so can LOAD and SAVE them faster and carry them out more quickly. Also it's easier for you to read the program.

It is logical to use this technique in cases where a small number of lines have one specific task, as with the delay loop routine above. Do *not* overdo the approach, as it can make a program *less* easy to read. However you can have up to about ten statements in one line if you wish. It depends on their length. Here are more examples:

```
120 PRINT CHR$(125): FOR STAR = 1 TO 912: PRINT "“*”"; NEXT STAR:
    FOR W = 1 TO 1000: NEXT W: REM star-fill
240 FOR GO = 0 TO 1 STEP 0: PRINT CHR$(125): POSITION 10,10:
    PRINT "What is your name?"; INPUT NAME$: IF LEN (NAME$) > 3
    THEN LET GO = 2: PRINT CHR$(253)
(250 NEXT GO)
```

Did you really try that latter example? If you did, you will no doubt have come across a major Atari problem for users of long lines. It is that the program line (or command) you enter on screen must not be greater than three screen lines in length. If it is greater than that, all kinds of problems can arise.

However, you can compress lengthy lines to make them fit by using these techniques:

- (a) Avoid all spaces.
- (b) Use abbreviated keywords as much as you can.
- (c) Before entering your line use POKE 82,0. This changes the left-hand margin of the screen display from column two to column zero—therefore giving you two extra characters per line.

If you later list a line that you have cheated with in this way, you will find that it takes up more than three screen lines—but that's no problem now. It's only in entry that that length must not be passed. However never try to edit a program line that's longer than three screen lines without reducing it to the limit.

The second example of complex multiple lines I gave above reminds me of another useful space-saving trick. It is that more than one input at a time can be accepted. In other words, with the Atari you can use structures like this:

```
40 PRINT "Please enter three numbers, with commas between them, and then
    press RETURN."
50 INPUT NUMBER1, NUMBER2, NUMBER3
```

This is in effect yet another useful multiple-statements-on-a-line technique. Try it in *your* programs, watch out for it in mine. (In fact I rarely use it!)

If things get really tight, and you are desperate to save every byte of memory to keep your program running, you'll need to investigate the many other little magic tricks that people use to keep storage needs down. Many such tricks depend on representing numbers in strange ways that take up less storage space than the numbers themselves. But I ought to close this section by reminding you that the higher the resolution of graphics mode you use, the more memory the micro needs to support it. See the table on Page 44.

STRUCTURE?

In some of the later chapters in this book I shall discuss methods of program design. If you read around the subject of computing, you will often find the phrase 'structured programming' in this context. Many people seem uncertain as to what structured

programming is, but this is not really the place to explain it. What structured programming is *not*, however, is screaming if ever a GO TO appears in a listing. It's much more than that, but it is true to say that a well-structured program uses very few such statements, if any.

The essence of structured programming is, I think, preparing a block of code that is easy for other people to follow as well as efficient for the micro. It is surely true to say that if you program includes many GO TOs, then it is *not* going to be easy to follow—by other people, by yourself a few months later, or by the micro. It may well work, and appear to carry out the task set without problems—but it is not 'efficient'.

The programming language we use with the Atari, BASIC, is one in which line numbers are essential. A very useful approach to structured programming is to prepare your code as if no line numbers were required. Then you will have to do without GO TOs, and therefore you will have a clearer, better set-out program. To summarize, then, structured programming is not the avoidance of the GO TO statement. However, a program in which GO TO appears more than a few times is not likely to be a well-structured program.

I must make the comment, here, however, that the Atari's version of BASIC is not one with which you can easily write structured programs. I've done my best throughout the book, but sometimes, I admit, it's been a struggle.

ODDMENTS

I'm sure you know well two of the points I want to put in here—but here is a good place to put them.

The DELETE/BACKSPACE key

Each time you press this, the last character in the material being entered or edited is wiped out ('deleted'). Keep the key down to delete a lot of characters. Users of your programs can use the same key to correct an input answer before pressing RETURN. I wish I had a key like this on my typewriter; it's one of the nice things about text-editing (word-processing)—easy to wipe out errors!

Repeat

If you hold any key down for more than about a second, you'll find its effect repeated as long as the key stays down. This is particularly useful with cursor control and DELETE, but works on *all* keys. Underlining, and printing patterns, become very easy.

FRE

I have been on about squashing program to save memory—FRE lets you find how much you've got left. Like RND, it comes with the nuisance of having to have "(0)" after it. Thus to find how many bytes of storage space are left empty, use PRINT FRE(0). FRE is short for 'memory free', of course.

SUMMARY

We have now sorted out how commands differ from program statements, and seen in detail how to use the main ones. I've taken the opportunity to tell you about editing and to say something about program compression and program structure. The keywords looked at are:

BREAK
BYE
CLEAR
CONT
CONTROL (CTRL)
DELETE/BACKSPACE
FRE
GO TO
INPUT

INSERT
LIST
NEW
POKE
PRINT
RESET (SYSTEM)
RUN
SETCOLOR
STOP

No DIY section this time—you just have to practise all the bits and pieces discussed as you work on your own programs.

10 Valley of Decision

Right at the start of this book (in Chapter 2), I went on about the fact that your Atari can carry out millions of actions in a second. The point then was (mainly, anyway) that this makes essential the 'stored program' concept.

Now it is time to think about *decisions*—the single most important aspect of any programming system. Their importance lies at least partly in the fact that they let us weak humans use the processing power of the micro with greatest effect.

There are two kinds of decision as far as we're concerned. Both make computers very powerful and yet easy to program. Both allow the micro to carry out many actions without too many instructions. I've called the two kinds 'implied' decisions and 'open' decisions. (Note that I made the words up, but I'm not alone in thinking the idea crucial.)

IMPLIED DECISIONS

Without knowing it (I guess) you're very much used to *implied decisions*. They are needed in the loop structures given by the Atari's FOR...NEXT statements. Look at the program fragment that follows. Can you see that it expects (implies) that the micro can make decisions?

```
FOR GO = 1 TO 1000
  (do something or other)
NEXT GO
  (do something else)
```

You can see the structure in Figure 10.1 below; it's rather like Figure 4.1 on Page 38. The decision part, in the diamond, is the crucial bit. Here's what the computer does:

1. Assign the starting value (1 in this case) to the variable 'GO'.
 2. Do something or other.
 3. Increase ('increment') the value (by 1 in this case).
 4. Test whether GO has reached the final value (here 1000):
 - (a) if not (false) go back to step 2;
 - (b) if so (true) go on to do something else.
- } Crucial bit

Each time round the loop, then, the computer carries out a test; it 'decides' what to do next as a result of that test. The need for test then decision is *implied* in the loop structure FOR...NEXT (and the REPEAT...UNTIL and WHILE...ENDWHILE you may meet with other micros).

Here's a second program fragment that has just the same effect. This time the decision is *not* implied—it is out in the open. That's why I call it an *open decision*. We have met some examples of its main structure, IF...THEN, already.

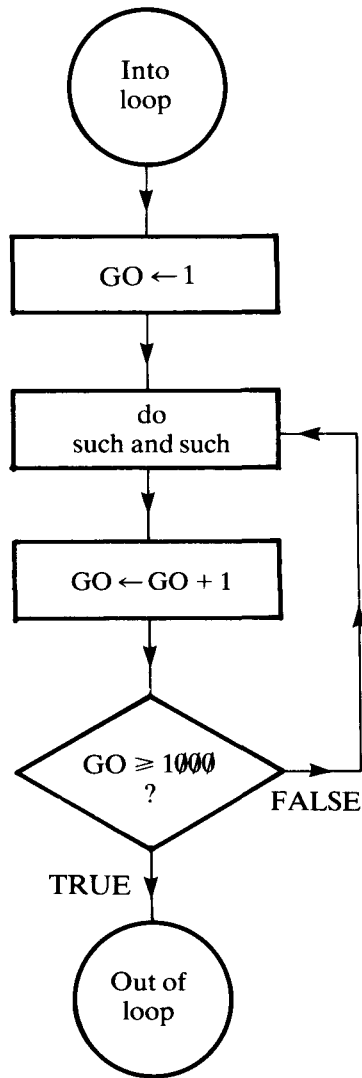


Figure 10.1

```

110 LET GO = 1
120 (do something or other)
130 LET GO = GO + 1
140 IF A < 1000 THEN GO TO 120
150 (do something else)
  
```

I hope you can see a couple of things here. First, do you agree that it is in effect just the same as before? That, in other words, Figure 10.1 closely describes it? What *this* approach does is simply bring the decision that was implied out into the open.

Something different may be slightly less obvious—I've had to use line numbers. This is because of the GO TO statement—the micro has to know where to go. Programs without GO TOs and the need for line numbers are much 'nicer' than ones with. As I said at the end of the last chapter, though, I do advise you to keep the use of GO TO as rare as you can. That will help to make your programs better structured—easier to follow.

IF...THEN

We can use this open decision structure in many more cases than just for looping. (In fact, why use it for looping anyway? FOR...NEXT is much better!) Decision-making is, as I've said, very powerful, and one of BASIC's most valued structures. Here's how it works:

IF (something is true) THEN (do this) otherwise don't.

As soon as the computer gets to the IF, it carries out the test 'is something (whatever it is) true?'. IF it is true THEN it carries out the next instruction. IF not THEN it doesn't.

Two points increase the power of this concept in the case of the Atari (and some other machines).

1. The THEN doesn't have to have GO TO after it: *any* keyword can do:

```
IF SCORE = 10 THEN PRINT "Excellent!"
```

Indeed, after what I've said before, you'll know that I'd advise you *not* to use GO TO after THEN!

2. The multi-statement approach works with THEN; use colons as usual:

```
IF SCORE = 10 THEN PRINT "Excellent!":  
    LET BONUS = BONUS + 1:  
    GOSUB REWARD
```

We don't need all those spaces in that last example; I put them in to make what's going on clearer. *It is very easy to become confused in IF... lines.* Figure 10.2 shows what that last example does. Complex! Please take *great* care with IF.... Otherwise you'll spend ages trying to find out why your precious program doesn't work properly. You have been warned! To make matters worse, you don't actually need the GO TO before a line number after THEN:

```
IF ANSWER < TARGET THEN 600
```

This is an "implied" GO TO, but it's just as bad as a real one!

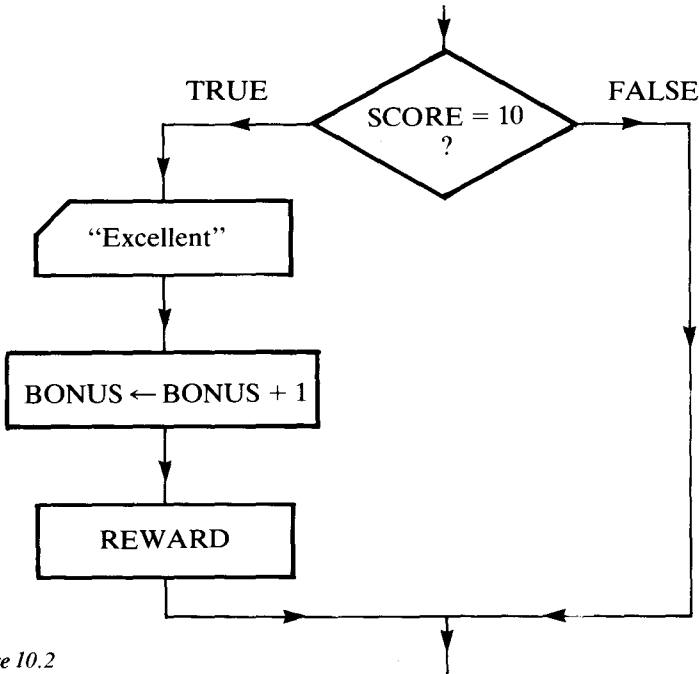


Figure 10.2

I'm really telling you all these last points so that you recognize them in program listings. I strongly advise you that if you *must* use line numbers always put in GO TO rather than implying it. Thus your programs become easier to read, less prone to error, and less hard to check. And here's a final, *very* important warning . . . *This works:*

```
100 IF ANS$="cat" THEN PRINT "Good!"
110 PRINT "Now try the next one."
```

Whatever the value of ANS\$, the message "Now try . . ." appears. *This* looks as if it should do the same:

```
100 IF ANS$="cat" THEN PRINT "Good!": PRINT "Now try . . ."
```

But the second message comes on screen *only* if ANS\$ = "cat".

The colon multi-statement idea needs special care in IF lines. The reason is that in such a line, this is what the computer does:

1. Carry out the test after IF.
2. (a) If the test gives a TRUE result, THEN carry out all the statements in the rest of the line.
(b) If the test gives FALSE, ignore the rest of the line.
3. Go to next line.

Check that with my second "cat" example.

So—to repeat, yet again, *it is very easy to become confused in IF... lines!* I must say that things get even tougher if you use IF...THEN IF...THEN... See next section.

The program that follows should help you to understand the versatility—and dangers—of IF. Enter it, try to predict the outcome of different values of A, and see if you are right. If you are really keen, try to draw a diagram of it like those earlier in this section. And if you are keener still, rewrite it so that it is much clearer.

Program 32: IF what?

```
10 FOR GO=0 TO 1 STEP 0
20 LET A=INT(RND(0)*5)
30 IF A=1 THEN PRINT 1
40 IF A=2 THEN GOTO 10:REM ** OK, naughty!
50 IF A=2 THEN PRINT CHR$(125)
60 IF A<3 THEN PRINT A*100
70 IF A=3 THEN PRINT "A=3"
80 IF A>2 THEN IF A<5 THEN FOR B=1 TO 1000:NEXT B
90 IF A*A<20 THEN PRINT A;" squared is ";A*A;"."
100 IF A=4 THEN SOUND 1,2,3,4:GOTO 10
110 IF A<3 THEN STOP
120 IF A=5 THEN SETCOLOR 2,A,10
121 REM ** Catch here....
130 IF A<>3 THEN PRINT A;" mess!"
140 NEXT GO
```

MORE IFs AND BUTS

Quite often we need an IF...THEN IF...structure, as I just warned you. We may need it, but beware. . . . "IF it is raining THEN IF I have an umbrella THEN I'll use it." The same is true of computing as of real life—"IF the score is ten THEN IF there've been ten goes THEN congratulate and finish." Sometimes we may need even more complexity—IF...THEN IF...THEN IF...THEN and so on. I shan't try to give an example; I'm sure you take the point.

We can re-word the raining sentence like this. “IF it is raining AND I have an umbrella THEN I’ll use it.” And we can do the same in BASIC: “IF the score is ten AND there’ve been ten goes THEN congratulate. . . and so on.”

The AND here is a useful keyword. Strictly its name is *logical operator*. Here’s what that last line would actually look like when coded:

```
IF SCORE = 10 AND GOES = 10 THEN PRINT “Full marks!”; GOSUB STOP
```

Take a look at Program 31 (Page 91); this shows how this AND business works in a sort of game. See what’s going on?

Another logical operator of value in this context is OR. With care (again!) you can compare it to the English ‘or’—IF it is raining OR it is snowing AND I have an umbrella THEN I’ll use it.” So—in BASIC:

```
IF SCORE = 10 OR SCORE = 9 AND GOES = 10 THEN PRINT “A good score.” . . .
```

and so on.

But you *can* see the rocks of complexity again, can’t you? Fortunately we can use brackets to ensure the right priority for the computer if we wish:

```
IF (SCORE = 10 OR SCORE = 9) AND GOES = 10 THEN . . .
```

or even (to be completely sure)

```
IF ( (SCORE = 10 OR SCORE = 9) AND GOES = 10) THEN . . .
```

LET’S BE LOGICAL!

I just introduced you to the *logical operators* AND and OR—but I made no special comment at the time because their use should have been clear enough. There are some cases when one uses these and other logical operators where the use is *not* so clear. Let’s recap a little. You may have a line like this in a program, following an input:

```
LET FLAG = 0: IF A < 0 OR A < > INT(A) THEN LET FLAG = 1
```

This will, I hope you see, set the ‘flag’ (make FLAG equal 1) in *either or both* of the cases:

1. If A is not positive.
2. If A is not an integer (whole number).

There is another way we can arrange for the same thing to happen. What we want is that FLAG stays at zero only if the user supplies a positive whole number. In other words A must be positive and A must be integral—if *not*, then the flag is set. That kind of thinking gives the following version of the line.

```
LET FLAG = 0: IF NOT (A > 0 AND A = INT(A)) THEN LET FLAG = 1
```

Make *sure* you understand that this is just the same in effect as the original.

We gave AND, OR, and NOT the name ‘logical operators’ because they follow certain rules defined in the science called logic. In general language we know that ‘logic’ involves precise thinking. How we use these structures must be precise as well.

The examples we have met so far have not, I think, been too hard to follow. However the crucial line in the next program is not so simple. Line 90 is the one to study.

Program 33: Fail safe

```
10 DIM N$(5):LET YES=0
20 FOR GO=0 TO 1 STEP 0
30 SETCOLOR 2,YES,RND(0)*12:SETCOLOR 4,15-YES,RND(0)*15:SETCOLOR 1,0,15
31 REM ** Guided random colours
40 PRINT CHR$(125):FOR A=0 TO (YES*2)+1:SOUND 1,250-RND(0)*YES*5,10,10:NEXT A:SO
UND 1,0,0,0
```

```
41 REM ** Guided random sounds
50 POSITION 10,10:PRINT "Give me a number!"
60 FOR I=0 TO 1 STEP 0:POSITION 2,11:PRINT "      ":POSITION 2,11:INPUT N$:IF N$
>"0" AND N$<"9999" THEN LET I=2
70 NEXT I
80 LET N=VAL(N$):IF N<>INT(N) THEN GOTO 60
90 LET CASE=2:IF N=20 OR (N<=10 AND N>5) THEN LET CASE=1
100 PRINT CHR$(125):GOSUB CASE*1000:FOR W=1 TO 1000:NEXT W:PRINT CHR$(125)
110 NEXT 60
999 REM ** Success & maybe win!
1000 LET YES=YES+1
1010 IF YES=5 THEN POSITION 15,12:PRINT "You win!":FOR R=0 TO 1 STEP 0:SETCOLOR
1,RND(0)*15,RND(0)*15:SOUND 1,RND(0)*255,10,10:NEXT R
1020 SETCOLOR 2,12,5:SETCOLOR 4,4,2:FOR L=2 TO 22 STEP 2:POSITION 17,L:PRINT "Sa
fe!":NEXT L:FOR F=1 TO 90:SETCOLOR 1,0,RND(0)*15:NEXT F
1030 RETURN
1999 REM ** Failure
2000 IF YES>0 THEN LET YES=YES-1
2010 SETCOLOR 2,0,0:SETCOLOR 4,0,0:SETCOLOR 1,0,14:FOR A=1 TO 100:POSITION RND(0
)*32,RND(0)*22:PRINT " FAIL! ":SOUND 1,250-RND(0)*50,0,10:NEXT A
2020 SOUND 1,0,0,0
2030 RETURN
```

The aim of the ‘game’ is for the user to input non-negative integers (whole numbers) until he or she wins having found enough acceptable ones. There are in fact seven such here, and to win you must get five accepted ones in a row—you may like to extend the program to punish the cheater who gives the same safe number each time!

There are a few fairly snazzy ideas around for livening up the display with colour and sound—lines 1010, 1020, and 2010 in particular.

I introduce the program as an exercise in logical expressions—line 90. However such expressions also appear elsewhere for various reasons, and I hope you’ll look at those as well.

With what input values would this program *not* ‘fail’? I would like you to attempt to work out the answer before you play the ‘game’ or let others have a go. The aim is to win by finding out which numbers are safe to input. That means always being able to input different, but valid numbers. However, as far as you are concerned, the aim is to help the understanding of AND and OR. Try variations on line 90 for yourself.

That single line of tests can be set out graphically with this table:

Table 10.1

Input	Result
$N = 20$	SAFE
$5 \leq N \leq 10$	SAFE
All other values of N	FAIL

It could have been written in other ways with the same effect, and I am sure you can adapt the program to make a much more exciting one yourself when you have sorted out the logic. I use it as an example to show two things—that we can combine logical operators to cover various tests in a single statement, and secondly that we can combine them to save quite a lot of memory.

Logical concepts, based on what is called Boolean algebra and the evaluation of logical expressions, are of great importance in more advanced programming. This book is not the place to go deeper into the ideas, but I would like to put before you one or two that you may care to think about. Skip the next bit if you like!

Example 1

```
LET B = A = 10
```

This is the same as `LET B = (A = 10)`. What the computer has to do is to work out first whether `A = 10` or not. If it is, then the logical expression in brackets is `TRUE`, and the Atari gives it the value 1. Otherwise the expression is `FALSE` and takes the value 0. So now the assignment means the following:

```
IF A = 10 THEN LET B = 1
```

```
IF A < > 10 THEN LET B = 0
```

Or, more simply

```
LET B = 0: IF A = 10 THEN LET B = 1
```

Both are much less wieldy than the logical expression

```
LET B = A = 10
```

Example 2

```
GO TO 100 + 100 * (A = 10)
```

This replaces

```
IF A = 10 THEN GO TO 200
```

```
GO TO 100
```

Okay, I admit I shouldn't use `GO TO`—but you'll find such structures as this in other people's programs even if you don't want to use them yourself. Again we use the fact that the value of a logical expression is 1 if the expression is a `TRUE` statement, and 0 if it is `FALSE`.

Example 3

```
100 INPUT ANS$: GOSUB TEST: IF FLAG THEN SOUND 1,255,0,10
```

Here “`IF FLAG . . .`” means “`IF FLAG = 1 . . .`”. The subroutine `TEST` would set `FLAG = 1` if the test failed.

Example 4

```
IF this AND NOT that THEN PRINT the other.
```

This will print the value of ‘the other’ only if ‘this’ is non-zero *and* if the value of ‘that’ is zero.

Note that the Atari can deal with such structures as `PRINT X AND NOT Y` (I'll leave you to work out what this does for various values of `X` and `Y`!). It can deal with `OR NOT` too.

Everyone learning `BASIC` for the first time gets into some confusion because of the new meaning of the ‘`=`’ sign used in assignments. Thus the structure `LET SCORE = SCORE + 1` does not seem to make sense. By now, I hope, you have come to terms with the assignment symbol, `=`, but I know that it is confusing. I suggest you always read it as ‘becomes’, or even ‘takes the value of’, rather than ‘equals’. In the first example above `LET B = A = 10` we have *both* uses of that symbol together.

All the same, hard or not, it really is worth your spending some time on trying to get on top of logic.

CONCLUSION

IF is *very* important, and its use can be very flexible. However, that flexibility leads to all kinds of possible dangers—TAKE GREAT CARE WITH IF!!

IF you are human, THEN by now you are probably getting a bit lazy with the projects in these chapters. But please take some time over the ones that now follow, otherwise you are likely to have many problems in future. . . .

SUMMARY

In this chapter we've explored these keywords:

AND	NEXT (N.)
FOR	NOT
GO TO (GO.)	OR
IF	THEN
LET	TO

Implied decisions, as in loops and logical statements, and open decisions have been my main topic.

DO IT YOURSELF

1. Code the loop shown in Figure 10.3 in two different BASIC ways.
2. Write your own program like (but better than!) Program 33.
3. Find out what 'truth tables' are. Now write a program which will display truth tables for logical expressions involving AND, OR, and NOT.

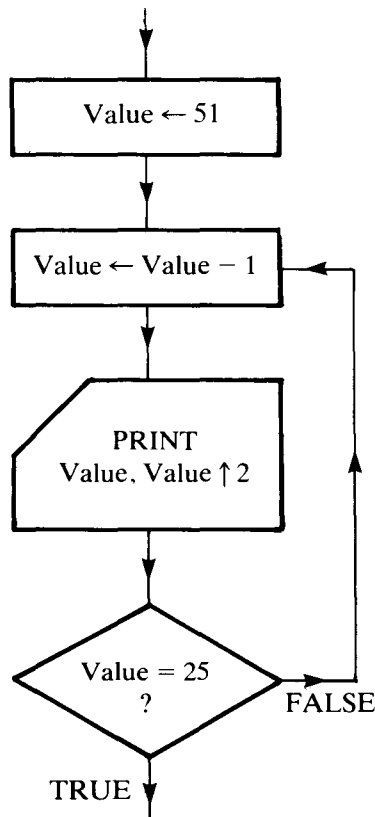


Figure 10.3

11 Get the Picture

I'm not going to tell you much in this chapter about how to program your Atari—as far as using its BASIC is concerned anyway. But please don't skip these pages—truly this is one of the more important chapters in the book. Here I shall attempt to give you at least a few ideas about the concept of *programming* itself. Some of the ideas have appeared before, in Chapter 6 especially, but that too is no reason to skip!

As you know, programming (or 'coding') means the design of a set of instructions, in a form that the computer can follow, to carry out some given task. Most of the material in this book concerned with the actual 'language' of programming is, in fact, to ensure that the instructions in *your* programs are in a form that the computer can follow. All the same, there are broader ideas that we need to bring in. If you skip this chapter—fair enough, it's your book, yet I do think you'll be missing an essential ingredient in the future successful growth of your hobby.

PLANNING

In Chapter 6, I discussed this in general. I said that the 'essence of good habits in programming is *planning*'. Indeed I went further and said that 'good programmers plan their planning'. I then introduced the idea of 'top-down development', and said that program design aims at software that is efficient and friendly.

What top-down development entails, briefly, is to develop an idea, stage by stage, in a logical kind of way, so that by the time you need to do the coding itself, there is really no problem. Well, not much anyway. . . . Here are the steps involved:

1. Have an idea for a program: suggest a problem that can be solved best with a computer.
2. Work out, from the user's point of view, what the program should do—in other words, define the problem.
3. Work out, from a programmer's point of view, how the software should do it—define the solution. This stage involves producing what folk call an *algorithm*. An algorithm is a solution to a problem, set out as an ordered set of logical steps. It may appear as a series of plain-language sentences, or in graphical form (a "flowchart"), or both.
4. Code the program section by section, with each section relating to a step in the algorithm.

In a minute I'll go into this top-down development in detail, showing you how it works in practice by designing a program from scratch. But first I think you ought to know about those flowcharts.

PROGRAM PICTURES

Folk in computing use a special kind of diagram that shows graphically the action of different parts of their programs. The diagram is called a *flowchart*. It is as important for a programmer (a 'software engineer') to be able to develop and to read flowcharts, as it is for an electrician (or an electrical engineer) to be able to handle circuit diagrams. Indeed the concepts are very much the same. This is because in both cases:

1. The diagram can show anyone very quickly and efficiently what is going on.
2. A diagram is easier to check than the structure it represents.
3. A diagram is widely understood as a simplification of reality.

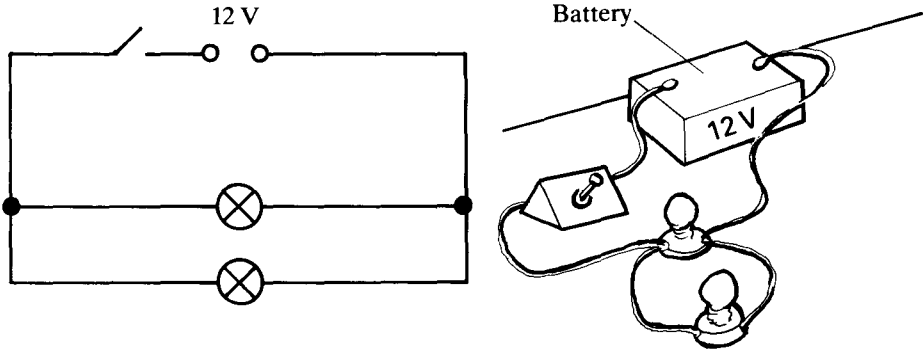


Figure 11.1

Compare the circuit diagram in Figure 11.1 with the actual circuit beside it. Which better shows a reader who knows the rules what the circuit's about?

I also find it useful to compare the use of flowcharts with that of the well-known London Underground (subway) map that appears in so many British diaries (even for people who never go to London!). Such a map is far easier to use than a normal one would be.

I must admit there are quite a number of different flowcharting standards. However the variations appear mainly in advanced use. It is the concept that matters, and my concept here is that you should be able to sketch out in picture form the structure of the programs you develop. It's a useful skill, surely much simpler than writing/reading algorithms set out as wordy sentences. I'll give you here the 'rules' and the corresponding symbols. Not many rules, don't worry!



Figure 11.2

1. A flowchart should have only one start and (preferably no more than) one end. The symbols we use are the START and STOP "boxes" shown in Figure 11.2.
2. The direction of flow is from START at the top to STOP at the bottom. The normal direction of horizontal flow is towards the right. The symbols for flow are straight lines joining the boxes concerned, with arrows at *least* where the normal directions

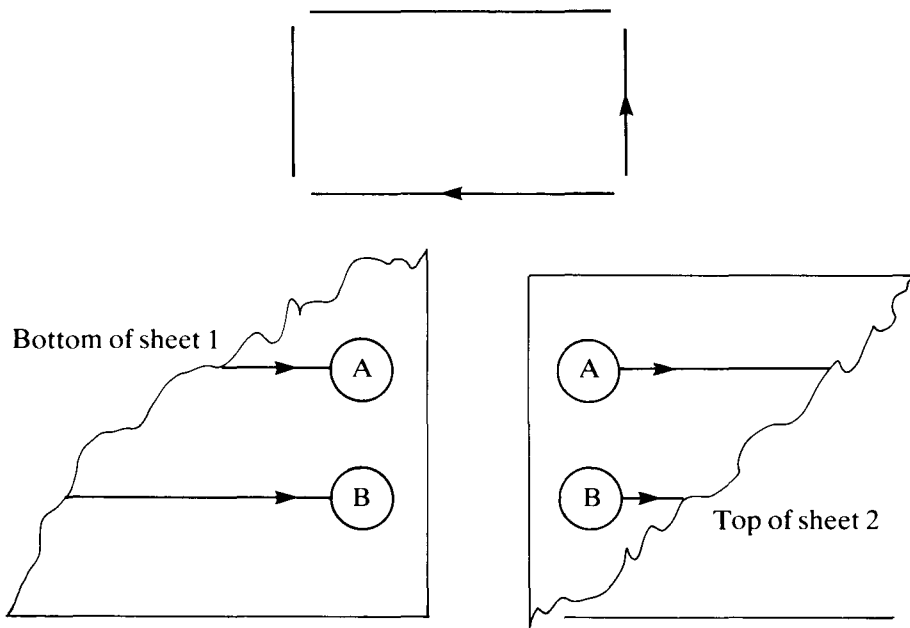


Figure 11.3

are *not* followed. Refer to figure 11.3; this also shows the 'connector boxes' that we have to use if a chart covers more than one sheet of paper.

3. Each stage of the algorithm leads to a box in the flowchart. The symbols we use for INPUT, OUTPUT, DECISION and any other kind of ACTION appear in Figure 11.4.

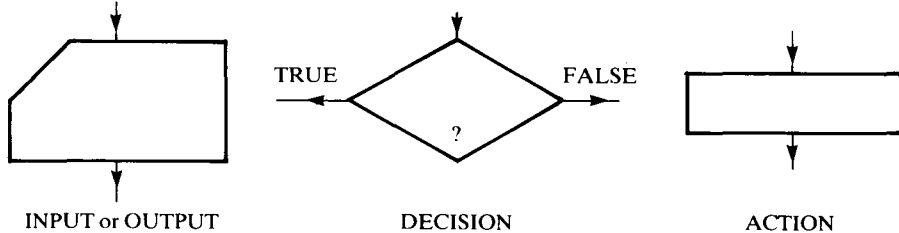


Figure 11.4

And that's it. Easy, eh? No? Now come on—look again at Figures 3.11 and 10.2 (for instance). You see, I already used flowcharts in this book and you didn't have any problem with them then, did you? The only difference now is that I want you to start thinking about drawing them yourself.

When I teach science (which is officially my job . . .) and get on to circuit diagrams, I give practice in two ways. Firstly, given a circuit description, prepare the diagrams; secondly, given a diagram describe or prepare the circuit. My pupils and students need to be very familiar with circuit diagrams. And it's just as true of flowcharts. They are very useful indeed—so there are projects on them at the end of this chapter. But in what ways *are* they so very useful?

Not so long ago—a few years—programmers had to develop flowcharts in great detail before starting to code any of the program itself. This was because computer keyboard time was so costly. In those days there were at least two stages of flowchart development. First one would write an outline *flowchart*, showing in not more than a couple of pages, the very broad progress of the project. Then one or more *detailed flowcharts* would set out almost line by line what the program would have to do.

I think that most programmers now feel there is no need for detailed flowcharts. The program itself should be very well laid out (structured), and quite clear to the casual glance. Indeed some people go further than that and suggest that the outline flowchart, too, is less than essential, but if it has to be drawn, it should be done *after* the program is finished. That's up to you. Some people find flowcharts very helpful indeed to give them a good mental picture of what the program is to do. Lots of us need mental pictures, after all. What you draw doesn't need to be neat, but it *does* need to be clear. Cross my heart, the flowchart habit will save you a lot of blood, sweat, and tears in the long run. . . .

TOP-DOWN DEVELOPMENT

Now, at last, I can set out that approach in a fairly concise way. Using the words I have used above, here are the stages of sensible development of a program:

- Stage 1* Have a brilliant idea . . .
 THEN break it down into concepts (modules).
- Stage 2* Sketch the outline flowchart . . .
 Each box relates to a module.
- Stage 3* Write the program section by section . . .
 Each subroutine relates to a box in the flowchart.

Well, that doesn't look very difficult—but I agree there is a much simpler way:

- Stage 1* Have an idea.
- Stage 2* Write the program.

However, life is rarely that easy! Program coding is much more like writing a letter of application for a job than writing a letter to a friend—what comes out must be very precise in the way it deals with its task. You must plan it. If you are writing to a friend you can of course get away with things like “Oh, by the way . . .” and “Well, anyway” and “PS”—but in a program you can't! Well, you *can*, but the end-product is not likely to be a good program; more likely it'll be what we call ‘spaghetti code’—a real mess! I would like to show the above points by developing a fairly simple program using these three stages. I'm afraid it *must* be a simple program, because I'm fairly short of space—and that makes it rather a trivial one. But then, if it's short, simple, and trivial, it should show more easily what I am trying to put across.

IDEAS TO MODULES

This is Stage 1—have an idea *then* break it down into its modules. Program specification is the posh name. The *idea* is:

“I want a program to convert a length input in metres to inches.”

Already you should be able to see some sub-ideas (concepts) in that simple statement. Here they are:

1. Accept length in metres.
2. Convert to inches.
3. Give the result.

This set of instructions would be quite enough if we were telling a reasonably mature person what to do. Unfortunately micros are not (yet?) as clever as people, so if we're instructing them we must be more precise:

1. Ask for a length in metres.
2. Accept a number; reject any non-numeric input and restart.
3. Work out the number of inches.
4. Give the answer.

That's getting much more precise, though of course no computer in the world (as far as I know) will accept instructions even in this form. Still, before we get down to BASIC, there are other things we ought to introduce. For one, in practice we want our programs to restart automatically until told to stop, rather than stopping at the end of one run. After all, the user is fairly likely to want to convert more than one length into inches, he or she would not bother with a program to do the task only once. Also, in practice, it is important that a program helps the user as much as possible—you know I describe this as being 'user-friendly'.

The best way to get a program to restart until told to stop involves using what folk call a *rogue value*. We program the micro to recognize this and stop, but to keep on going if it doesn't find that value. I will use the capital S for this particular purpose. S for Stop, see?

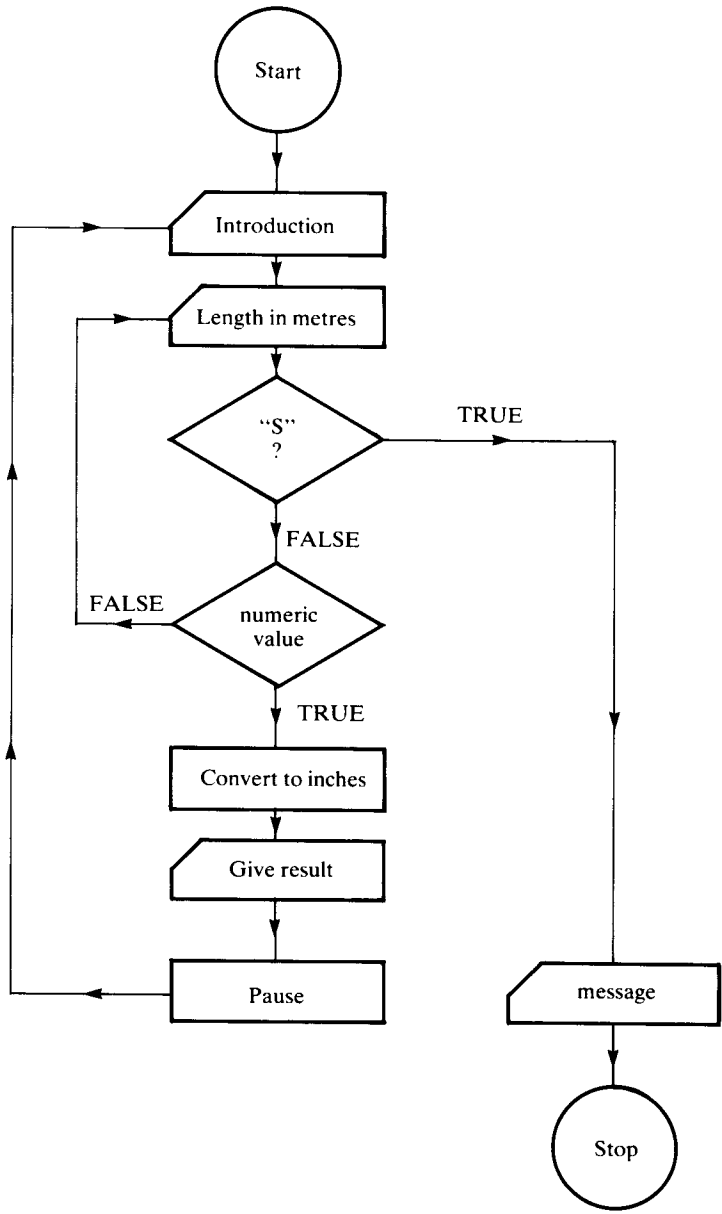


Figure 11.5

Introducing these two ideas into our list of concepts gives us a final list of what I think I can now call *modules*. This list of modules is our algorithm. Here it is.

1. Clear screen and say what the program does.
2. Politely ask for a length in metres. Get input.
3. If input is S, stop with polite message.
4. If input is non-numeric, politely reject and ask again.
5. Convert to inches.
6. Give the result with a suitable message.
7. Pause and then restart.

MODULES TO FLOWCHART BOXES

And now we reach Stage 2. This is very easy—all we have to do is to express the above list of modules in diagrammatic form, in other words as a flowchart. This must clearly show the exact lines of flow as the program progresses. See Figure 11.5.

Please check that the boxes in this flowchart correspond to the modules listed above in the algorithm. (It is, by the way, worth pointing out at this stage that program design is never quite as precise as I may make it sound. Quite often one finds at flowchart stage and/or when actually coding, that changes must be made to the list of modules. The reason is that each next stage of program design takes more account than the last of what a computer can actually do, while the first stage, the specification, is most like a dream!)

CODING THE PROGRAM

This phrase means turning the modules/flowchart boxes into lines of program instructions in a form acceptable to the computer—Stage 3 of the development. (The noun ‘code’ is often used to mean computer programming ‘language’.) If all has gone well so far, coding the program should pretty well be a doddle. Our only worries need be how to turn each module into BASIC, for we should be fairly sure that the modules relate correctly to each other.

Our approach is to write the program lines that carry out the function described in each flowchart box. Those lines can either be part of the main program (in which case they form a so-called *open subroutine*) or they can go as a separate chunk, a *closed subroutine*. I’ve used closed subroutines in this book before (recall the keywords GOSUB and RETURN), but I don’t need to use them here. We should be able to test the program as we go along, RUNning each section when coded.

However, before we finally get going on the code, it is good practice to set out a ‘memory map’ of the program. This involves allocating blocks of line numbers in advance to at least the major parts of the program. Such a memory map will be extremely complex where large fiddly programs are concerned, for these will have many separate blocks of closed subroutines and many separate open subroutines in the main program. Our program is simple, however, so the map is as well—see it in Figure 11.6.

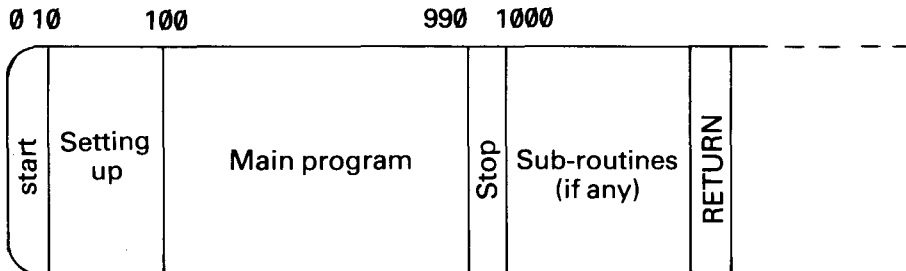


Figure 11.6

Notice that the main program itself begins at line 100, after the various *initialization* (start-up) lines we may have to include. And I'm planning any closed subroutines to go from line 1000. OK, we shouldn't have any closed subroutines here, but in practice, you may want to dig some out of your subroutine library (see Page 66) so I'll leave them on the map.

Here's the starting skeleton for the program then, note the free use of REM statements.

Program 34: Minches (under construction—enter and test each block)

```

9 REM ** Initialisation
10 PRINT CHR$(253)
11 REM ** Buzz user who loads with RUN"C:..."
12 REM ** Useful to end REMs with 9 or 1
99 REM ** Main program
989 REM ** End main program
990 STOP
991 REM ** Block accidental entry to sub-routines
998 REM ** Sub-routines
1990 RETURN
1991 REM ** Lest we forget!

```

Module 1: clear screen and say what program does.

```

20 PRINT CHR$(125)
21 REM ** Clear screen
30 SETCOLOR 1,0,14:SETCOLOR 2,6,0:SETCOLOR 4,2,4
31 REM ** Restful and pretty?
100 POSITION 4,5:PRINT "M E T R E S   T O   I N C H E S":PRINT " =====
===== "
101 REM ** NB layout
110 POSITION 4,10:PRINT "Give a length in metres, and I'll   tell you how many i
nches it is."
120 FOR W=1 TO 1000:NEXT W
121 REM ** Time to read!

```

RUN—still no problems? Good!

Module 2: politely ask for a length in metres. Get input.

```

40 DIM M$(9)
130 POSITION 4,13:PRINT "How many metres"
131 REM ** No, I've not forgotten my grammar!
140 POSITION 4,15:PRINT "Please type  a number":PRINT "and press  RETURN...."
160 POSITION 19,13:INPUT M$

```

RUN again to test how we are so far. In truth, there's not much chance of error yet! No point in doing anything special with the INPUT bit—that can come later. Just press RETURN. However, you may ask why a string input to get a number—well, wait and see! I hope you find the layout OK, by the way—in particular, the trick I used in lines 130 and 160.

Module 3: if input is 'S', stop with polite message.

```

150 FOR W=1 TO 200:NEXT W:POSITION 4,18:PRINT "Or press S/RETURN to STOP."
170 FOR L=13 TO 19:POSITION 0,L:PRINT "                                     ":N
EXT L
180 IF M$="S" THEN PRINT "S pressed: program stopped.":PRINT "RESET, RUN, RETURN
to re-start.":FOR W=0 TO 1 STEP 0:NEXT W

```

RUN to test this, now—the program should 'blank out' the keyboard with the stopped routine if you press the 'S' key, but otherwise go to the STOPPED message. See what I mean about being able to test as you go along in the top-down approach?

Module 4: if input is non-numeric, politely reject and try again.

```
50 TRAP 2000
51 REM ** Implied GO TO! - Sends control here on ERROR
190 LET M=VAL(M$)
191 REM ** Convert to number if possible; else let TRAP, line 50, act.
1999 REM ** TRAP routine: Deals with non-numeric inputs
2000 SETCOLOR 1,0,0:SETCOLOR 2,2,10:SETCOLOR 4,2,12
2010 POSITION 4,16:PRINT "Please enter numbers only!"
2020 PRINT:PRINT "Let's try again...."
2030 FOR W=1 TO 2000:NEXT W
2040 RUN
```

Sorry, but I couldn't avoid bringing VAL in here. (She's *such* a useful lass.) What VAL is is a 'function' that acts on a string. If the string comprises the characters of a number, VAL gives that number. Otherwise it leads to an ERROR. To handle the ERROR, I used TRAP—see it in line 50. Line 50 sends the computer off to line 2000 if any ERROR appears while the program is running. It clearly involves an implied GO TO!

RUN to test this feature. You should get the ERROR-trapping routine if you don't enter a number at the input stage. Now do you see why I used a string INPUT earlier rather than a simple numeric one?

That line 50, once tested, is going to be a hazard as the program develops. I mean that any error that occurs will now be trapped by it. So use your editing facilities to insert a REM after the line number. And don't forget to pull it out when your program is finished. Now you can RUN and reRUN to your heart's content, checking that only numeric inputs let the program proceed to the STOPPED message.

Module 5: convert to inches.

Easy-peasy, as my children say. Well, as long as you know that there are 39.37 inches in a metre.

```
200 LET IN=39.37*M
201 REM ** Conversion at last
```

RUN to test if you like, but as this module/subroutine is pure processing, you won't see anything new.

Module 6: print the result with a suitable message. Here we go:

```
220 POSITION 4,16:PRINT M;" metres is ";IN;" inches."
```

Isn't this easy? RUN to test: reRUN to retest, again and again. Happy? Well, maybe not. Here comes Deeson on his dashing white hobby-horse again: this time with yet another digression.

Metrication arrived in Britain in a big way in the late 1960s. With it came all sorts of problems. One problem was what happened when uncertain sub-editors (so unlike those at Shiva Publishing) tried to convert an author's 'feet' into 'metres'. Thus if the author wrote "He was about six feet tall", the sub-editor would translate the sentence to give "He was about 1.8288 m tall". True! Can you see the error? The result of a calculation should not be much more precise than the data it used. 1.8 m would have been OK.

At this stage in our BASIC programming we'd find it too hard to advise how to restrict the answer to say one more significant figure than the input (if you know what that means anyway). So we'll just give the result to the nearest tenth of an inch and hope for the best. We have to bring in another function—not such a pretty one as VAL, though. It is INT.

Recall we met INT before, more than once in fact? We'll get to proper grips with this function in Chapter 13 (if you can wait that long)—I'll just add the line we need:

```
210 LET IN=INT(10*IN+0.5)/10
211 REM ** Round to one decimal place
```

That's a very useful trick for folk doing mathematical programs. Rounding to three decimal places would have 1000 instead of each 10, and so on. Still, as I say, we'll deal with that shortly.

Now RUN and test a few times to see how INT has improved the value of our program.

Module 7: Pause and then restart.

The user-friendliest way uses the good old input-to-go-on idea:

```
60 DIM CONT$(1)
230 FOR W=1 TO 500:NEXT W:REM ** Simple pause first
240 POSITION 2,21:PRINT "Ready to go on? Please press RETURN!"
250 INPUT CONT$
260 RUN
```

We should be pretty well finished now—so remove the REM from line 50. Then RUN and test again as fiendishly as you like. Nothing should go wrong—after all, top-downly we coded and tested each part!

So now we have a fairly good implementation of the original idea (though I admit that original idea wasn't particularly fantastic). The program as it ended up above can certainly be improved on, and it can certainly be made more complex and useful. But remember that I included it only as a demonstration of the top-down development of programs!

Figure 11.7, if such things turn you on, is a detailed flowchart of this program. Compare it with the outline flowchart given before. Try drawing such pictures of your own programs occasionally as an exercise. It's quite fun. Honest!

Bearing in mind all I've said about *user-friendliness*, I thought I'd do a quick check on how much of Program 34 comes under this heading. First I removed the REMs, for these are programmer-friendly, rather than user-friendly; then I roughly separated the user-friendly statements from those truly essential to the program. I did it only roughly, but found that the essence of the program requires well under 200 bytes, but adding user-friendliness meant putting in well over 1000 more.

Someone said that about 90% of any decent program would concern user-friendliness—clearly that's not far out. Let me make a couple more points about program design/development before I get back to coding itself again.

Firstly, I hope I have shown the need to test and test again throughout the development process. No program is ever perfect, but it comes closer to perfection the more you test and polish it. Top-down development allows testing to be made much more simple, but don't think you can avoid it!

The second thing concerns those REMs I so bravely wiped out a few lines back. When you build up a program that is not trivial, in other words that is more than a few lines in length, it is crucial to keep some kind of record about what you are doing, how, and why. The simplest way to do that is to include lots of REM statements in the listing. Then when you SAVE your program, you save all those details too. On coming back later to finish, extend or polish the program, you will have no problem in understanding what's going on. If, however, your program is very long, those REMs are going to take up a significant part of memory. Then they are not as useful as having detailed notes on paper. Those notes should form part of the *file documentation* that we discussed on Page 67.

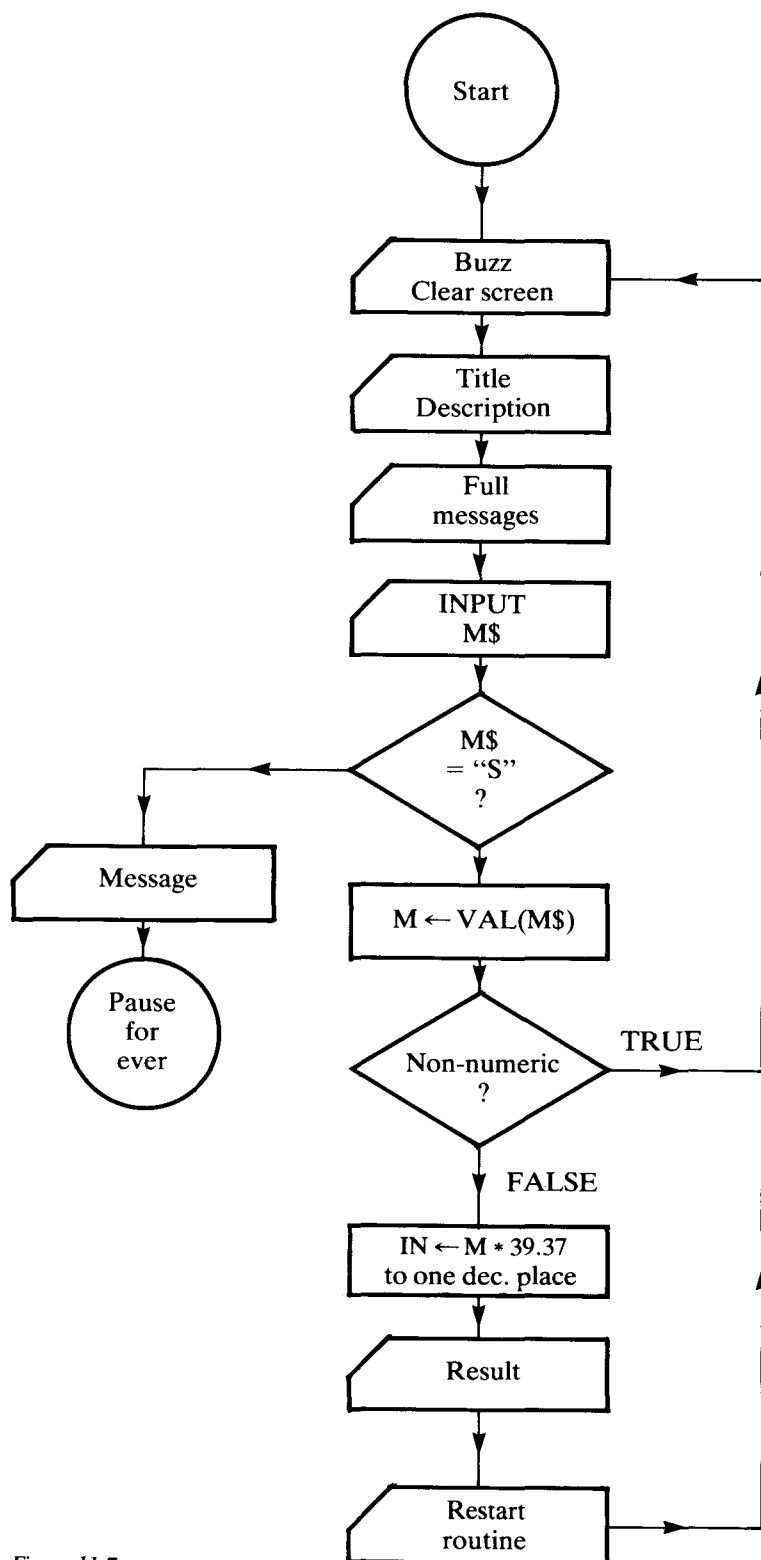


Figure 11.7

DO IT YOURSELF

No, don't worry, I shan't give you many specific exercises on this particular material; all the same I do urge you to develop your expertize in top-down programming as you develop your Atari BASIC coding expertize itself. You may care to do simple flowcharts for some of your existing programs, or for some of those we have looked at in this book. But the important thing, I think, is that you get into the right frame of mind!

1. Study the flowchart in Figure 11.8 and then describe the idea in words.
2. . . . and then, if you are keen, write a program to express it.
3. Prepare a careful outline flowchart of one short program in this book.
4. A week or two later, try to write a program based on that flowchart. Compare the result of the original and decide how good your flowcharting *and* coding are!

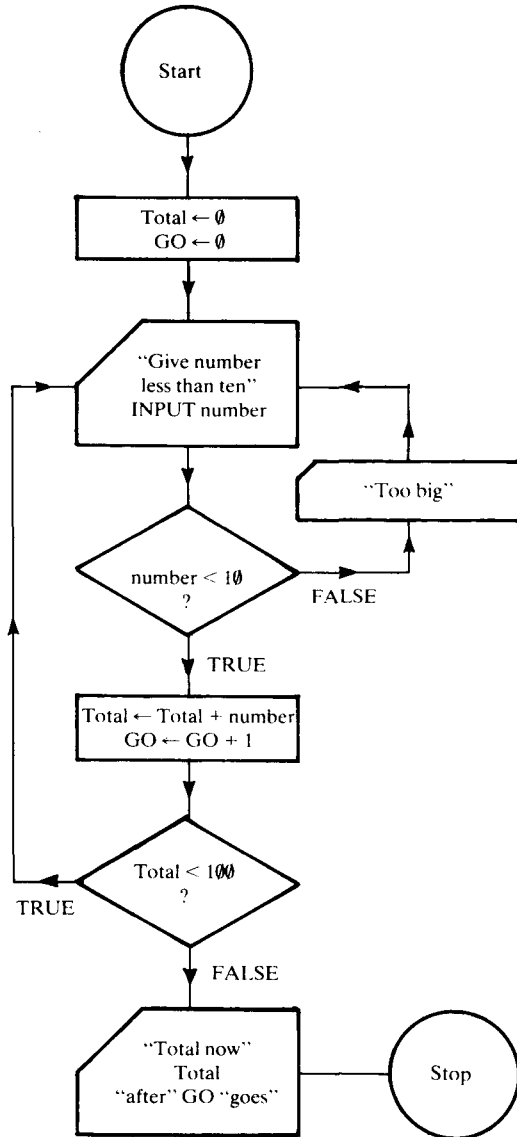


Figure 11.8

12 Yellow Subroutine

Now that our Atari programs can include open decision-making, they can gain a more complex structure. Indeed that's why I took time off just now to look at flowcharts. As you know, because I've used the feature a number of times before, Atari BASIC has a facility for keeping things simple in the more advanced programs we may now be working on. This feature is the *subroutine*. Subroutines are of great value and I'm delighted to be able to use them in the rest of the programs in this book without having to say 'sorry' to you first.

A subroutine is any section of a program ('routine') which has a clear single purpose. I have just explored the background theory in some detail (in the last chapter), so let me just give a summary now.

The most efficient way of developing a program is to break the first concept down into a number of chunks. Once the task is complete we have an 'algorithm', in verbal or graphic form, that describes how to solve the problem faced. We can use the word '*module*' to describe either a written algorithm stage or a flowchart box.

Each one of these modules has one single function; we should therefore be able to code it into a subroutine to go in the program as a whole. Figure 12.1 overleaf shows the sort of thing that can happen. You should be able to see that each box in this flowchart—each module in the program concerned—should form a logical block in the final whole program (routine). Anyway I spent quite a lot of time on this in Chapter 11.

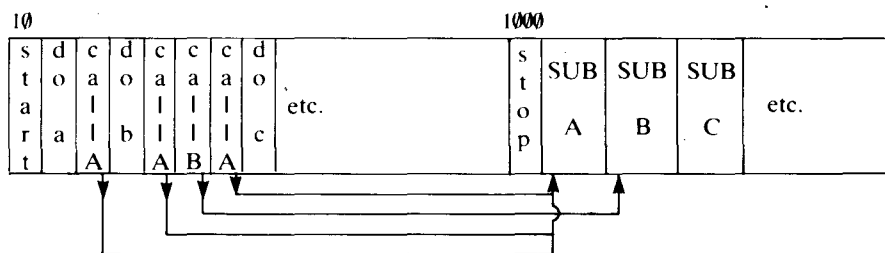


Figure 12.2

The word subroutine applies to a chunk of a complete program with a given job to do that relates to a module in an algorithm. What most folk mean by subroutines are chunks that appear in the listing apart from the main program stream. Figure 12.2 shows this. What the main program then does is to 'call' on a subroutine whenever it is needed. We call such subroutines '*closed*'.

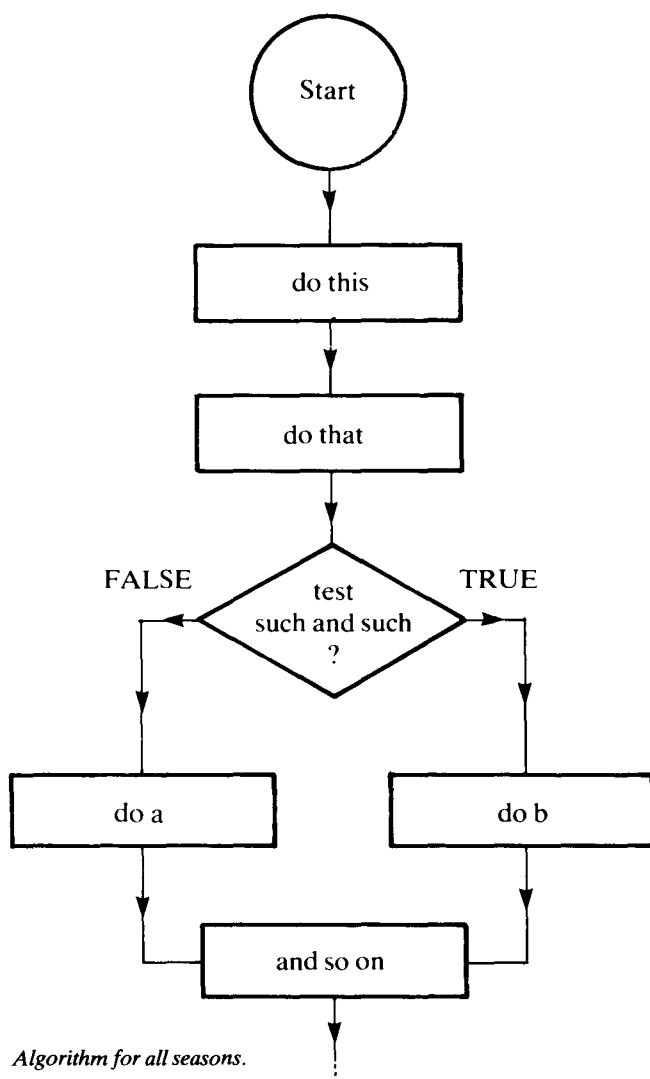


Figure 12.1 Algorithm for all seasons.

Open subroutines, on the other hand, are those which appear within a listing without any clear beginning or end. They still have a single function to do, they still relate to a given algorithm module, and we still code them one at a time. Open subroutines are blocks with a given task in the main program, or indeed within closed subroutines. I think Figure 12.2 shows this as well.

GOSUB...RETURN

The way Atari BASIC deals with closed subroutines is (as far as I know) common to all BASIC dialects. (Having said that, I must note that some other versions of this programming language have posher methods of dealing with closed subroutines as well.) We use two new keywords here. To call up a subroutine, GOSUB *n* is at our service, *n* being the number of the first line of the subroutine. To tell the micro that the subroutine is finished with, use RETURN. Control then goes back to the statement after the one calling with GOSUB.

The approach has two major advantages. The first must be obvious—in a program of much length you are likely to find that the same module has to be used several times (an

example is SUB A in Figure 12.2). Then it need appear only once in the listing, yet you can call it as often as you want. That can lead to massive savings in program storage.

The second advantage of the closed subroutine is less obvious. It is that the effects of the program chunk are particularly easy to check and correct without involving the main program. This can save a huge amount of time and trouble. (This second advantage also applies to work with open subroutines if you are particularly careful with your top-down development.) Figure 12.3 shows more clearly what I mean.

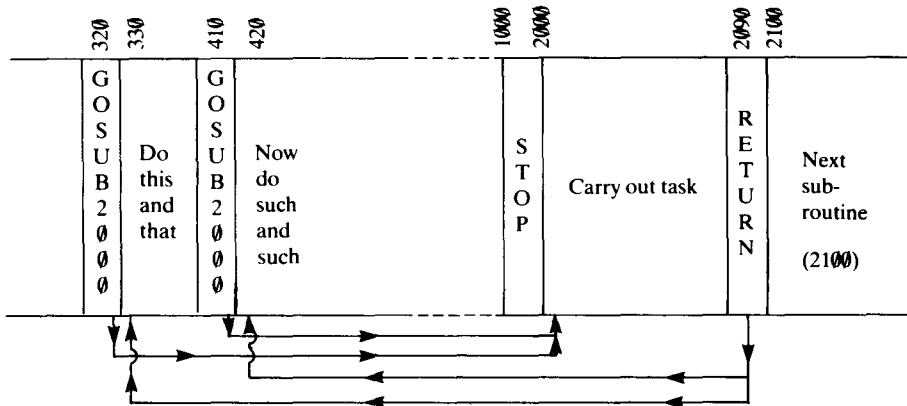


Figure 12.3

When the computer meets GOSUB *n* it stores in memory its current position in the program, and flashes off to line number *n*. When the computer later meets RETURN it fishes round in memory to find where to return to, and back it comes. It's just like using a book mark to keep your place while you are looking at the pictures in the middle.

The section of memory that stores the current position when you dash off to deal with a subroutine (we call it 'the stack') can get pretty busy. This is because you can 'nest' subroutines almost as much as you like—one can call a second, which can call a third, and so on. Just as with nested IF statements, though, things can get tricky in practice—you need to take care.

The closed subroutine is a jolly useful structure. Get used to it, especially—but *not* only—when your programs need the same task carried out time and time again. Program 35 will do for your first example. Study it!

Program 35: Screen-fill

```

9 REM ** Initialisation
10 LET FILL=1000:LET WAIT=1500
11 REM ** Sub-routine addresses
20 SETCOLOR 1,0,14:SETCOLOR 2,4,2:SETCOLOR 4,4,4
30 PRINT CHR$(125)
40 DIM F$(10):DIM N$(10)
50 POKE 82,0
51 REM ** Set left margin at edge
60 POKE 755,1:REM ** Hide cursor
99 REM ** Main program
100 LET D=1:GOSUB WAIT
101 REM ** Set planned delay (D in seconds) before calling sub-routine WAIT
110 POSITION 2,10:PRINT "Please enter one character & RETURN!"
120 FOR GO=1 TO 2 STEP 0:POSITION 3,13:PRINT "          ":POSITION 2,13:INPUT F$:IF LEN(F$)=1 THEN LET GO=3
121 REM ** NB Mug-trap
130 NEXT GO
140 GOSUB FILL
150 LET D=1:GOSUB WAIT
160 PRINT CHR$(125):POSITION 2,10:PRINT "What's your name"
170 FOR GO=1 TO 2 STEP 0:POSITION 19,10:PRINT "          ":POSITION 19,10:INPUT N$:IF LEN(N$)>3 THEN LET GO=3
171 REM ** Similar mug-trap

```



```

180 NEXT G0
190 LET F$=N$:GOSUB FILL:LET D=1:GOSUB WAIT
200 PRINT CHR$(125)
210 POSITION 0,10:PRINT "Now give any set of characters you like!"
220 POSITION 2,14:INPUT F$:GOSUB FILL:LET D=1:GOSUB WAIT
230 FOR B=0 TO 1 STEP 0
240 SETCOLOR 1,0,RND(0)*8:SOUND 1,50,10,10:LET F$="*Thank-you":GOSUB FILL
250 SETCOLOR 2,RND(0)*15,8+RND(0)*8:SOUND 1,40,10,10:LET D=1:GOSUB WAIT
260 NEXT B
261 REM ** Typical flashy ending; BREAK to escape
990 STOP
991 REM ** Used only in development
998 REM ** Sub-routines
999 REM ** FILL
1000 PRINT CHR$(125);CHR$(253)
1010 FOR A=1 TO 960/(LEN(F$)+1):PRINT F$;" ";NEXT A
1020 LET D=2:GOSUB WAIT
1021 REM ** Nested sub-routines
1030 POSITION 0,0
1040 RETURN
1499 REM ** WAIT
1500 FOR W=1 TO 500*D:NEXT W
1510 RETURN
1990 RETURN
1991 REM ** Used only in development

```

Here are some points about the GOSUB...RETURN business that you should note.

1. You can use structures like this in Atari BASIC (but not in all others!):

```

PRINT "Please type 1,2,3 or 4 and press RETURN."
INPUT CHOICE
GOSUB CHOICE * 1000

```

This would need suitable subroutines starting at lines 1000, 2000, 3000 and 4000. Each ends, of course, with RETURN. This lets your program go darting off in any one of the four directions, depending on what the user chooses.

2. The keyword ON can give the same effect. Here's the corresponding chunk of code:

```

PRINT "Please type 1,2,3 or 4 and press RETURN."
INPUT CHOICE
ON CHOICE GOSUB 1000, 2000, 3000, 4000

```

This approach is of value only when you find the maths too tough to let you use the one described in the previous paragraph. That really implies that you didn't structure the program properly before (with a memory map . . .)! Otherwise there's little use for ON, for as you see it takes up more space. Let me note at this point, however, that you can also use ON with GO TO in just the same kind of way.

3. Both the previous ideas use a kind of implied GO TO. I mean we have to use line numbers. After all my comments about using line numbers within programs, you'll be pleased to know that the Atari does have a way of getting round this, though I admit it's cheating somewhat. You can 'name' your subroutines. Here's how we could do it with the same chunk of code as before:

```

LET NORTH = 1000: LET EAST = 2000: LET SOUTH = 3000:
LET WEST = 4000 . . .
PRINT "Please type 1(N), 2(E), 3(S), or 4(W) and press RETURN."
INPUT CHOICE

```

ON CHOICE GOSUB NORTH, EAST, SOUTH, WEST

999 REM ** CHOICE: NORTH etc

If you need to renumber your program later, it may be that your subroutine starting addresses must change. It is now easier to change them in a line near the start of the program than before. This approach thus lets us program in a more structured way.

4. You can enter a subroutine in the middle if you have the need. Follow this fragment through.

```

20 PRINT "Give a number between 1 & 4.": INPUT NUMBER
25 IF NUMBER < > INT(NUMBER) THEN GOSUB 100
30 IF NUMBER = INT(NUMBER) THEN GOSUB 140
...
100 PRINT "I can work only with whole numbers!"
120 LET NUMBER = INT(NUMBER + 0.5)
130 PRINT "I shall take it as "; NUMBER; ". "
140 etc.
```

Did you follow? The subroutine *really* starts at 100, but the first lines deal specially with cases where a user didn't enter a whole number. If he or she *does* enter a whole number we don't want to use those lines, so we can GOSUB straight to line 140. This is a useful trick.

5. Another useful trick is to have several exits from (several RETURNS in) a subroutine. But *don't* jump out of a subroutine with GO TO—that will clog up the memory. (If you *do* do this, which you shouldn't, you are lucky—Atari BASIC is fairly unique in having the keyword POP to help you out of a mess. If you pop back to somewhere in your program listing using GO TO, *which you shouldn't*, make POP the first statement the program meets. What it does is to clear from the stack the address the subroutine should have RETURNed to. But again I say, very strongly, *you should not use GO TO to escape from a subroutine.*)
6. Keep your closed subroutines well away from the main program. The best plan is to put them after line 5000 (for instance) with STOP at line 4990 to prevent accidental entry. When you've finished writing your program you should be able to take that STOP out of the listing. This is because a really user-friendly program should never come to a real end. (See Program 30 for a better way!) If you do find finally that your program *does* need a barrier between the main stretch and the subroutines, replace that STOP with END. END causes the program to stop working but does it more pleasantly for the user.
7. The higher a subroutine's starting line number, the longer it takes for the routine to be found. This is because when the micro's told to GOSUB n, it starts at the beginning of the program in its search for the line with that number. If you want highest speed, you can do this instead:

```

10 GO TO 1000
20-990 SUBROUTINES, most often used ones first
1000 START OF MAIN PROGRAM
```

Slightly more ugly ways of doing the same thing involve asking the user to start the program going with GO TO 1000 (instead of RUN) or using the START function key (Page 172).

8. Some folk suggest that the first line of a closed subroutine should always be REM. This is because in some versions of top-down development it is common practice to code the main program first without working out the details of the subroutines.

Then a REM after a line number will tell you that the line is the beginning of a routine (especially if you put more details after that keyword). RETURN a few lines later is still of course needed.

Myself I don't do that, partly because I don't top-down develop in the same kind of way, and partly because I often remove REMs from final versions of my programs. (This is to make them load and run faster.) As the Atari gives you an ERROR message if you use GOSUB n (or GO TO n) where n does not exist, this use of REMs is unsafe. Personally I find it better to put the REM statement that describes a given closed subroutine at a line just before it starts.

OK? Let me summarize what we've had so far.

1. A subroutine is a section of program with a single function.
2. A closed subroutine is one kept apart from the main program, called when needed by GOSUB n and closed by RETURN.
3. The main use of a closed subroutine is to carry out repeated tasks without having to repeat the program lines.
4. A major subsidiary benefit is that you can GOSUB n as a command in order to test the subroutine's workings as a single unit.

I suggest you keep your eyes open for the use of GOSUB...RETURN in programs in this book and elsewhere. Each time you come across the feature study why it is used, and how. But I shall give you some development work to do of your own on the subject in a moment!

SUMMARY

In this chapter we have looked at the Atari's handling of closed subroutines. These are the keywords met:

END	POP
GOSUB (GOS.)	RETURN (RET.)
ON	STOP

I've also made some points about the use of closed subroutines in top-down program design and structured coding.

DO IT YOURSELF

1. Write a BASIC program using GOSUB...RETURN to fill the screen with the user's name in the colour of his/her choice on a background of chosen colour. Program 35 will give you a start if you need.
2. Select one or more of the programs in this book which use closed subroutines. Make sure you know how or why they are used (as always), and draw a memory map with control flow lines (like Figure 12.3 on Page 123) for one of the simpler ones.
3. You can animate pictures like this.

Program 36: Robot

```
9 REM ** Initialisation
10 LET X=0:LET Y=0:LET DRAW=1000:LET UNDRAW=2000
11 REM ** See naming of sub-routines here
20 POKE 755,0
21 REM ** Stop cursing
30 SETCOLOR 1,0,0:SETCOLOR 2,12,14:SETCOLOR 4,11,12
40 PRINT CHR$(125)
99 REM ** MAIN PROGRAM
100 FOR X=0 TO 20
110 SOUND 1,255,0,10
120 GOSUB DRAW
130 SOUND 1,0,0,0
```

```

140 FOR W=1 TO 25:NEXT W
150 GOSUB UNDRAW
160 NEXT X
170 GOSUB DRAW
180 SOUND 1,10,10,14
190 FOR W=1 TO 500:NEXT W
200 FOR Y=0 TO 18
210 SOUND 1,150,4,10
220 GOSUB DRAW
230 SOUND 1,0,0,0
240 FOR W=1 TO 20:NEXT W
250 GOSUB UNDRAW
260 NEXT Y
270 GOSUB DRAW
280 SOUND 1,80,10,14
290 FOR W=1 TO 500:NEXT W
300 FOR FLASH=0 TO 1 STEP 0
310 SOUND 1,80,10,14:POSITION X+2,Y:PRINT "Hallo!"
320 FOR W=1 TO 100:NEXT W
330 SOUND 1,0,0,0:POSITION X+2,Y:PRINT "Hallo!"
340 FOR W=1 TO 100:NEXT W
350 NEXT FLASH
990 STOP
991 REM ** Needed only in development
998 REM ** SUB-ROUTINES
999 REM ** DRAW sub-routine
1000 POSITION X,Y:PRINT "0"
1010 POSITION X,Y+1:PRINT "1":REM * Shift & =
1020 POSITION X,Y+2:PRINT "2"
1030 POSITION X,Y+3:PRINT "/*"
1040 RETURN
1999 REM ** UNDRAW sub-routine
2000 POSITION X,Y:PRINT " "
2010 POSITION X,Y+1:PRINT " "
2020 POSITION X,Y+2:PRINT " "
2030 POSITION X,Y+3:PRINT " "
2040 RETURN

```

Try this simple example and make up more complex animated routines of your own in the same way.

13 Numbering Our Days

If you think about it for a moment, you will realize that the word ‘computer’ implies something for doing mathematical calculations. My dictionary says that ‘compute’ means ‘to calculate: to number: to estimate’. It says that a ‘computer’ is ‘a calculator: a machine or apparatus, mechanical, electric or electronic, for carrying out especially complex calculations, dealing with numerical data or with stored items of other information’. Perhaps the definition I gave you in Chapter 2 is a bit clearer?

The first computers in the sense we use the word were certainly just programmable electronic calculators. (And they weren’t as good at the job, in *many* ways, as the little pocket machines we can now buy for a couple of pounds.)

When I defined the computer in Chapter 2, I spoke of it as a data processor. In fact whatever kind of data the computer processes, it all has to be represented in number form (what we call binary numbers in fact), and all the processing consists of arithmetical processes. Indeed I called the central processor the Arithmetic and Logic Unit (ALU) in that chapter.

OK—that’s not very relevant to us as users or as new programmers. But it is time to find out a bit more about arithmetic in Atari BASIC. We’ve met quite a lot already, and I hope it didn’t worry you. Anyway, let’s look at this little program. Enter, use, consider.

Program 37: Four-function calculator

```
10 GRAPHICS 2:SETCOLOR 4,0,0:SETCOLOR 0,0,0:DIM CONT$(1)
11 REM ** Mode 2 for a change: math's can be colourful!
20 FOR GO=0 TO 1 STEP 0:PRINT CHR$(253)
30 FOR L=0 TO 9:POSITION 0,L:PRINT #6;" " " :NEXT L
31 REM ** Easiest way to clear Mode 2 screen!
40 POSITION 2,1:PRINT #6;"PLEASE ENTER TWO NUMBERS!"
50 POSITION 2,4:PRINT #6;"return AFTER EACH."
60 PRINT :INPUT N1,N2
61 REM ** Must be Mode 0 for INPUT, hence use of Mode 2 not 18
70 FOR L=0 TO 9:POSITION 0,L:PRINT #6;" " " :NEXT L
80 POSITION 0,0:PRINT #6;" ";N1," ";N2
90 POSITION 0,1:PRINT #6;"sum",N1+N2
100 POSITION 0,3:PRINT #6;"diff.",N1-N2
110 POSITION 0,5:PRINT #6;"product",N1*N2
120 POSITION 0,7:PRINT #6;"quotient",N1/N2
130 PRINT :PRINT :PRINT "RETURN to go on...."
131 REM ** Note care in formatting text window too!
140 PRINT :PRINT :INPUT CONT$
150 PRINT CHR$(125):NEXT GO
```

Note here, apart from the use of text in Mode 2 (and text-window), two uses of INPUT met before only a couple of times. One appears in line 140—the use of INPUT to handle the ‘Press RETURN to go on.’ feature.

Then in line 60, we have a double input. The message asks the user for two numbers with (R) after each. In fact, as you may recall, it would also accept two numbers with a comma between them.

ARITHMETIC

Anyway—the arithmetic. Program 37 showed simply how BASICS deal with the four main arithmetic functions ('operations'): add, take away, times, and divide. Note again the special symbols: '*' for times, '/' for divide.

The Atari can do arithmetic with numbers between about -10^{96} and $+10^{98}$ ($\pm 2^{325}$). The smallest numbers it can deal with are $\pm 10^{-97}$. Numbers are handled with an accuracy of at least nine significant figures. If all that means anything to you—you'll realize that the Ataris can do pretty tough arithmetic in astronomy and nuclear physics, for instance. Well, maybe one of those is your interest. All I really want to note here is that you can do fair value calculator work with your computer.

We have also met a fifth important arithmetic operator, whose symbol is \wedge . The posh name for the process this controls is—wait for it—'exponentiation'. I call it 'raise to a power', but that's a mouthful too. 3 raised to the power 4 (written 3^4) is $3 \times 3 \times 3 \times 3 (= 81)$. We get it on screen with `PRINT 3^4`. Add this to the above program to let it also use the power of power-raising:

```
125 POS. 0,9: PR.# 6; "power", N1^N2
```

Again, try it.

I hope when you were running this program, you tested it fully. What happens when you input decimal numbers, negative numbers, zero, and so on? How does the micro react at line 120 if N2 is 0? Computers can't divide by zero—do you know why not? (Can you work out how to use TRAP to stop the machine crashing if N2 is zero?) And—what would happen if we had more complex sums ('expressions') to work out than those in Program 37?

Think, for instance, about this simple little problem . . .

*Two people each have three ball-points and two felt-tips.
How many pens have they in total?*

OK, I know you can work it out in your head. But how do you do it? And, more to the point, how do you tell your Atari to do it? Does *this* give the right answer (try it)!

```
PRINT 2 * 3 + 2
```

I hope you tried it. I hope you got the 'wrong' answer. But computers can't make mistakes, can they? What happened to the missing two pens? Try *this* then:

```
PRINT 3 + 2 * 2
```

After all, that's how you'd do it in your head, and it's how you would do it on most hand calculators. Wrong answer still! A *different* wrong answer.

So, there's a problem. It's not the computer's fault. The thing is that the machine is trained always to multiply before it adds. So it goes

$2 * 3 + 2 = 6 + 2 = 8$ in the first case

and

$3 + 2 * 2 = 3 + 4 = 7$ in the second.

The posh term for this is priority. Multiply has a higher priority than add. If you want to break priority, as in fact we do here, you have to use brackets (). (Note that the Atari has square brackets too, but they're for something else, and don't work here. Find them, if you want, on the comma and full-stop keys.) So, to solve the case of the missing pens, enter

```
PRINT 2 * (3 + 2)
```

and the computer goes

$$2 * (3 + 2) = 2 * 5 = 10.$$

Or you can use

```
PRINT (3 + 2) * 2
```

to get the same result.

IF YOU DON'T PUT BRACKETS WHERE THEY'RE NEEDED, YOU'LL GET THE WRONG ANSWER. . . . On the other hand, if you use them where they're *not* needed, you get the right answer. So—if in doubt, put the brackets in. Here is the Atari's priority list as we need to know it so far.

Highest	(...)	brackets
	INT	and other functions
	^	raise to a power
	*, /	times, divide
Lowest	+, -	add, subtract

There are others in the full list, but we needn't worry about them yet.

Now, if you are not what you'd call mathematically inclined, you are not likely to want to do too much in the line of sophisticated number-crunching programming. You have my permission to skip the rest of this chapter, then—but please glance through the projects at the end. One day you'll want to come back, I hope. But first. . . .

An important note

Computers can't always do arithmetic with perfect accuracy, any more than humans can. There are many cases in real life where *we* have to round numbers off for our convenience, as when working out the cost per gram of some supermarket item priced at 32p for 5¼ oz! Inaccuracies in computer arithmetic often surprise people because they don't expect them, as when, for instance, one finds a calculator gives 9/3 as 2.9999. (Early Ataris are as bad as this sometimes!)

The reason for the problem is our good old friend, binary arithmetic. Numbers that are held exactly in decimal (human form) are not necessarily held exactly in binary (the computer works in binary), so the machine has to approximate them. I'm not going to go into this matter in depth here, but do not be surprised if you sometimes find apparently strange errors in what seem to be simple calculations. The cause is not a bug in the computer chip, but a problem of binary arithmetic.

Most people new to computing find this problem for themselves—and then spend a great deal of time trying to explore and explain it. Many even write to the computer press, or even (!) phone me up. So, to avoid all that hassle—read this important note again!

NUMBER-CRUNCHING

Right—if you're still here let's go on. What I want to do in this section is to develop a number of important 'number-crunching' ideas through a short series of (I hope) useful programs. If you find it unnecessarily hard going at any point, do feel free to start skipping through until you meet the section on 'functions'—then try to get with us again.

The first of these programs, Program 38, is designed to produce multiplication tables on demand. It is not specially novel—indeed, I think that everyone tries to do something like this fairly early on when learning BASIC programming! I use it here to show you how the various BASIC structures we have already met can be applied to computation.

Program 38: Tables

```
10 GRAPHICS 1:SETCOLOR 0,0,0:SETCOLOR 4,9,4
20 FOR GO=0 TO 1 STEP 0
30 FOR L=0 TO 19:POSITION 0,L:PRINT #6;"          ":NEXT L
31 REM ** Clear screen as in 37
40 POSITION 3,2:PRINT #6;"MULTIPLICATION":PRINT #6;" ====="
50 POSITION 2,10:PRINT #6;"please enter table number & RETURN!"
60 PRINT :PRINT :PRINT :INPUT T
70 FOR W=1 TO 500:NEXT W
80 FOR L=0 TO 19:POSITION 0,L:PRINT #6;"          ":NEXT L
90 FOR L=1 TO 10
100 POSITION 0,L*2:IF L<10 THEN PRINT #6;" ";
110 PRINT #6;L;" times ";T;" is ";L*T
120 NEXT L
130 FOR W=1 TO 500:NEXT W
140 PRINT :PRINT :PRINT :PRINT "Again (0:no, 1:yes)";:INPUT CONT:IF CONT THEN NE
XT GO
141 REM ** Logic!
150 FOR L=0 TO 19:POSITION 0,L:PRINT #6;"          ":NEXT L
160 PRINT :PRINT :PRINT :PRINT
170 FOR B=0 TO 1 STEP 0
180 POSITION 6,11:PRINT #6;"bye-bye!":FOR W=1 TO 150:NEXT W
190 POSITION 6,11:PRINT #6;"          ":FOR W=1 TO 150:NEXT W
200 NEXT B
```

Note, as usual, a number of display lay-out tricks—such as in lines 40, 60, 100, 110, and 140. That last line also contains a touch of the logics—see Page 109. *And* the same line gives you a look at a fairly common yes/no response routine. It's of value if you are short of memory, have a numeric input check subroutine you can already use (I didn't), and like the logical condition that it involves.

The next program is a bit more like number-crunching! Again it isn't very novel—being once more the sort of thing that many people take on fairly early in their programming development. It is important particularly for its use of the numeric function INT, one we have already used a number of times.

Program 39: Factor

```
10 DIM TGT$(10):DIM CONT$(1):SETCOLOR 1,0,0:SETCOLOR 2,4,10:SETCOLOR 4,4,6
20 FOR GO=1 TO 2 STEP 0
30 PRINT CHR$(125)
40 POSITION 5,2:PRINT "* F A C T O R I S A T I O N *":PRINT " _____"
50 FOR T=0 TO 1 STEP 0:POSITION 21,12:PRINT "          ":POSITION 2,12:PRINT "What
number, please";:INPUT TGT$:IF LEN(TGT$)>0 THEN LET I=2
51 REM ** Mug-trapped numeric input this time
60 NEXT I
70 LET TGT=VAL(TGT$):REM ** Recall VAL??
80 PRINT CHR$(125):POSITION 2,2:PRINT "The factors of ";TGT;" are:"
90 FOR F=1 TO TGT/2:IF TGT=F*INT(TGT/F) THEN PRINT F,
100 NEXT F:PRINT TGT
110 FOR W=1 TO 2000:NEXT W:PRINT :PRINT "RETURN";:INPUT CONT$:NEXT GO
```

What this program does is to find the 'factors' of an input number. That means those numbers which divide into the 'target' without leaving a remainder.

The crucial lines here are just the loop of lines 90/100 in which each number is tested in turn to see if it's a factor of TARGET. Do you understand how this test works? Can you also explain the limit of the loop (TARGET/2)? Check too, as ever, the formatting tricks used at various points of the listing, and the brief 'restart' routine in line 110. (It's brief because it may well be that a screen is full of numbers that the user wishes to write down.)

The third program in this little suite is even more formidable mathematically. But I guess you may well have come across the concept at some stage or other. It deals with

things called quadratic equations. These are expressions like $ax^2 + bx + c = 0$. Here a , b , and c are numbers (called 'coefficients') and x is a variable. For any set of values of a , b , and c there can be no more than two values of x that fit the equation. Those two values we call the 'roots' of x . In algebra classes at school we find them like this:

$$\text{Root 1} \quad x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

$$\text{Root 2} \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Now there's a couple of nice numeric expressions to get our teeth into! And here is Program 40 to attempt the task. It has to display the roots of x for any input set of a , b , and c . That is, if those roots exist—they don't always!

Program 40: In the quad

```

10 PRINT CHR$(125)
20 POSITION 10,2:PRINT "Q U A D R A T I C":PRINT "          ====="
30 POSITION 2,8:PRINT "To find the roots of":PRINT :PRINT "a(x*x)+bx+c=0"
40 POSITION 2,14:PRINT "PLEASE ENTER COEFFICIENTS..."
50 PRINT :PRINT "a = ":INPUT A
60 PRINT :PRINT "b = ":INPUT B
70 PRINT :PRINT "c = ":INPUT C
80 FOR W=1 TO 500:NEXT W:PRINT CHR$(125)
90 PRINT :PRINT "The expression is:":PRINT :PRINT "A;"(x*x) + "B;"x + "C;" = 0
100 POSITION 6,9:IF B^2<4*A*C THEN PRINT "No real roots!":FOR W=1 TO 1000:NEXT W
:RUN
110 LET X1=INT(((B+SQR(B^2-4*A*C))/(2*A))*10000+0.5)/10000
120 LET X2=INT(((B-SQR(B^2-4*A*C))/(2*A))*10000+0.5)/10000
130 PRINT "The roots are:":PRINT :PRINT ,X1,X2
140 FOR W=1 TO 2000:NEXT W:RUN

```

The important lines in this context are 110 and 120 (which is nice to edit from it). If you can follow them, you have a good grasp of the Atari priorities list. These lines also use the function INT to round off to four decimal places (see Page 133 and the next section), and the function SQR, used to find the square root of the number after it.

The square root of a number is that number which when multiplied by itself brings us back to the first number. The square root of 9 is 3, because 3×3 (3^2) = 9. The square root of a negative number has no real meaning, so cannot be worked out by the micro. Line 100 has the function of trapping the possible error and ERROR message that result in that case. Of course I could have used TRAP instead, but as I have said before, the danger with TRAP is that *any* error leads to the special action you work out.

Two numeric functions in one line! It's really time we looked at functions with care.

FUNCTIONS

Your Atari offers various 'numeric functions', which I shall discuss briefly in this Section. A function is something that operates on a number—a numeric constant, a numeric variable, or a numeric expression—to produce a new numeric result. The number (or whatever) that the function acts on is called its 'argument'. Comes from the Latin.

BASIC also offers a number of functions which concern strings. I'll deal with those in Chapter 14 rather than here.

So—let's get on with the list of numeric functions; it's in alpha order, and I shall treat each one the same, whether we've met it before or not.

ABS This function gives the 'absolute value' of the argument. In other words, it ignores any sign. Thus $ABS(-4)$ gives 4, as does $ABS(4)$ and $ABS(+4)$. We use it, for instance, to convert what might be a positive or negative number to one that is definitely positive. Examples are:

```
LET ERROR = ABS (RESPONSE - ANSWER)
IF ABS (GUESS - TARGET) ≤ 5 THEN PRINT "Not bad!"
```

ATN This gives the value of the angle, with the unit called radian, whose tangent is the argument. Thus $PRINT ATN(5)$ gives an answer of 1.373.... This means that the angle whose tangent is 5 is 1.373... radians. If, like me, you are more used to dealing with the degree as the unit of angle rather than the radian, enter **DEG** as a direct command, or a statement early in the program concerned. **DEG** tells the Atari to come down to our level and work as if in degrees all the time. **RAD** gets you back to the radian approach, as do **RUN**, **NEW**, and **RESET**.

CLOG This does not have the function of making your micro act like a dancer from the North of England as you might expect. In fact it returns a value which is the 'common logarithm' of the argument. Not so long ago everyone had to learn in great detail about how to use tables of 'logarithms', but it's a dying art now. All the same, logarithms are of value sometimes in scientific work.

$PRINT CLOG(2)$ gives 0.3010...

Common logarithms (sometimes called Brigg's logarithms) are to base 10. Take a look at **LOG** if you are into 'natural logarithms'.

COS This gives the cosine of the argument, assuming it to be an angle in radians. (Unless you've used **DEG** before.) Thus $COS(0.5)$ gives 0.8775..., unless you did the **DEG** bit, in which case you get 0.9999....

EXP **EXP** returns the value of the number 2.71828... (called 'e' for short), raised to the power of the argument. Should you care about such things. You can have values of the **EXP** argument within the range -225 to +225. That's a pretty big range!

INT **INT** gives the whole number ('integer') to the left of its argument on the 'number line'. Try the following direct commands to test what it does:

? $INT(X)$, where X is each one of a set like 1.2, 1.25, 1.8, 0.1, 0, and similar negative numbers.

I hope you can see that $INT(X)$ does indeed give us the whole number left of X on the number lines. See the sketch in Figure 13.1.

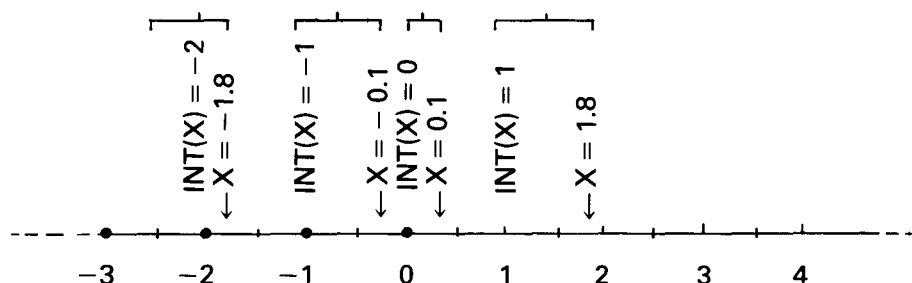


Figure 13.1 *INT on the line.*

We can often use INT just like that. Indeed I have done so a number of times in this book. However there are two particularly useful INT structures that you should get to know. I've used them both, but here we'll summarize.

1. We can use INT to convert a mixed ('decimal') number to the nearest whole number rather than to the one on the left of it. Thus 1.2 should give 1 and 1.8 should give 2. Here's the structure:

LET X = INT (X + 0.5)

Try it on a set of numbers as above, then refer to the number line sketch so you can see how it works. This feature is called 'rounding', rounding to the nearest whole number.

2. We can also use INT to round a mixed number to a value correct to a given number of decimal places. To two decimal places, for instance (as when dealing with decimal money) we want 1.23 from anything between 1.225 and 1.234999.... Here's *this* structure:

LET X = INT (X * 100 + 0.5)/100

Test it out fully as usual. Adapt it to give a different number of decimal places.

LOG Many micros use LN for this (as do many people!). LOG returns the 'natural logarithm' (logarithm to base 'e') of the argument. The argument must be positive.

SGN The value this returns tells you whether the argument is positive (giving +1), negative (giving -1), or zero (giving 0). Thus SGN(-3) gives -1; SGN (3) and SGN (+3) give 1. A use of this is in:

IF SGN(SCORE) < 1 THEN LET SCORE = 0

SIN This returns the sine of its argument, assuming that to be an angle in radians (unless you used DEG before). In DEG-mode my Atari gives SIN (30) as 0.499999999 rather than 0.5. This is in fact a rounding error (see important note on Page 130). Older Ataris have this problem more often.

SQR As you found out in the last section, SQR gives the square root of its argument—the number which, when multiplied by itself, brings you back to the argument. (Thus 4 is the square root of 16, as $16 = 4 \times 4$.) As I said before, the argument of SQR must not be negative—or, if it is, you get an ERROR. Note that SQR (X) works more quickly than $X \wedge 0.5$. (In the same kind of way, $X * X * X$ is faster than $X \wedge 3$, and the same applies to squaring.)

And that's a full list of the Atari numeric functions—those keywords which, acting on a numeric argument held in brackets after them, return a numeric value. I must be honest and admit that this is a fairly basic list of functions—many micros now offer quite a lot more. All the same, with this basic list you can obtain many others by combining them in suitable ways. For instance, if you want the tangent of an angle X, use SIN(X)/COS(X). That's an easy one—there are more complex ones in Appendix 7, if this kind of thing is your scene.

Really I suppose I ought to have included FRE and RND in the above list. They're not quite true numeric functions in that their arguments are 'dummy' ones—any value will do, and you get the same result. Even so, this is a sensible place to note them again, though of course I've discussed both in the past.

What I mean by a dummy argument is that both these functions need to have a number in brackets after them, but it doesn't matter what it is. I suggest you always put zero as the argument of FRE and RND.

FRE This tells you the number of bytes so far unused by your program and data in the computer. We'd normally use this function as a direct command: ? FRE(0), giving the

amount of memory available (free). However, you can use the line

```
IF FRE(0) < 100 THEN PRINT "watch out—you're running out of memory!"
```

RND This gives a (pseudo-)random number in the range 0 to 0.999999999. I've used RND quite often before, and there's a lot about it on Page 79.

I'll close this section with a little program to give you some practice with numeric functions, in this case angular ones. Program 41, *Angle*, displays on screen a table of values of angles (in degrees and in radians) and their sines, cosines, and tangents. At the start of the program, the user has to give the value for the first angle in the list (assuming degrees). The table gives twenty lines of data, starting with that value, and increasing the angle by 1 (degree) each time.

Program 41: Angle

```
10 DEG :PRINT CHR$(125);"ANGLES":POSITION 2,9:PRINT "Please give starting value."
20 INPUT S
30 PRINT CHR$(125);"degANGLE rad Sine","Cosine","Tangent":PRINT " _____"
40 FOR L=2 TO 21:LET K=S+L-2
50 POSITION 2,L:PRINT K
60 POSITION 8,L:PRINT INT(K*3.14159/180+1000+0.5)/1000:REM ** Converts to radian
  s to 3 significant figures
70 POSITION 14,L:PRINT INT(SIN(K)*1000+0.5)/1000
80 POSITION 22,L:PRINT INT(COS(K)*1000+0.5)/1000
90 POSITION 32,L:PRINT INT((SIN(K)/COS(K))*1000+0.5)/1000
100 NEXT L
110 PRINT " _____"
120 POKE 755,1:FOR W=0 TO 1 STEP 0:NEXT W
```

SUMMARY

In this chapter I have taken a fairly hard look at arithmetical (numerical) data processing with the Atari. I hope you had a hard look too.

We have met the arithmetical operators (+, −, *, /,) and the order of precedence (priority) in which the micro deals with them—and with material in brackets and with functions. You now also have some notes on the Atari functions and their use.

The keywords discussed in this chapter are as follows:

ABS	LOG
ATN	RAD
CLOG	RND
COS	SGN
DEG	SIN
EXP	SQR
FRE	VAL
INT	

DO IT YOURSELF

1. Produce a program giving angles in degrees from input values of trigonometrical ratios. (Use the inverse trigonometrical function ATN, and refer to Appendix 7 for ASN, and ACS.)
2. Turn Program 38 (Tables) into a good game.
3. Follow a study of Program 39 (Factor) with one of your own to list the prime numbers up to some value you set. Prime numbers are ones with no factors apart from 1 and themselves. This program, if you succeed, will take a long time to run, so make sure you have a kettle to hand.

4. If you ignore air friction, you can say that an object falling towards Earth moves faster and faster. If it starts from rest, after T seconds its speed will be $10 \times T$ metres per second, it will have fallen $5 \times T^2$ metres. Develop an Atari program to accept time of fall (T) and to display speed and distance with suitable messages.
5. Develop a felt-tipped pen program to solve questions like the one we looked at on Page 129.
6. If you *did* Program 40, this one should be simple! Write a program to find y from $ay = bx + c$, with a given input set of coefficients a, b, c , and various input values of x . (The output data could be of use for plotting graphs and work with co-ordinates.)
7. Write a program to accept a set of numbers, display them in order and show the average (mean). If you know about such things and they are of interest in your work, extend the program to output mean, standard deviation, median and mode.
8. Devise a BASIC program to ask for and accept daily rainfall data and to display the highest, lowest, total and average values for a month.
9. To how many decimal places can you make your Atari calculate the value of 'pi'?

14 Heart Strings

I suppose it could be—though I hope it's *not* the case—that you may be feeling that what I've done so far is all somewhat mathematical. I'm afraid that if you *do* think that, there's not much anyone can do about it—computer programming *is* rather mathematical in concept. By that I mean that is has to use numbers and relationships, and has to involve fairly careful logical thinking.

The following direct command (direct so it doesn't need a 'mathematical' line number!), I agree, may not appear to be mathematical:

```
PRINT "Ataris make starry eyes."
```

All the same, that really *is* mathematical in that:

1. The computer must change each character you enter at the keyboard into a number.
2. It groups the numbers that result following certain rules—mathematical ones if you like.
3. Those numbers move around in the micro following more such rules.
4. Mathematics, in a sense, is the process involved when the command is carried out, or when it is rejected because of an error.

The Arithmetic and Logic Unit (Chapter 2) *is* a mass of arithmetical circuits as far as its action is concerned.

All the same, we don't need to think about all that, and in this chapter I'm going to tell you about *strings*—and we don't need to be too mathematical here. Logical, though, yes.

WHAT IS A STRING?

As you picked up long ago, a string is *any* set of characters. The characters in it can be alphabetical letters, numbers, punctuation marks, symbols—anything you can get from the keyboard. Thus "Ataris give you starry eyes" is a string. So is "I have an elephantine memory.", and so are " $3 \times 6 = 18$ ", "the year is/was 1984.", "What is your name?", "+, (-./G6a*Z", "24", and "□". All these are what we call *string constants*—that means they don't change. We enclose string constants in speech marks ("...") in writing and when dealing with the computer. There is an interesting exception to the last point—string constants *must* not have quotes when held in DATA statements. I'll come back to that later, when I deal with DATA statements. (Also, as you know, you don't *need* quotes when entering a string as an input, but we could argue over whether this is a constant or not.)

This speech mark rule means that *strings cannot contain speech marks*. Try this, for instance:

```
PRINT "Ataris make "starry" eyes."
```

The micro rejects it with one of its sweet little ERROR messages. Do you see why?

If you need to use quotes in a string, the easiest thing to do is to use the apostrophe:

```
PRINT "Ataris make 'starry' eyes."
```

That's OK, isn't it? (There are in fact complex ways to get double speech marks into a string constant, but why bother?)

As well as having numeric constants (like 4), we have numeric variables (like NUMBER, where NUMBER could be *any* number). In the same way, again as you already know, we can have *string variables*. Here's one—NAME\$ (say it 'name string'). The dollar symbol after a variable name shows that it belongs to a string variable. We've seen how to use this in earlier programs. It's like this:

```
10 DIM N1$(10):DIM N2$(10)
100 FOR GO=0 TO 1 STEP 0:POSITION 0,2:PRINT "What is your name, Player 1
   ":POSITION 27,2:INPUT N1$:IF LEN(N1$)>3 THEN LET GO=2
110 POSITION 0,2:PRINT "                                     ":NEXT GO
120 POSITION 0,2:PRINT "Thank-you, ";N1$;"!"
130 FOR GO=0 TO 1 STEP 0:POSITION 0,8:PRINT "And who are you, Player 2
   ":POSITION 25,8:INPUT N2$:IF LEN(N2$)>3 THEN LET GO=2
140 POSITION 0,8:PRINT "                                     ":NEXT GO
150 POSITION 0,8:PRINT "Thank-you, ";N2$;"!"
160 FOR W=1 TO 500:NEXT W
170 POSITION 0,18:PRINT "Right, ";N1$;" and ";N2$;"....":PRINT "Are you ready?"
```

Each time you run this program, the values of N1\$ and N2\$ are likely to change—that's what we mean by variables.

Line 10 in the above fragment reminds you of the need to use DIM (or COM, exactly the same) to reserve space in memory for the strings set in your program. Perhaps this is the place to say a little bit about DIM, or, as I say, COM.

A string variable can be DIMensioned to any length you like, up to the maximum available (a few thousand bytes). Sometimes there is a need for such long strings in fact. Think of using your Atari as a text-editor (word-processor): all the text would be within one variable. Once you have used DIM for a given variable name, any attempt to use it again for that name is doomed to ERROR. The keyword CLR will allow you to re-DIMension a string variable—but it empties the values of *all* variables in memory as well. Best avoid CLR until your coding is pretty advanced. Keep your DIMs at the start of your program where they will cause least problem.

There's a little bit of space saving you can use with DIMs if there are more than one to set. Thus the first line in the above fragment could be written like this:

```
10 DIM N1$(10), N2$(10)
```

Atari BASIC has the following rules for 'allowed' string variable names. A string variable name can be one or more characters. The first *must* be a letter, the others can be only letters and/or numbers. The length can pretty well be as much as you like—but as usual long names need care and take up more memory. Note that the letter(s) in a variable name must be upper case (capitals). Keywords are upper case too, so be warned—a *variable name should not be the same as a keyword*. (You *must* not give variable names the names of keywords with early Ataris. Later models are no problem, but you'd best avoid the practice as it can confuse.) As with numeric variable names, then:

1. You need a different name for each variable, unless there's *no* chance of overlap.
2. It's nice for a name to tell the reader something of its meaning (for instance, NAME\$), but:
3. Long names take up a lot of memory and can too easily be mis-typed. They can slow the program down as well.
4. Use upper case letters and numbers for your variable names, and avoid names the same as keywords.

Atari BASIC also has the same rules for using string constants and variables in PRINT statements as for numeric ones. All that POS., semi-colon, comma, ~~6~~ stuff applies in

other words. As usual, try to lay out messages with variables nicely on screen. See how I've structured the PRINT lines in that fragment above, for instance.

STRING OPERATIONS

We can do all sorts of things with numeric constants and variables. We can add, subtract, raise to a power, multiply and divide. And we can use functions like SQR and SIN. Well, we can (just about) add strings too, but the other operations do not have much meaning and can't be carried out as such.

The posh name for adding strings is concatenation. This means chaining together, and to understand how to do it you need to know about *sub-strings*. We'll deal with sub-strings in the next section.

The logical *operators* all work with strings in just the same way as with numeric quantities. I mean we can use all these symbols in IF statements:

< = > <> <= >=

Thus we can compare strings.

But what does that mean—how can one string be 'greater than' or 'less than or equal to' a second? Well, basically computers have inside themselves something like our own idea of alphabetical order. What they base their ordering on is the Atari version of the widely-used *ASCII codes*. Each character has a code, and what the micro does is to compare the strings character code by character code. (It has to do the same with strings used in variable names and as that takes time it's also another reason for using a short name.)

Here's a summary of the main Atari character codes.

Table 14.1

Codes	Characters (standard, Mode 0)
0–31	CONTROL graphics
32	space
33–47	keyboard symbols
48–57	0–9
58–64	more keyboard symbols
65–90	A–Z
91–96	more keyboard symbols
97–122	a–z
123–127	assorted symbols
128–154	inverse CONTROL graphics
155	RETURN
156–159	control characters
160–250	inverse of 33–122
251–255	keyboard and control characters

Appendix 5 goes into this in more detail, and also gives the character codes for the other text modes and the 'alternate' character set. Using these codes the micro can order strings in the same way as we humans can. (Most of us) put names in alphabetical order. Thus:

"ASMITH" < "BSMITH"
"A SMITH" < "ASMITH"
"SMITH" < "SMITHSON"

"SMITH1"	<	"SMITH2"
"SMITH"	<	"Smith"
"12345"	<	"SMITH"
"\$MITH"	<	"SMITH"
"1234"	<	"124"

And so on. Please check that you agree with the above, using the Atari code list!
So, we *can* compare strings. So we *can* use program lines like this:

```
IF A$ < B$ OR C$ > = "Money" THEN LET D$ = FINISH$
```

SUBSTRINGS

A string is a set of characters, right? Sets have subsets, right? So strings have *substrings*, and that's what *this* section is about.

As with subsets, we can define substrings as we wish. Subsets of the set of mammals could be: domesticated and otherwise; aquatic and terrestrial; tiny, small, medium, large, ginormous; and so on.

Here's a string: S\$ = "a line of characters". Substrings could be "in", "h", " ", "acters", and so on. We refer to any substring we want by its first and last character positions. Thus:

```
S$(1,3) = "a l"
```

```
S$(3,8) = "line o"
```

```
S$(2*2, 2*2+4) = "ine o"
```

```
S$(LEN(S$)-3, LEN(S$) ) = "ters"
```

These structures work with string constants as well as string variables:

```
"Shiva"(1,2) = "Sh"
```

and

```
PRINT "Shiva" (1,3); "for ever!" (6,8) gives "Shiver"
```

"Shiver"? Yes, you may tremble—but I think that word in Scots means someone who slices, and remember that string slicing is what we are doing here.

If the ending position is the end of the string, you can leave its value out. Thus:

```
S$(7, LEN(S$) ) = S$(7)
```

```
S$(1) = S$(1,LEN(S$) ) = S$
```

But beware of making the start value less than 1 or the end value greater than the length of the string, and *don't* make the end value less than the start value. Any of these can happen in a program, if you don't watch out, when the values are variables or expressions. Each leads to ERROR 5.

We can also assign new substrings. Enter a value S\$ of your own, and then use

```
LET S$(3,8) = "change": PRINT S$
```

Note that this doesn't insert extra characters, but replaces old with new.

Finally, on this subject, note that you can take a single character from a string by putting the character position twice in brackets after the string name. Thus if S\$ is "Shiva", then S\$(3,3) is "i".

The nearest the Atari micros can offer for adding strings together (poshly called 'concatenation') is as follows.

```
10 DIM S1$(20),S2$(10)
20 PRINT "Please pass me a string...":INPUT S1$:LET S2$=" for ever!"
30 LET S1$(LEN(S1$)+1)=S2$
```

```

40 FOR GO=1 TO 15
50 FOR STAR=1 TO GO:PRINT "*";NEXT STAR:PRINT S1$
60 NEXT GO

```

Of course, if you only want to print out “ATARI forever” fifteen times when the user enters “ATARI” in that program, you don’t need to go to all the trouble of line 30. The concatenation trick in line 30 is only for when you need to use the new string later in the program.

Substring handling also leads to the next nice trick. It is for when you want a long string to contain all the same characters. (You may need this for making patterns in a screen display, for instance.)

Say you want a string, P\$, to consist of 76 “hash” (#) symbols. You could do it with the instructions LET P\$ = “## . . . ##”, but that will use a lot of time and trouble and memory in a program. Try this routine instead.

```

10 DIM P$(76)
20 LET P$="#":REM ** First character of the 76
30 LET P$(76)="#":REM ** Last one now
40 LET P$(2)=P$
50 PRINT P$
60 REM ** In practice - DIM P$(76):P$="#":P$(76)="#":P$(2)=P$

```

If you can follow what that’s doing—you sure understand substrings and slicing! But it doesn’t matter if you don’t really follow it—it’s a very useful trick.

STRING FUNCTIONS

In the last chapter we looked at various numeric functions. Things like SQR and INT, these act on a numeric argument (constant, variable, or expression) to produce a number. Thus INT(3.6) gives 3 and SQR(NUMBER) gives 2 if NUMBER = 4.

BASIC also has various string functions. We’ve already met a couple—LEN, which gives the number of characters in a string argument, and CHR\$(N) which returns the character whose code is N.

LEN(“Good morning!”) = 13

and

LEN(NAME\$) = 5 if NAME\$ = “Atari”.

LEN is a string function that produces a numeric result. *Really* it’s a numeric function for that reason. Some other functions produce a string result, and those are the ones with a their name. They are the ones truly called string functions.

Here’s a full list of the functions which involve strings. There aren’t many!

ASC This gives the Atari character code for the first character of the string argument. Thus if the string S\$ is “Shiva”, then ASC(S\$) = 83, because CHR\$(83) is “S”. If the string is empty (S\$ = “”) the function returns the value 44. Don’t ask me why.

CHR\$ This gives the character whose code (see above) is its argument. The value of the argument N in CHR\$(N) should be between 0 and 255, but values outside this range do not lead to ERROR but simply repeat those within it. Some Ns give useful effects rather than characters, thus in Mode 0, ?CHR\$(125) clears the screen and ?CHR\$(253) buzzes the buzzer. Also try this:

?“I am the” ;CHR\$(34); “Atari”; CHR\$(34); “ computer.”

That solves a problem I mentioned earlier.

STR\$ This sounds strange, but is really very useful. It turns the number that follows it (its argument) into a string. Then we can use this as a string instead of a number. So

STR\$(1984) = "1984". If, for instance, you want to print a long variable number, BIGNUM, at the centre of line 5 in Mode 2, use this:

POS.9 - LEN(STR\$(BIGNUM)) / 2,5: PR.# 6; BIGNUM

Can you work that out? A trick of great value. The opposite of STR\$ is . . .

VAL This turns a string into a number (if the string contents are numeric). Thus VAL("1984") = 1984. ERROR number 18 is your prize for using VAL on a string which does not contain the characters of a number.

It's not really a string function, but I'd like to tell you about GET at this stage. We've met it before, and it is a keyword that takes in a keyboard character and turns it into a number. The number is in fact the Atari code of the character concerned. GET is better than INPUT when you hope for a single keyboard character (and even more than one if the program is suitable), and don't mind not having the prompt "?" on screen or the entered character displayed. You do not need to press RETURN after typing the character.

The little routine that follows could be part of a program, its function being not to allow the program to go on until the user presses "G".

```
OPEN # 1,4,0,"K:"
```

```
FOR A = 0 TO 1 STEP 0: GET # 1,G: IF G = 71 THEN LET A = 2
```

```
NEXT A
```

The first line there 'opens' the keyboard ("K:") for GETting. The GET statement itself is in the middle of the next line, GET # 1,G: meaning accept one character from the opened channel (number 1, the keyboard) and call its code G. The rest of the routine is simple enough. The next program gives one use for this keyword.

Program 42: GET behind me

```
9 REM ** This is a form-filling program
10 POKE 82,0:OPEN #1,4,0,"K:":LET GETIN=1000:DIM K$(100),NAME$(30),ADRS$(50),DOB$(15)
11 REM ** Change left margin & Prepare for byte-sized keyboard inputs
20 PRINT CHR$(125):PRINT "P E R S O N A L D E T A I L S F O R M ";
30 DIM U$(40):LET U$(1)="-":LET U$(40)="-":LET U$(2)=U$:PRINT U$
31 REM ** Sub-string 'multiply' trick; see last Section
40 POSITION 0,18:PRINT U$; " Type answers at white square.":PRINT " Press R
RETURN at end of each.":PRINT U$
50 POSITION 0,5:PRINT "NAME: _____":POSITION 4,5:PRINT " ";
51 REM ** Prepare line for filling
60 LET X=5:LET Y=5:GOSUB GETIN:LET NAME$=K$
61 REM ** Use GET sub-routine; note assignments before and after
70 POSITION 0,10:PRINT "ADDRESS _____":POSITION 7,10:PRINT " ";
80 LET X=8:LET Y=10:GOSUB GETIN:LET ADRS$=K$
90 POSITION 0,15:PRINT "DATE OF BIRTH _____":POSITION 13,15:PRINT " ";
100 LET X=14:LET Y=15:GOSUB GETIN:LET DOB$=K$
101 REM ** See power of sub-routines: used again and again
110 PRINT CHR$(125):POSITION 13,2:PRINT "S U N A M A R Y":PRINT U$
120 PRINT :PRINT "NAME: ";NAME$:PRINT :PRINT "ADDRESS: ";ADRS$:PRINT "DATE OF BIRTH: ";DOB$
130 POSITION 0,20:PRINT
140 FOR W=0 TO 1 STEP 0:NEXT W
990 END
999 REM ** GETIN: Input sub-routine with GET
1000 LET K$="":FOR CHR=1 TO 100:REM * Select max length of input string
1010 GET #1,K:REM * Await key-press; put code into K
1020 IF K=155 THEN RETURN
1021 REM ** Detect RETURN key press; Note sub-routine RETURN not always at end
1030 POSITION X-1+CHR,Y:PRINT CHR$(K);
1040 LET K$(LEN(K$)+1)=CHR$(K)
1041 REM ** Concatenation
1050 NEXT CHR
```

As well as showing you how to process the result of the GET statement (lines 1020, 1030 and 1040), I use the program to show you in practice a couple of points made in the last section. They are the 'fill a long string with identical characters' trick (line 30) and the Atari form of concatenation (line 1040).

That program is well worth study, therefore, as a general routine using strings. It is also quite important as an example if you intend to develop form-filling software.

SUMMARY

Here we have looked at strings—what they are, and how to handle them. I've told you about substrings and string functions, and some uses of GET. The keywords met here are as follows:

ASC	GET
CHR\$	LEN
• COM	STR\$
DIM	VAL

DO IT YOURSELF

1. Improve Program 15 by using suitable string functions so that once it has the input string, it will produce screen after screen of variations on the pattern. Include a few seconds' pause after each display. And include colour variations and sound as you wish.
2. Write a program to print out on screen as much of the Atari's character set as you can.
3. Devise a program rather like number 37 to utilize string functions.
4. Write a program which asks the user to enter his or her date of birth as a string of digits, and then prints out the 12 times table for the number. The input should be as a string, and fully mugtrapped.
5. Study Program A2 . . . (at the end of the book) in the light of string-handling.
6. Convert the routine given on Page 142 for centring on the line into a closed subroutine. Then use that subroutine time and again within a program to print out a set of titles etc. like the title page of a book.
7. Devise a version of Program 42 of your own to use string operations and functions to
 - (a) accept items of personal data about the user;
 - (b) display the complete set of data neatly on screen.
8. Explore GET as a way of getting, checking, and using keyboard input. Add VAL if you wish.

15 Bug in a Rug

The story (myth?) goes that in the very early days of electronic computing, far less than half a century ago, computer operation failures were often traced to the activities of insects making their homes among the nice warm circuits. Alas, users of modern computers no longer have that excuse for program failures. Coffee on the keyboard is *our* main threat. Still, the use of the word 'bug', to mean an unknown cause of software malfunction, remains. Here I would like to look very briefly at the process of 'debugging', one of the last stages of program development.

In Chapter 11, and elsewhere, I have tried to describe a structured approach to program development—the top-down modular approach. I have also noted that a program written as a series of subroutines is much easier to test. That's because you can test each subroutine on its own. In this chapter, I shall discuss debugging approaches with a complete bugged program rather than a single subroutine.

A BUGGED PROGRAM

The program that follows, therefore, was not developed in a structured way. It contains four major faults and a number of minor ones. I'd like you to enter it—as you do this, by all means look out for the faults. If you find any feel proud, but please do not correct them!

Program 43a: BUGGED bugs

```
10 DIM BUG1$(2):LET BUG1$="*#:"
20 DIM BUG2$(2):LET BUG2$=">#:"
30 DIM BUG3$(4):LET BUG3$=">=;:"
40 DIM BUG4$(3):LET BUG4$="<0:"
50 GRAPHICS 1:SETCOLOR 1,0,0:SETCOLOR 2,9,10:SETCOLOR 4,9,12
60 FOR BUG=1 TO 50
70 POSITION RND(0)*20,RND(0)*20
80 IF RND(0)*4=1 THEN PRINT BUG1$:GOTO 120
90 IF RND(0)*4=2 THEN PRINT BUG2$:GOTO 120
100 IF RND(0)*4=3 THEN PRINT BUG3$:GOTO 120
110 IF RND(0)*4=4 THEN PRINT BUG4$:GOTO 120
120 FOR T=1 TO 120:POSITION T,T:PRINT " BUGS ":NEXT A
130 END
```

Entered? Don't RUN for a moment. First—can you see what this program is set up to do?

Lines 10–40 define four bugs (small creature type) and 50 sets up a Mode 1 screen. Then, from line 60, we print 50 random bugs at random screen sites. After that, we display a diagonal title and stop.

Not a fantastic program—but does it work? Try it with RUN. And you get a bug—but

not an insect, rather one ERROR message—plus the title BUGS in the text window. The ERROR report tells us of a fault in line 120. It's ERROR 13, and if you like you can refer to Appendix 6 to find out what that means. But here's the line—can you see what's wrong?

```
120 FOR T = 1 TO 120: POS.T,T: PR. "BUGS": N.A
```

I hope you can find, and correct the error yourself! (Maybe that's too hard a problem for me to give you—there is more than one error in this line, we'll find. . .)

RUN again—to get a strange mix-up of "BUGS" in the text window, with not a sign of a screen bug (insect) above. Still, there's no ERROR report this time—the program has run happily to the end. But it hasn't worked as planned. (There's a useful lesson—a program can work but not be correct!) For a start—no fifty insects. And why are the BUGS titles in the text window?

Aha, we are working in Mode 1, and we need to use the Channel 6 instruction after each PRINT statement. I'd forgotten that. All the PRINT statements designed to put stuff on screen should have #6 after them. Let's correct those. Thank Atari for the editing routine. . . . Each of lines 80–120 needs this change.

The next time we RUN, we do seem to be getting somewhere. No insects appear, but at least there's something of a diagonal line of titles. But ERROR again—the same line, but a different error number. Appendix 6 tells us that the message we get here means that we've tried to print off the screen limits. Look again at line 120—ah, wait a moment—that should be 20 not 120. A typing error, with a profound effect. 20 lines in Mode 1, not 120! Line 120 thus becomes:

```
120 FOR T = 1 TO 20: POS.T,T: PR.# 6; "BUGS": N.T
```

But it still doesn't work—the title doesn't start at the top of the screen, *and* it isn't a simple diagonal. And we still have that ERROR message. The former is one of the four serious faults—can you explain it? Can you solve it? The other two are connected—they show bad planning. There are twenty lines, but our title has six characters and there are only 20 character spaces in a line. See what I mean?

To sort out all these faults, we need several changes to line 120 still. Here is the final version:

```
120 FOR T = 0 TO 15: POS.T,T: PR.# 6; "BUGS": N.T
```

Did you think to get one ahead of me by using POKE 82,0? (Recall that this sets the left hand margin to the edge of the screen.) Hard luck if you did—a good idea, but that statement works only in Mode 0.

Anyway, we do at last have our diagonal title. It's a shame that the title appears at the end of the program rather than at the start—because we still haven't got any sign of a bug on screen. We know the last two lines work, so let's get them out of the way: insert 115 STOP. That will let just the first part of the program work—the part that doesn't work, if you see what I mean.

We want fifty 'insects' on screen—but can't get even one. Why is that? Maybe you've spotted major error number two by now—the FOR in line 60 is not matched by a NEXT anywhere. We *do* get an ERROR report if there's a NEXT without a FOR—but few computers can spot the reverse problem. (Why not?) So we have to watch out for it all by ourselves—and we've missed it so far.

Well, we need a NEXT BUG somewhere. And we need it before the closing credits line (120)—so insert 112 NEXT BUG. OK? Simple!

RUN. OK? *Not* simple! There's no change in the display. Unless you count the fact that it now lasts much longer. What's wrong? Quite a lot! For a start each of lines 80–110 have GO TO 120, which is after the STOP statement, and certainly not the correct address—112. We could change all of those GO TO addresses, but let's be clever—delete 112, and put NEXT bug instead of GO TO 120 in each of the four lines. There's nothing wrong with having several NEXTs in a loop, and you know my views about GO TO. (Those views have been well borne out now, because the line number had to be changed.)

Line 80 now looks like this:

```
80 IF RND(0)*4 = 1 THEN PR.# 6; BUG1$: N.BUG
```

I can leave you to edit the next three lines in the same kind of way. If you now try to RUN yet again, you'll find no change. Well, still no insects—but the loop seems to go through much more quickly.

In practice, either you see at once what's wrong or you spend ages trying to find it. There's still another problem with those four IF lines. (More than one, in fact, as I've just said.) I'll be honest and tell you that the next error we'll correct is one that I did not make on purpose. The Atari RND structure differs from that in most other micros, and I was confused when I put those lines down.

Here's the correct version of line 80, getting over the RND problem that I'd forgotten, putting in an INT and changing the possible values from 1–4 to 0–3. I hope you see why we need these changes!

```
80 IF INT(RND(0)*4) = 0 THEN PR.#6; BUG1$: N.BUG
```

Change line 80, and the next three lines similarly, and RUN again.

Progress at last! Each time we RUN the program now, at least a couple of 'insects' appear on screen. (I tried it about thirty times, and the record was six.) In most cases we get the text-window message STOPPED AT LINE 115; but still sometimes an ERROR message appears.

For some reason, the loop does loop now, but never gets anywhere near its official two score and ten goes. We can check that readily enough, using a valuable little debugging trick. Add at the front end of line 70—PRINT BUG:. Now, each time you run, the text window shows how far BUG gets—nowhere near 50, that's for sure. The printing out of carefully chosen variable values is an extremely useful aid to finding bugs in programs. (Don't forget to remove the statements concerned when they're finished with, though.)

Well, we are getting closer to the program as at first 'designed', even if there's a fair way still to go. Perhaps it's time to remove line 115.

TRACING

Some versions of BASIC have another useful debugging tool, called TRACE. When you use the feature the display includes a printout of each line number met. What happens in practice is that you end up with a screen full of line numbers, rarely much help. The use of TRACE needs fair care. Anyway, in the case of the Atari, you can't use it, because we don't have it. What we really *want* with TRACE is to be able to trace the micro's path through no more than a dozen or so lines. You can then check on paper that the path is correct.

It is, therefore, often better to use your own 'trace' statements in a few lines of your choice. Say your program should follow through lines 100, 110, 170, 180, 120. Put PRINT 100 at the start of line 100, PRINT 110 at the start of 110, and so on—then you can see *exactly* where the program goes. You will most likely need to slow the program down a lot to see the line numbers appear. The best way to do that is to use lots of STOP statements and CONT after each.

Try this kind of trace on our partly debugged program now, if you want. It will help you to find out what the bug is that's still messing up *our* program. . . . And its cause is tricky—but this is big fault number 4. *Can* you spot it? Figure 15.1 shows a flowchart of what I aimed at. Figure 15.2 shows a flowchart of what we actually have. Of course if I had drawn the flowchart before coding, the errors would have been fewer, surely. Can you see the subtle problem? The computer isn't choosing a number from one to four at random and then printing the corresponding insect, is it? It's choosing a random number between 0 and 3. If it's 0 it prints insect 1—if not it chooses again. And so on. So the loop comes to an early end whenever the micro gets to line 110 and chooses a number that isn't 3. Geddit?

How do we get over this problem? There are several ways. The one nearest the correct flowchart involves adding LET INSECT = INT(RND(0)*4) at the end of line 60 and changing the left-hand side of the IF statement expression in each of lines 80–110 to INSECT. Much better now—the value of BUG gets well on towards 50—perhaps 20 or 30 or even more—before the program hits ERROR 141. You may recall that this

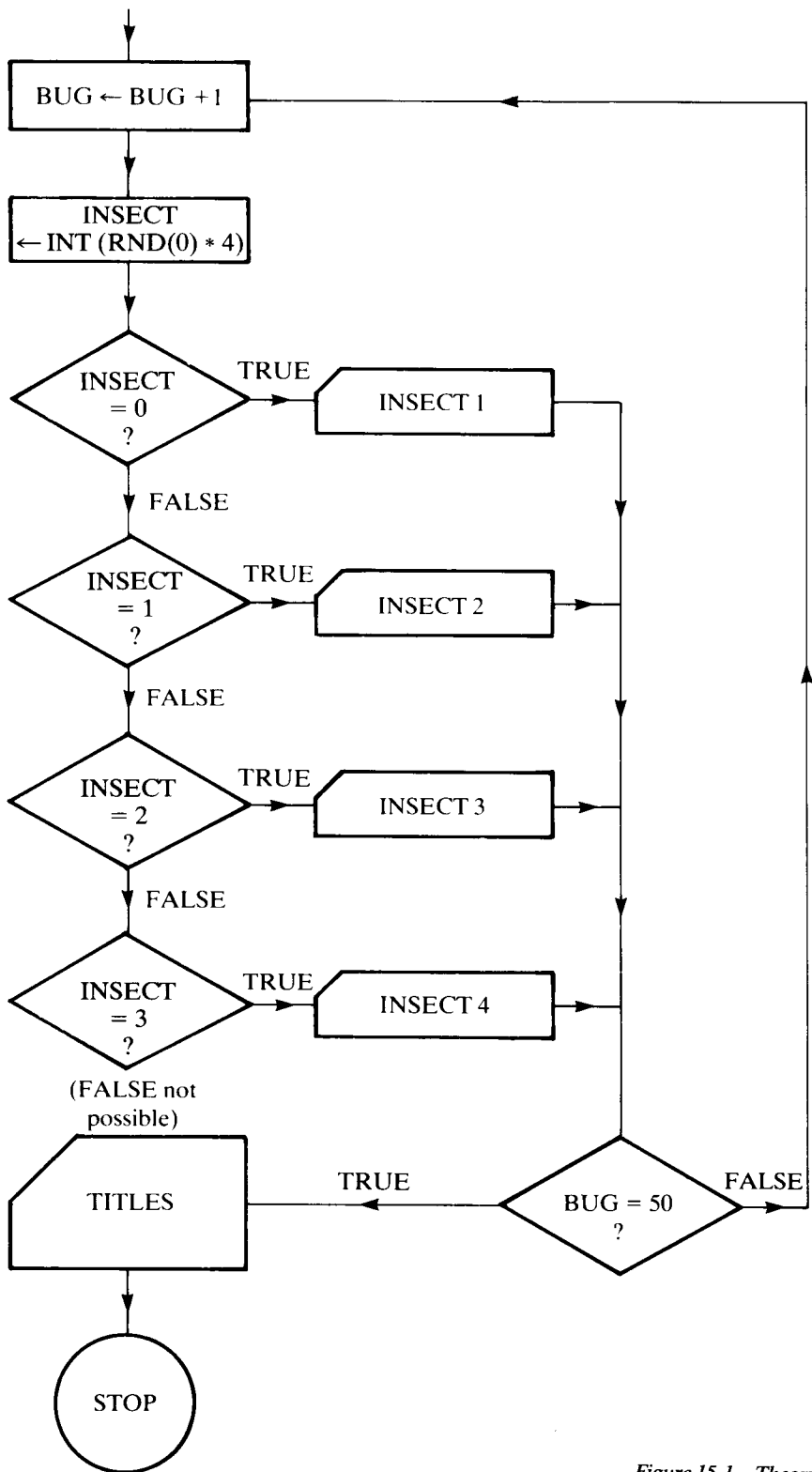


Figure 15.1 Theory.

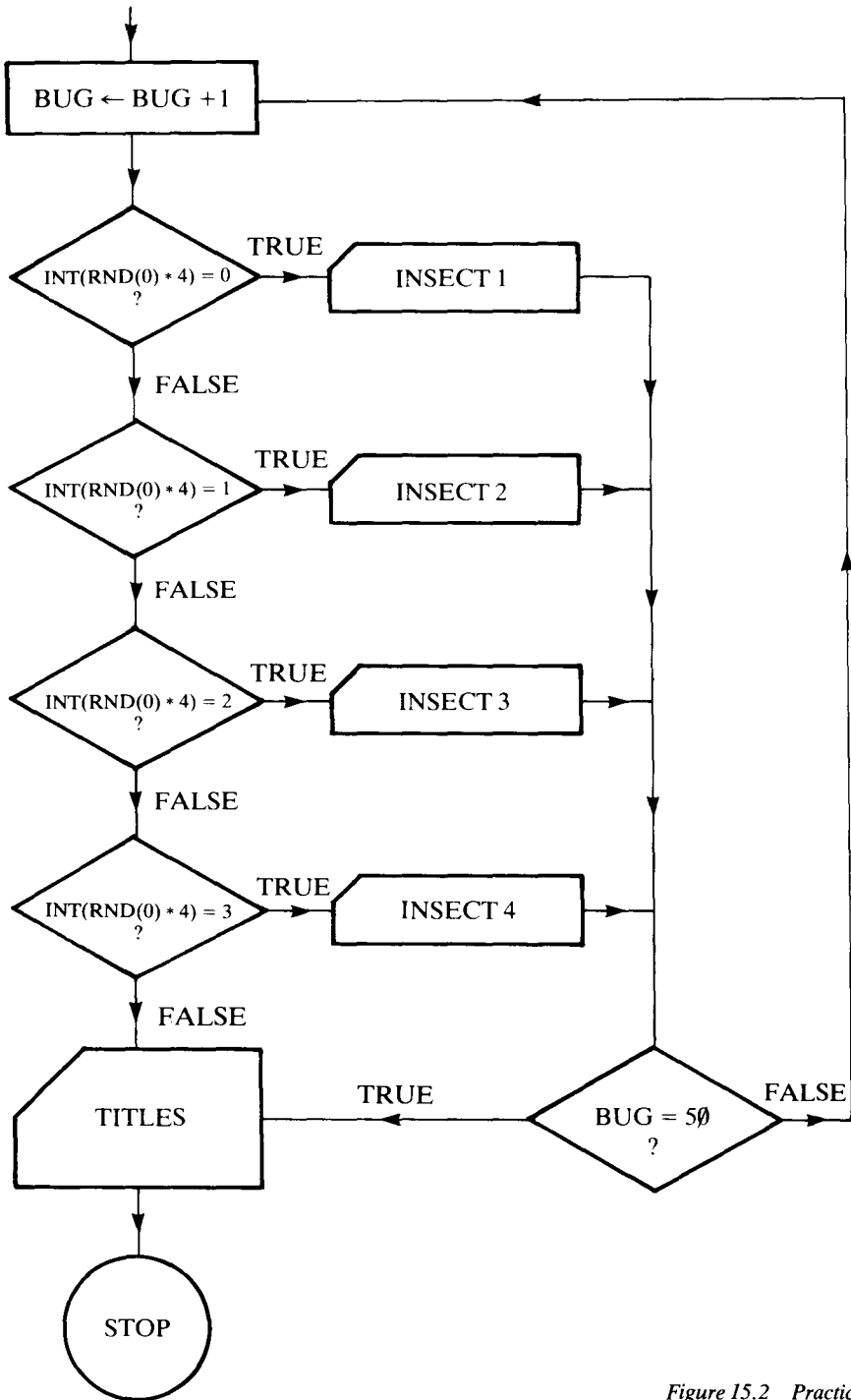


Figure 15.2 Practice.

ERROR means that you've tried to PRINT outside the screen limits. Our last major change, therefore, involves line 70. Can you make it?

If you succeed, you've debugged the program! A nice display of random 'insects' (ugly ones, I admit) and the diagonal bugs title. We've got the program working as scheduled.

We've cleared syntax and typing errors, reminded ourselves of correct RND and loop structures and seen that a flowchart *can* help program development. I've also emphasized the need for careful screen layout.

There are still little problems left, and there are still ways we can structure the program better too (like putting 115 NEXT BUG back in, and clearing the NEXTs from the four lines before it). Mainly the program's too fast—slow down the loop then. And it's a shame the title splats some of the insects so quickly—a wait loop will solve that. And we could restart after a while. All that (and more) gives us this polished version. Check through with care . . .

Program 43b: Bunny bugs

```
10 DIM BUG1$(2):LET BUG1$="#:"
20 DIM BUG2$(2):LET BUG2$=">#"
30 DIM BUG3$(4):LET BUG3$=">=:"
40 DIM BUG4$(3):LET BUG4$="<0:"
50 GRAPHICS 17:SETCOLOR 1,0,0:SETCOLOR 2,9,10:SETCOLOR 4,9,12
60 FOR BUG=1 TO 50:LET INSECT=INT(RND(0)*4)
70 POSITION RND(0)*19,RND(0)*22:SOUND 1,10*5*INSECT,10,10:FOR W=1 TO 50:NEXT W
80 IF INSECT=0 THEN PRINT #6;BUG1$
90 IF INSECT=1 THEN PRINT #6;BUG2$
100 IF INSECT=2 THEN PRINT #6;BUG3$
110 IF INSECT=3 THEN PRINT #6;BUG4$
115 NEXT BUG:FOR W=1 TO 1000:NEXT W:SOUND 1,0,0,0
120 FOR T=0 TO 15:POSITION T,T+3:PRINT #6;" bugs ":NEXT T
130 OPEN #1,4,0,"K":GET #1,K:RUN
```

I'm not going to comment about the 'and more' changes—most should be obvious at once or after a little study. One thing worth noting, however—the last line trick—wait for a key press to go on. Useful!

CLOSING TIPS

Here are some other debugging tricks worth bearing in mind.

1. Edit in a REM before a statement you don't want used for the time being. If you delete a statement that offends you, you may need to put it back later—and will you remember it then? But beware of REMs in multistatement lines—the REM will tell the computer to ignore everything that comes after it, more perhaps than you expect.
2. Statements often worth REMoving in this way are ones with RND. It often helps to have a fixed value rather than one that varies at random during testing. You may have, for instance, LET X = RND(0)*10. Change this during testing to LET X = 5: REM LET X = RND(0)*10.
3. It's very easy to forget where you've put in extra REMs like this. Use the 'inverse' key for the messages after each REM that you plan to remove. Then, when you list, these lines stand out very clearly.
4. In this chapter I've not tried to use TRAP. You will recall that this has the effect of diverting program control to a closed subroutine of your choice if an ERROR appears during a RUN. It can be useful to use TRAP during your testing process—then the command PRINT PEEK(195) will tell you the error that opened the TRAP—but you can't find out the *line* in which the error occurred.

The best way to remove the trials of debugging is not to have bugs in the first place. Well, none of us is perfect, we all make some programming mistakes! All the same, the modular top-down flowchart-based approach I've discussed in Chapter 11 is a *great* help to bug-prevention. And if your program's in modules, you'll find it far easier to sort out if it does go wrong.

MORAL Use modules and bug off . . .

16 Graphic Description

And now we come to one of the last major untouched areas of Atari BASIC programming—‘graphics’. The word includes all aspects of ‘drawing’ and ‘painting’ as applied to a colour micro computer—putting points, lines, and shapes on screen, and blocking in areas solidly. The term also includes making up your own characters and moving them about—but we’ll leave this ‘sprite’ (player-missile) feature to the next chapter.

The simple graph work I’m going to cover now is not very hard—after all, we’ve met pretty well all the ideas before.

MODES

As you well know by now, your computer has a number of different ‘modes’ of operation. When you switch it on, you get Mode 0, the ‘pure text’ mode. And you can call on any of the others using the GRAPHICS (GR.) instruction. Here again is a list of the nine main modes, repeated from Page 44.

Table 16.1: The main Atari modes

Mode	Action	Memory demand	Number of screen sites	Colours
0	Text	992	24×40	3
1	Big text	672	$20 \times 20 + 4 \times 40$	5
2	Large text	480	$10 \times 20 + 4 \times 40$	5
3	Low resolution graphics	432	$20 \times 40 + 4 \times 40$	4
4	Higher resolution	696	$40 \times 80 + 4 \times 40$	2
5	Higher resolution	1176	$40 \times 80 + 4 \times 40$	4
6	Higher resolution	2184	$80 \times 160 + 4 \times 40$	2
7	Higher resolution	4200	$80 \times 160 + 4 \times 40$	4
8	Highest resolution	8138	$160 \times 320 + 4 \times 40$	3

What is of most interest in this context is the fourth column, the number of screen sites available to you in each mode, and the number of colours you can have on screen at once (fifth column). The table includes mention of the text window (the 4×40 in it)—you know that if you wish to avoid having that feature you add 16 to the mode number concerned. You also know that GR. such and such clears the screen. All the same, I

suggest you always use PR.CHR\$(125) after it to remind you to think about layout. (Use #6 of course for main screen in a graphic mode.) Anyway if you add 32 to the mode number you get a version which does not clear the screen for you.

The normal clear-screen approach gives you a black background ('paper') for your work, but you can use SETCOLOR to change that. Really the best way you can revise all these ideas is to enter, RUN, and study the next program. There's nothing new here, except the beautiful effect!

Program 44: SF skyline

```
10 GRAPHICS 5:LET DLY=1000:REM * Named subroutine address
20 PRINT CHR$(125):SETCOLOR 2,4,4:SETCOLOR 4,4,4:REM * Clear to uniformly yakky
   coloured screen
30 SETCOLOR 0,6,0:SETCOLOR 1,12,0:REM * Change plotting colour registers
40 FOR X=0 TO 79 STEP 6:COLOR 1:PLOT X,0:SOUND 1,20+2*X,10,10:DRAWTO 79-X,47:GOS
   UB DLY:NEXT X:REM * Register 0
50 FOR Y=0 TO 47 STEP 4:COLOR 2:PLOT 0,Y:SOUND 1,200-3*Y,10,10:DRAWTO 79,47-Y:LE
   T X=Y:GOSUB DLY:NEXT Y:REM * Reg 1
60 GOSUB DLY
70 PRINT "SF skyline          by          hmed Bloggs"
80 FOR F=0 TO 1 STEP 0:SETCOLOR 0,RND(0)*15,RND(0)*15:SETCOLOR 1,RND(0)*15,RND(0
   )*15:SOUND 1,RND(0)*250,10,10:NEXT F
1000 FOR W=1 TO 10*X:NEXT W:RETURN :REM * Simple!
```

Got it? Here are all our old friends of the graphic work days (Chapter 8): GRAPHICS, SETCOLOR, COLOR, PLOT, DRAWTO, to which I could have added, but didn't, POSITION. No problem there, I hope. . . .

GRAPH-PLOTTING

The simplest form of graph is the bar-chart (histogram) you may have met at first school. Program 45 provides a versatile example of plotting the former. Here I use an equals sign for the plotting character, you may prefer the inverse space.

Note the DIM statement at the start of the program—clearly it does not refer to a string as before. This use of DIM is to set up what's called a numeric array (see Chapter 19). Briefly, an array is a set of numbers we can pull out in any way we like, in the same way as a string is truly a set of characters we can pull out in any way (by using substrings). You may well find that you follow what's going on with this array; if not, don't worry—just concentrate on the graph-plotting.

Program 45: Rain for a year

```
10 DIM M(12),DATA$(4),TITLE$(10):LET TEST=500:PRINT CHR$(125):POKE 752,0
20 FOR MONTH=1 TO 12
30 FOR G=0 TO 1 STEP 0:POSITION 5,(MONTH-1)*2:PRINT "Data for month ";MONTH;"
   ":POSITION 22,(MONTH-1)*2:INPUT DATA$
40 LET FLAG=0:GOSUB TEST:IF NOT FLAG THEN LET G=2:REM ** Input check; YOU chec
   k 'logic'!
50 NEXT G:LET M(MONTH)=VAL(DATA$):REM ** Valid data into 'array' (list)
60 NEXT MONTH
70 FOR W=1 TO 500:NEXT W:PRINT CHR$(125)
80 FOR LINE=1 TO 20
90 FOR MONTH=1 TO 12
100 IF M(MONTH)/5<20-LINE THEN PRINT " ";
110 IF M(MONTH)/5>=20-LINE THEN PRINT "==" ;
120 NEXT MONTH:PRINT
130 NEXT LINE
140 PRINT " 1 2 3 4 5 6 7 8 9 10 11 12  Month":REM * : Give x-axis
150 FOR LINE=2 TO 20:POSITION 0,LINE:PRINT 100-(5*LINE):NEXT LINE:REM * Give y-a
   xis
160 POSITION 0,22:PRINT "Title (<10 characters)":INPUT TITLE$
170 POSITION 0,22:PRINT " "
180 FOR F=0 TO 1 STEP 0:POSITION 27,10:PRINT TITLE$:FOR W=1 TO 500:NEXT W:POSITI
   ON 27,10:PRINT " " :FOR W=1 TO 250:NEXT W:NEXT F
```

```

490 END
499 REM ** Input checks
500 IF NOT LEN(DATA$) THEN LET FLAG=1:RETURN :REM ** Reject simple RETURN key p
ress
510 FOR CHR=1 TO LEN(DATA$):IF ASC(DATA$(CHR,CHR))<46 OR ASC(DATA$(CHR,CHR))>57
THEN LET FLAG=1:REM * Check + care
520 NEXT CHR:RETURN

```

This program currently assumes input data no larger than 100 (mm)—if your rainfall is greater, you can either move, or change the 5 in lines 100 and 110 to a more apt divisor. But of course this program isn't just for rainfall.

Should you wish to have more columns than twelve—up to 18, just change all the 12s; up to 36, ditto, but also halve the strings in lines 100 and 110. In both cases you may have to change the TITLE routine. Various other polishes are possible, but I'll leave you to work on those when you get to the last section of the chapter.

The step to a pictogram is a small one. Figure 16.1 shows a typical screen display with such a program, where the little cars are user-defined as well. (Yes, it was made with a different micro!) As I hope Program 45 has shown, it is not hard to instruct the Ataris to 'plot' histograms. (I put the word plot in speech marks, because the simplest approach is quite good enough: using PRINT in Mode 0.)

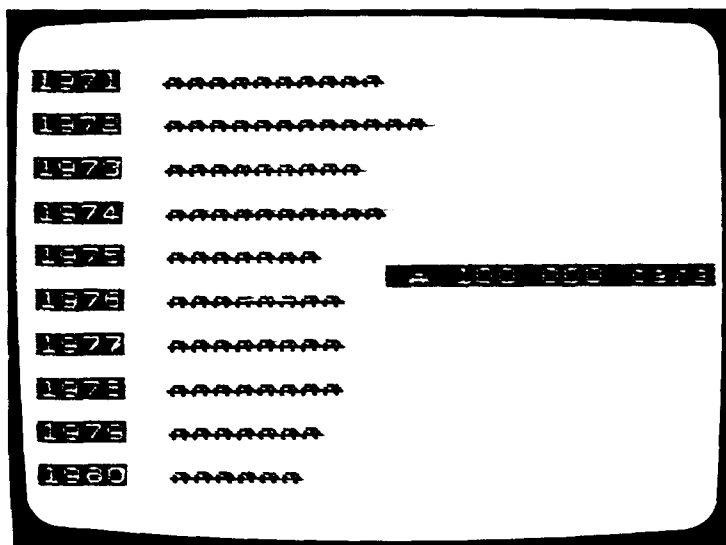


Figure 16.1 A pictogram.

These micros, like most others, are also able to plot proper line graphs. However, this is quite a lot more tricky than dealing with histograms, and needs more understanding.

You can plot material on the screen in two ways—firstly, by using PRINT in some clever fashion as with the histogram, and secondly by using the special PLOT and DRAWTO statements. The former approach was shown in Program 12, Tom Tiddler's Ground, if you can remember that far back. It shows how one can use POS....: PR.... to place a character at any given point on screen.

Using this approach brings us to Program 46. Here I have simply placed a full-stop at the top of each bar of the histogram. As you may now see, a line graph is in fact no more than a special form of histogram.

Program 46: Rain line

```

10 DIM M(12),DATA$(4),TITLE$(10),FLAG(12):LET TEST=500:LET PRINT=600:PRINT CHR$(
125):POKE 752,0:GOSUB 700
20 FOR MONTH=1 TO 12
30 FOR G=0 TO 1 STEP 0:POSITION 5,(MONTH-1)*2:PRINT "Data for month ";MONTH;"

```

```

":POSITION 22,(MONTH-1)*2:INPUT DATA$
40 LET FLAG=0:GOSUB TEST:IF NOT FLAG THEN LET G=2:REM ** Input check; YOU check 'logic'!
50 NEXT G:LET M(MONTH)=VAL(DATA$):REM ** Valid data into 'array' (list)
60 NEXT MONTH
70 FOR W=1 TO 500:NEXT W:PRINT CHR$(125)
80 FOR LINE=1 TO 20
90 FOR MONTH=1 TO 12
100 IF M(MONTH)/5<20-LINE THEN PRINT " ";
110 IF M(MONTH)/5=20-LINE THEN GOSUB PRINT
120 NEXT MONTH:PRINT
130 NEXT LINE
140 PRINT " 1 2 3 4 5 6 7 8 9 10 11 12 Month":REM * Give x-axis
150 FOR LINE=2 TO 20:POSITION 0,LINE:PRINT 100-(5*LINE):NEXT LINE:REM * Give y-axis
160 POSITION 0,22:PRINT "Title (<10 characters)";:INPUT TITLE$
170 POSITION 0,22:PRINT " ";
180 FOR F=0 TO 1 STEP 0.5:POSITION 27,10:PRINT TITLE$:FOR W=1 TO 500:NEXT W:POSITION 27,10:PRINT " "
190 FOR W=1 TO 250:NEXT W:NEXT F
490 END
499 REM ** Input checks
500 IF NOT LEN(DATA$) THEN LET FLAG=1:RETURN :REM ** Reject simple RETURN key press
510 FOR CHR=1 TO LEN(DATA$):IF ASC(DATA$(CHR,CHR))<46 OR ASC(DATA$(CHR,CHR))>57 THEN LET FLAG=1:REM * Check + care
520 NEXT CHR:RETURN
599 REM ** PRINT routine
600 IF FLAG(MONTH) THEN PRINT " ";:RETURN
610 LET FLAG(MONTH)=1:PRINT "..":RETURN :REM ** CTRL+I,0 is better
699 REM ** Initialise FLAG array
700 FOR A=1 TO 12:LET FLAG(A)=0:NEXT A:RETURN

```

Compare this with Program 45 (from which it is easily edited). Pretty much the same comments apply.

Extra, however, is the new FLAG-array; I DIMensioned this in line 10, set the values of its 'elements' to zero in line 700, and used it in the new subroutine, PRINT, starting at line 600. Maybe you won't quite follow this yet as we haven't studied arrays fully. If you don't, don't worry—but check again after working through Chapter 19. There's a useful trick here you may like to use yourself later—putting a set of flags in an array. (A 'flag' is a variable whose value is 0 in one set of conditions and 1 in another—looking at the value of the flag will then tell us what the conditions are.)

With this kind of approach it is not too hard to set up the display to show any combination of symbols (standard, graphic, or even defined) to make any picture, design, graph, or map you wish. The finished design may be stored on cassette or printed out on the printer.

This use of PRINT to put designs on the screen, although fairly effective in the end, has what I have always called only low resolution. The procedure is also pretty slow! Using it we can place characters in only 960 different positions in Mode 0—so that the detail is coarse in the best of cases.

Let's use the Atari plotting facilities in a proper graph program. 'Line graph', Program 47, asks the user to supply the *x* and *y* coordinates of a given number of points in order of increasing values of *x*. The screen is cleared and straight lines appear from point to point. The program tests for invalid data, up to a point at least. A better approach to drawing line graphs follows in a moment, so don't be concerned if you feel this is complex.

Program 47: Line graph

```

10 DIM A$(4):LET INPUT=400:LET TEST=500:OPEN #1,4,0,"K:"
20 PRINT CHR$(125)
30 POSITION 4,4:PRINT "How many points";:LET X=19:LET Y=4:GOSUB INPUT
40 IF A=0 THEN FOR W=0 TO 1 STEP 0.5:NEXT W
50 LET N=A:DIM X(N):DIM Y(N):REM ** Setting up two numeric arrays (lists) here
60 PRINT CHR$(125);"Please enter values....",NOTE 0<x<320 0<y<150," in order of increasing x"
70 FOR V=1 TO N
80 PRINT "Point ";V;": x=";:LET X=14:LET Y=PEEK(84):GOSUB INPUT:LET X(V)=A
90 POSITION 20,PEEK(84)-1:PRINT "y=";:LET X=22:LET Y=PEEK(84):GOSUB INPUT:LET Y(

```

```

V)=A
100 NEXT V:FOR W=0 TO 1000:NEXT W
110 GRAPHICS 0:SETCOLOR 1,0,0:SETCOLOR 2,3,14:SETCOLOR 4,0,10:COLOR 1:FOR V=1 TO
N-1
120 LET M=(Y(V+1)-Y(V))/(X(V+1)-X(V)):REM * Gradient
130 LET C=Y(V)-M*X(V):REM * Intercept
140 FOR X=X(V) TO X(V+1):LET Y=M*X+C:LET Y=159-Y:PLOT X,Y:NEXT X:NEXT V
150 FOR W=1 TO 1000:NEXT W:GET #1,CONTINUE
160 RUN
390 STOP
391 REM ove when finished and tested
399 REM ** INPUT routine
400 FOR IN=0 TO 1 STEP 0:POSITION X,Y:PRINT "          ":POSITION X,Y:INPUT A$:G
OSUB TEST:IF NOT FLG THEN LET IN=2
401 REM ** Nested sub-routines
410 NEXT IN
420 LET A=VAL(A$):RETURN
499 REM ** Numeric TEST routine
500 LET FLG=0:IF NOT LEN(A$) THEN LET FLG=1:RETURN
510 FOR CHR=1 TO LEN(A$):IF ASC(A$(CHR,CHR))<46 OR ASC(A$(CHR,CHR))>57 THEN LET
FLG=1
520 NEXT CHR:RETURN

```

Apart from the line-drawing routine itself (which is a bit tough), there are some points I'd like you to note. See, for instance, the way I have mixed GET and INPUT inputs. Study the quite neat (in my opinion) input and input testing routines. (I must admit I have had to put in PEEK (84) in line 90: what this function does is to give the value of the screen line the micro is on at the moment. If it troubles you, ignore it. But do still use it!)

If at any time in the distant past you studied what teachers call 'coordinate geometry', you may recognize the form of the second statement in line 140. If not, don't worry too much about how this program works. All I'd like you to do is to come to grips with PLOT.

The Atari has another important high-resolution graphics command, as you know. It is DRAWTO. We use DRAWTO X,Y to put on screen a straight line from the last point visited to the point X,Y. The next program, Program 48, uses this rather than $y = mx + c$ to draw the lines. As before (third statement in line 140 above), I've had to allow for the fact that the Atari graphics screens have the 'origin' (point $x = 0, y = 0$) at the top left of the 'paper', rather than, as is normal, the bottom left.

Program 48: Line graph 2

```

10 DIM A$(4):LET INPUT=400:LET TEST=500:OPEN #1,4,0,"K:"
20 GRAPHICS 0:PRINT CHR$(125)
30 POSITION 4,4:PRINT "How many points";LET X=19:LET Y=4:GOSUB INPUT
31 REM ** Input 0 to stop
40 IF A=0 THEN FOR W=0 TO 1 STEP 0:NEXT W
50 LET N=A:DIM X(N):DIM Y(N)
60 PRINT CHR$(125);"Please enter values....",,"NOTE 0<x<320 0<y<150",," in
order of increasing x"
70 FOR V=1 TO N
80 PRINT "Point ";V;": x=";:LET X=14:LET Y=PEEK(84):GOSUB INPUT:LET X(V)=A
90 POSITION 20,PEEK(84)-1:PRINT "y=";:LET X=22:LET Y=PEEK(84):GOSUB INPUT:LET Y(
V)=A
100 NEXT V:FOR W=0 TO 1000:NEXT W
110 GRAPHICS 0:SETCOLOR 1,0,0:SETCOLOR 2,3,14:SETCOLOR 4,0,10:COLOR 1:PLOT X(1),
159-Y(1):FOR V=2 TO N
120 DRAWTO X(V),159-Y(V)
140 NEXT V
150 FOR W=1 TO 1000:NEXT W:GET #1,CONTINUE
160 RUN
399 REM ** INPUT routine
400 FOR IN=0 TO 1 STEP 0:POSITION X,Y:PRINT "          ":POSITION X,Y:INPUT A$:G
OSUB TEST:IF NOT FLG THEN LET IN=2
410 NEXT IN
420 LET A=VAL(A$):RETURN

```

```

499 REM ** Numeric TEST routine
500 LET FLG=0:IF NOT LEN(A$) THEN LET FLG=1:RETURN
510 FOR CHR=1 TO LEN(A$):IF ASC(A$(CHR,CHR))<46 OR ASC(A$(CHR,CHR))>57 THEN LET
FLG=1
520 NEXT CHR:RETURN

```

You will not *see* much difference between this program and the last one, but it's simpler and uses less memory. Indeed you can edit Program 48 from Program 47 in only a few minutes. The result is shorter.

GET YOUR FILL

Atari BASIC offers one more graphics statement, one we haven't yet met. It is a version of the 'general input/output statement', XIO. XIO has in fact over twenty forms, but the one we'll use now is the only one of those that is of much value in a book like this. It is the 'fill' statement, and has the form XIO 18,6,0,0,"S:" (the final colon is not really needed).

What XIO 18... does is to fill a four-sided area of screen with the colour you state. It needs some care to set it up—not only must you define the four corners of that area, but you must do so in a certain order and also use a POKE to define the colour.

Program 49, Paper maze, is quite a nice demonstration of this in practice. See the statement itself in line 80. I suggest that you enter and RUN this, and then study the notes that follow.

One point first, however—I've written this in graphics Mode 11. The early Ataris do not support this Mode, hence the program won't run in any other without fairly major changes. If you can't enter it as it stands, study the notes first and then modify it as they suggest.

Program 49: Paper maze

```

10 GRAPHICS 11:FOR B=20 TO 2 STEP -2:FOR A=0 TO 15:COLOR 15-A
20 LET X=5+RND(0)*52:LET Y=5+RND(0)*165
30 PLOT X+B,Y+B/2
40 DRAWTO X+RND(0)*B,Y
50 DRAWTO X,Y
60 POSITION X,Y+RND(0)*B/2
70 POKE 765,15-A
80 XIO 18,6,0,0,"S:"
90 SOUND 1,RND(0)*250,10,10:FOR W=1 TO 100:NEXT W
100 NEXT A:NEXT B
110 FOR W=0 TO 1 STEP 0:SETCOLOR 4,RND(0)*15,RND(0)*15:SOUND 1,RND(0)*250,10,10:
NEXT W

```

Notes

- Line 10 Select mode, set up loops, and choose colour of points and lines.
- Line 20 Select values of top left corner of each area.
- Line 30 Plot the bottom right-hand corner.
- Line 40 Draw the right-hand side.
- Line 50 Draw the top.
- Line 60 Use POSITION to select the bottom left corner.
- Line 70 POKE into register 765 the colour of the paint used in filling—normally this is the same colour as the lines already drawn.
- Line 80 The fill statement.
- Line 90 Random beep and delay.
- Line 100 End of loops.
- Line 110 Flashy close.

The steps used to set up the XIO 18... statement are as follows, then:

1. PLOT the lower right-hand corner of the area.
2. Use DRAWTO to draw the line from there to the top right-hand corner.

3. In the same way draw the line at the top of the figure, from top right to top left.
4. Define the bottom left-hand corner with POSITION.
5. POKE 765 with the colour number to be used.
6. Use the XIO 18 statement.

The reason I used Mode 11 in the program is because it lets you have sixteen colours on screen at once. If you have to use other modes, or wish to explore them, you will need to change the program as follows:

1. Select the right range of colours with the COLOR statement.
2. Select the right range of X and Y values to ensure you don't go off screen.

COLOR and SETCOLOR work in the usual ways. It's worth noting that with this routine you can also fill triangles by making two adjacent corners very close on screen. More advanced programmers develop routines allowing much more complex shapes to be filled.

MODES 9-11

Like Mode 8, Modes 9-11 are not available on the older Ataris. If you are lucky enough to have them in your machine, you can extend your graphics programming quite a lot. None of these Modes have a text window—they offer pure graphics—and Figure 16.2 shows the X Y layout.

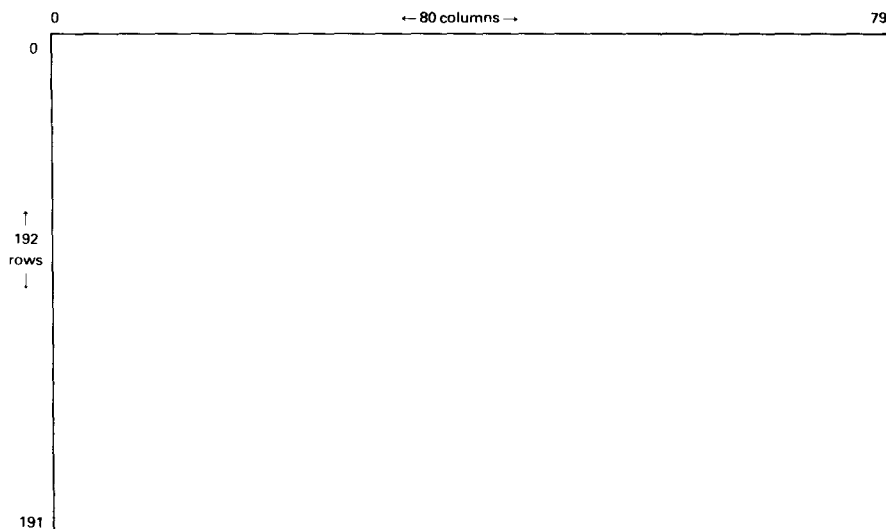


Figure 16.2 Modes 9 to 11 screen layout.

How strange! Here we have Modes in which there are more horizontal rows than vertical columns! This means that the shape of each screen site is very long and thin. I've put the basic details of these modes in the table on Page 44.

In Mode 9 you have a choice of one colour only, but then you can use any of the sixteen shades of that colour on the screen at one time.

In Mode 10 you can have eight colours on screen at once, with any brightness at any time. You can also control the colour of the 'paper'.

In Mode 11 you are allowed sixteen colours on screen at once. This time, though, you cannot control the brightness; all colours have the same brightness at the same time.

Program 50 Drapery, will give you some idea of the power of Mode 11. When you have played with it it is easy to explore Mode 9, because all you need to do is to change the 11

to 9 in line 10, and add a SETCOLOR 4, X, 0 statement to let you select colour X. (You may find you need to adjust your TV set with some care to get a good screen image in Mode 9. Turn the colour control down at least.)

Program 50: Drapery

```
10 GRAPHICS 11:LET DRAW=300
20 FOR BAND=0 TO 11:SOUND 1,(BAND+1)*20,10,10
30 LET Y1=BAND*16:LET Y2=BAND*16:GOSUB DRAW
40 NEXT BAND
50 FOR BAND=0 TO 11:SOUND 2,BAND+1*20,10,10
60 LET Y1=BAND*16:LET Y2=0:GOSUB DRAW
70 NEXT BAND
80 FOR BAND=0 TO 11:SOUND 3,BAND+1*20,10,10
90 LET Y1=0:LET Y2=BAND*16:GOSUB DRAW
100 NEXT BAND
110 FOR W=0 TO 1 STEP 0:NEXT W
299 REM ** Sub-routine 'BAND'
300 FOR COLOR=0 TO 15
301 REM ** The newer Ataris that support this mode also let you name variables w
ith key-words
310 COLOR COLOR
320 PLOT 0,COLOR+Y1:DRAWTO 79,COLOR+Y2
330 NEXT COLOR
340 RETURN
```

Mode 10 is not so easy to program, because you need to use POKE to get the colour you want into store. There are eight colour registers involved here: 705–712. Each can contain any number between 0 and 255, and that number stands for one of the 256 possible Atari colour numbers/brightnesses. You then use COLOR..., with a value between 1 and 8 inclusive to access each of those registers. Magic mountain, Program 51, shows this effect delightfully.

Program 51: Magic mountain

```
10 GRAPHICS 10
20 FOR GD=0 TO 1 STEP 0
30 FOR REG=705 TO 712:POKE REG,INT(255*RND(0)):NEXT REG
40 FOR RAD=1 TO 12:COLOR INT(7*RND(0))+1:PLOT 39,0:DRAWTO RAD*6,191:SOUND 1,RND(
0)*255,10,10:NEXT RAD
50 NEXT GD
```

You can of course still use SETCOLOR in these modes. I'll leave you to explore what to do. Just one note, however—in Mode 10 the five colour registers used with SETCOLOR (0–4) are the same as the POKE registers we've just met, numbered 708–712.

The Atari micros that came out in 1983 and 1984 also offer Modes 12–15. The Table on Page 44 gives you the basic data—there are no problems in dealing with any of these in the standard ways.

SUMMARY

In this chapter I have been concerned with two main topics—the drawing of graphs by PRINT and by PLOT methods, and the use of the graphics modes above number eight. Here's the usual list of keywords explored:

COLOR (C.)	POKE
DIM	PRINT (PR.)
DRAWTO (DR.)	SETCOLOR (SE.)
PLOT (PL.)	XIO
PEEK	

DO IT YOURSELF

1. Choose the programs in this chapter worth developing for your own use. Develop them to include mugtraps, suitable messages, and so on.
2. Write a program to produce high-resolution graphics random patterns on screen. Use PLOT and DRAWTO, and *really* explore COLOR and SETCOLOR. Make sure you know, in particular, how to ensure that the micro never tries to work outside the display window.
3. Devise a 'measles' program (one that produces dots at random in the display area), in which the dots appear only within 15 blocks on the screen. There should be three rows of five such blocks, the size depending on the mode you choose to use. Add sound.
4. Devise a program to give relevant histogram displays for your own use.
5. Devise a program to give relevant pictogram displays like that in Figure 16.1.
6. Develop a 'computer-assisted design' type of program as follows. Decide the shape of a room, and get the micro to draw it for you. Then call up from memory symbols for doors and windows and insert them where you want in the plan. Similarly furniture shapes can be used to plan the furniture layout of the room. If you are really ambitious, try to get the furniture and aperture units available in different sizes and colours. You will need to use the CONTROL graphics blocks in this program. (An even more ambitious touch would be to allow any arrangement to be analysed ergonomically, so that the 'best' layout can be found.)
7. Modify Program 44 to let the routine cycle through a large range of colour (using SETCOLOR) and sound variations. Also try other modes.
8. Add a subroutine to allow the user to correct data entered in either of the line graph programs. (You will note, perhaps, that the usual editing feature does not work with this table of input values, even if it looks as if it does.)
9. Explore the XIO "fill" routine to your heart's content.
10. Use Mode 11 in an attempt to program your Atari to produce the display shown in recent advertisements, of several pretty shaded 'cylinders' on screen.

17 Graphic Descriptions

The first thing I'd like to deal with in this chapter is what Atari call 'player-missile graphics'. Charming name! We shall do no more than tip our toes into this water—fast-action 'arcade game' programming needs you to work in machine code rather than in BASIC, but at least I'll tell you some of the BASIC ideas to let you play. Even in BASIC, we need to do quite a lot with PEEK and POKE, which are fairly close to machine code instructions anyway.

I'll throw you in at the deep end with a simple game that uses these features. I shan't use the words 'player' and 'missile' anymore—too belligerent for me! Instead I'll use the word 'sprites', which is, in any case, the name used with most micros that have the feature.

SPRITELY WORK

Here's a pretty simple game. Simple or not, it is pleasant to see and to use, and shows you what this sprite thing is about. You are the Commander of a space ship which appears on screen at the right-hand side of a star-field. Your home base is like a yellow Greek temple at the left-hand side. Your mission is to guide your ship between the stars to come to rest in the base.

The game *is* simple—no one will shout at you if you crash into a star (I'll let you keep your own sort of score). No aliens will dart from hiding to attack you. Just admire the visual effects, especially the way the rocket can move 'behind' stars (which in this game is cheating). Steer the ship using the four cursor-control keys (you don't need to press CONTROL), and tell the micro you've arrived in the base by pressing RETURN.

Program 52: Enterprise

```
10 GRAPHICS 2:POKE 755,0:SETCOLOR 2,9,0:SETCOLOR 4,9,0:PRINT CHR$(125):GOSUB 100
0:LET X=210:LET Y=58
20 LET R=PEEK(106)-8:LET B=256*R:POKE 54279,R
30 POKE 53277,3:POKE 559,46:POKE 53256,1:POKE 53240,X:FOR A=512 TO 640:POKE A+B,
0:NEXT A:POKE 704,50
40 FOR A=Y TO Y+4:READ L:POKE A+B+512,L:NEXT A:DATA 18,63,255,63,18
50 SOUND 0,250,0,10:OPEN #1,4,0,"K"
60 LET CT=0:FOR GO=0 TO 1 STEP 0:GET #1,C
70 IF C=43 THEN POKE 53240,X:LET X=X-1
80 IF C=42 THEN POKE 53240,X:LET X=X+1
90 IF C=45 THEN FOR A=0 TO 6:POKE A+B+Y+511,PEEK(A+B+Y+512):NEXT A:LET Y=Y-1
100 IF C=61 THEN FOR A=6 TO 0 STEP -1:POKE A+B+Y+512,PEEK(A+B+Y+511):NEXT A:LET
Y=Y+1
110 LET CT=CT+1:IF C=155 THEN LET GO=2
120 NEXT GO
130 FOR WL=1 TO 50:POSITION RND(0)*11,RND(0)*11:PRINT #6;"welcome":SOUND 1,250-R
ND(0)*2*W+2*W,10,10:NEXT WL
```

```

140 PRINT "You made it - in ";CT;" moves...."
150 FOR W=0 TO 1 STEP 0: SOUND INT(RND(0)*4),RND(0)*255,RND(0)*15,RND(0)*15:SETCO
LOR 4,RND(0)*15,RND(0)*15:NEXT W
1000 FOR STAR=1 TO 50: POSITION RND(0)*19,RND(0)*11:PRINT #6;"*":NEXT STAR:POSITI
ON 0,5:PRINT #6;"i":RETURN

```

Sure, you've seen better games for the Atari. Still, they do cost several times as much as the whole of this book, and don't teach you to program at the same time.

What the Atari 'player-missile' (sprite) system gives you is the chance to program up to five special characters, the 'players', which can have their own colours and move independently of each other under user or program control, around the screen. What the system does is to put the design of the sprite into a block in memory which then copies straight on to the screen, whatever's there already. To move the sprite around, the pattern is moved around in memory and copied on to screen all the time.

Getting it into the memory, and moving it around in there, is not simple. And that is where all those PEEKs and POKEs come in. Program 52 shows that clearly.

The effect is that there are really a number of separate screens on top of each other on your TV set. You can program the order of these layers, so that one object can disappear behind another but in front of a third, and so on. I didn't even try to do that in Program 52, but you got the idea. The spaceship moves behind the printed characters but over the 'paper'. And all this is independent of the Mode you are in, and the material you put on screen using that mode. I use Mode 2, simply because I wanted to use PRINT# 6... to get characters on there rather than making up more of my own. Now I shall go through Program 52 pretty well line by line. I hope you can follow! There'll be a summary after the notes.

- Line 10 Set up Mode 2 screen, remove cursor, select 'paper' colour, select 'border' colour, clear screen. Use the subroutine starting at line 100, choose the starting values of X and Y, the coordinates for the sprite. The sprite screen is about 240 pixels across and 120 down.
- Line 20 Use PEEK to look into memory at site 106. This site gives the value of RAMTOP, and thus measures the amount of memory available for use in the machine on which the program runs. The other two statements on the line use that value to reserve an area of memory for the sprite image (pattern).
- Line 30 POKE 53277,3 tells the micro that sprite graphics is to be used. Putting a zero at that address has the reverse effect.
POKE 559,46 tells the micro that the sprites are to have 'double' resolution. You can use 62 instead of 46 to get 'single' resolution.
POKE 53256,1 tells the micro to make the sprite in question double length. 0 gives normal length, and 3 gives quadruple length.
POKE 53248,X maps the starting X-position of the sprite into the sprite block of memory.
The rest of line 30 clears out any 'garbage' from the block of memory concerned.
- Line 40 This uses READ...DATA (see next section). The structure of each line of pixels in the sprite is entered into memory at the current Y-position. Different numbers in the DATA statement lead to different defined characters on screen. I'll come on to this in a moment.
- Line 50 'Spaceship' sound—prepare keyboard for direct RETURN-less entry.
- Line 60 Set the count of moves to zero, set up the main control loop, wait for a key-press.
- Line 70 If the left-cursor key is pressed the X-position details change.
- Line 80 The same applies for the right-cursor key.
- Line 90 The same applies, but more complexly, for the up-cursor key.
- Line 100 And for the down-cursor key.
- Line 110 Add 1 to the value of the move count; check for RETURN key-press.
- Line 120 End of main loop.
- Line 130 Welcome loop, print "WELCOME" in yellow at random positions on screen with a related sound effect.
- Line 140 Put final message into text window.

Line 150 Closing sound/flash routine.
Line 1000 Print asterisks at 50 random sites on the screen.

To be logical, I ought next to deal with the routine for defining your own shape to go into the sprite system. However, please allow me to leave that to the next section, so that I can close this one with a summary of the POKE addresses for use with the four main sprites.

Table 17.1 Player/missile addresses

Addresses	Contents
704–707	Colours of sprites 0–3
53248–53251	X-positions of main sprites ('players') 0–3
53252–53255	X-positions of minor sprites ('missiles') 0–3
53256–53259	horizontal length of sprites 0–3
53260	horizontal length of all minor sprites

In conjunction with the notes above, POKE the values you want into these storage sites.

Before ending this section, I say again—sprite graphics with the Atari is incredibly slow in BASIC. You found that out with Program 52. The effect is worse the more sprites you try to control. Only machine code routines can give you real speed.

DIY CHARACTERS

The sprite we used in Program 52 consisted of five rows of eight pixels. Recall, though, that I used double length, making the thing appear sixteen pixels long on screen.

In certain circumstances you can define characters which consist of more than five rows of eight pixels. In this section I'll tell you how to do the simple ones—just the same idea applies in all cases. When we did this in Program 52, we had to use the READ... DATA structure. I'd best go over that in more detail now.

READ...DATA

When the micro meets READ (in a program, or as a command), it looks for DATA (stored in memory with a program line). It then assigns what it finds to the variable named after READ. READ, therefore, is yet another 'assignment' statement (like LET and INPUT).

So READ Y: PRINT Y will give output 3.14 if you already have it in memory a line number followed by DATA 3.14. Try it. Try it again. This time it doesn't work. You get an ERROR message, number 6, which means 'out of DATA'.

What happens is that when a DATA item has been used by a READ, it is sort of ticked off in the data list. The next READ ignores it and looks for the next DATA item. If there is no such item, then we get that warning message.

READ and DATA can be used with either numeric or string material, or a mixture of both. You *must*, however, make sure that all data is listed in the right order in the DATA line(s), so that the READ statements find the right data to work on. Enter and RUN the little program that follows.

```
10 LET HI = 10
20 DIM D$(2)
```

```

30 DATA 5,7,9
40 DATA HI
50 READ A,B,C,D$
60 PRINT A,B,C,D$

```

When you run this, no problem. On screen 5,7,9, HI appear, nicely spaced out. Try it again, but this time don't type RUN, rather restart using GO TO 30.

At once we get ERROR 6 noted for line 50. This means that the computer has run out of DATA to READ; all the DATA items have been ticked off. To untick those data items, you could use RUN. There is another keyword which has the same sort of effect—it is RESTORE. Try it:

```
RESTORE (R): GO TO 30
```

The data list appears nicely on screen again.

RESTORE can be used with a line number after it, telling the micro to untick only from the start of that line. In this case, for instance, we could use RESTORE 40, unticking the DATA in line 40, but leaving that in line 30 alone. If we do that and then use GO TO 30 once more, a new ERROR message appears. ERROR 8 means that the micro has tried to READ a number, but found that the first unticked DATA item available is a string.

In this case, the micro tried to read the DATA in line 40. That is the characters HI. This looks like a number, rather than a string, not just because we have no speech marks round it, but also because of line 10. But HI is *not* a number—it is a string.

Indeed, if you put speech marks around string constants in DATA statements, you'll find the Atari doesn't like it. Use the editing feature to put speech marks round the HI in line 40 in the above program. And RUN to see what happens. The print out on screen of the value of D\$ is not HI, but "H". Because of line 20 the length of D\$ is kept to two characters—the first two characters found by the READ are the first speech mark and H. (We get the same sort of interesting effect if you put speech marks around a string entered in an INPUT statement. In both cases, the 'feature' can lead to problems when you try to compare strings.)

You can put your DATA statements anywhere in the program listing you like, at the start, after the corresponding READ statements, at the end, or even after an END statement. All the micro wants is that they are there somewhere with a line number in front. In the same way, in some programs you may find that you need to have several READ statements at different points. No problem—as long as there's enough DATA around, the READs will find and assign values.

There are two main uses of READ...DATA. The first is with non-interactive programs, in which there is no user to INPUT data in response to questions. The second is for a recorded program which you want to carry its own data, and in which LET may be too clumsy. In summary, then:

1. READ VARIABLE(\$) assigns the next free DATA item (if suitable) to the variable and 'ticks off' the one used. It can be a command or a statement.
2. DATA carries the stored DATA items, separated by commas if there's more than one, and without quotes if they are strings. DATA statements must be entered with program lines (even if the program is not run) and can appear just anywhere in the listing without affecting its operation.
3. RESTORE unticks all the DATA items so the next READ will take the first one.
4. RESTORE n unticks all the DATA items starting at line n.

Earlier I gave the two main uses for the READ...DATA...RESTORE structure. A non-main use, but the one that most people find first, is that used in Program 52—to carry the DATA needed for forming your own characters.

To define your own shapes is a bit lengthy. Be warned! But it's worth it. It's only a pity that, once defined, those shapes can't be used easily in programs.

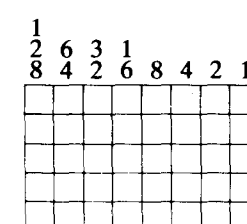
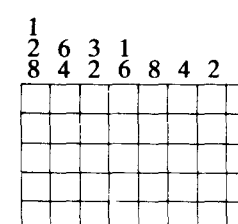
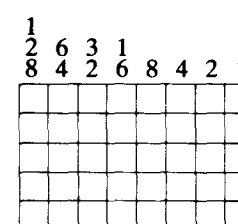
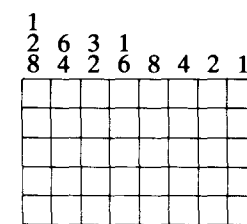
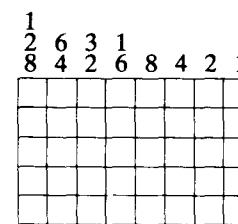
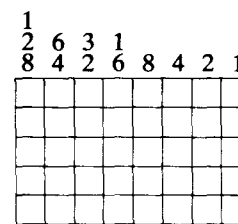
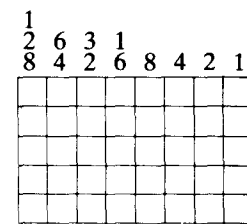
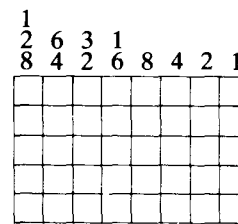
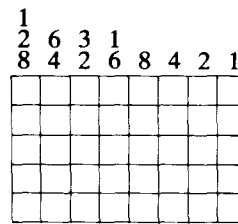
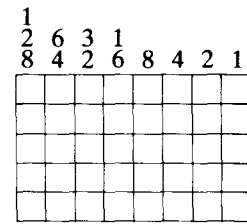
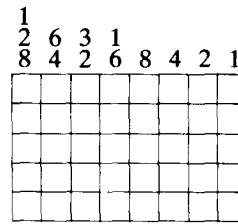
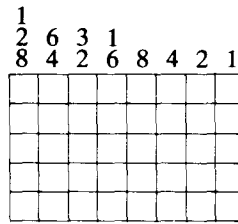
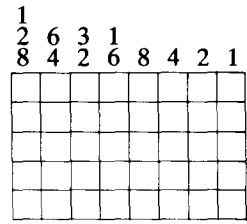
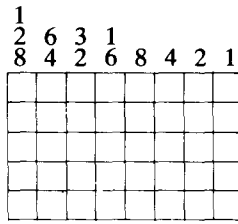
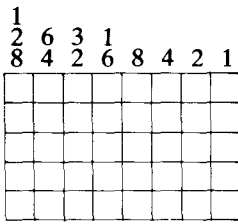


Figure 17.1

1. You need to sketch your shape on a 5×8 grid. (Or on a grid of more lines than 5 in those cases where you need it.) Use old graph paper, or pages from a square ruled exercise book, or make copies of the grids in Figure 17.1.
2. Sketch your design faintly on the 5×8 grid. Let's say we want a little fish—I give the sketch in Figure 17.2.

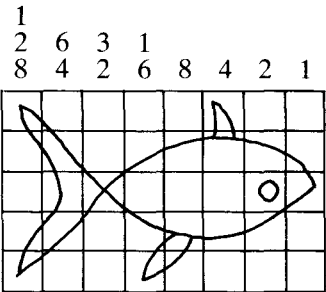


Figure 17.2

3. Reproduce the shape as close to that sketch as you can by blacking in the right squares of the grid. Figure 17.3 is my attempt.

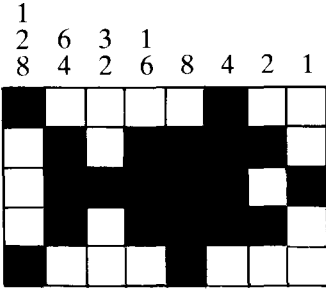


Figure 17.3

4. Now—the tricky bit. Refer to Figure 17.4 to make it simpler. . . . For each of the five lines of eight squares in turn—write down the total of the numbers at the tops of the columns containing blacked-in squares. First line: black in 128 and 4, gives 132; second line: black 64, 16, 8, 4, and 2—gives 94; and so on. I guess you'd best check my adding up! Enter the five numbers you get into a DATA statement in your program. If you've still got Program 52 around you can put these figures into it in line 40 instead of the rocket ship ones. The DATA line is DATA 132, 94, 125, 94, 136.

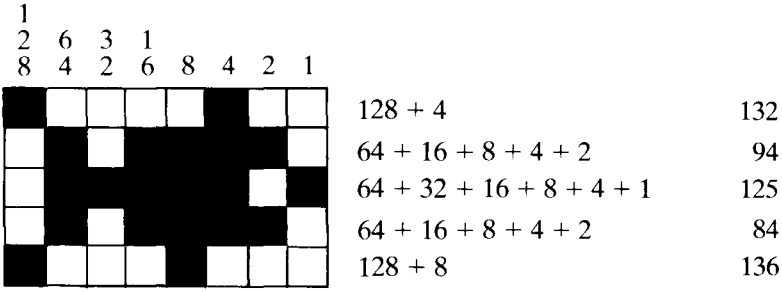


Figure 17.4

Now the graphic design world is yours! Try your own shapes, putting the resulting five numbers into a suitable DATA statement in a suitable program. The sketch in Figure 17.5, and the DATA line beside it, may be a space invader. Atari, watch out!

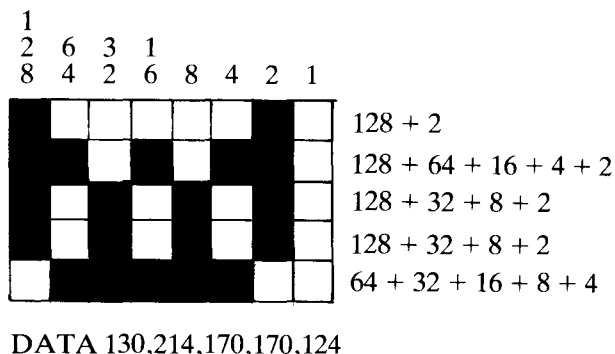


Figure 17.5

Advanced programmers are able to string designs together to make large and complex defined shapes. First, I think, you should practise getting on top of this 5×8 system. After all, don't forget you can have double or quadruple length sprites, so that gives you a fair degree of flexibility.

To cheer you up, try the closing program. This will give you just a hint of what can be done with defining characters (this time 8×8 ones). I shall give only very brief notes—the idea is enough to start you off in this direction if you so wish.

Program 53: Face of the tiger

```

10 DIM CHR(2):GRAPHICS 2:POKE 755,0:SETCOLOR 1,4,8:SETCOLOR 2,7,2:SETCOLOR 4,7,2
20 FOR FACE=1 TO 2:LET CHR(FACE)=(PEEK(742)-FACE*4)*256
30 FOR L=CHR(FACE) TO CHR(FACE)+7:READ A:POKE L,A:NEXT L
40 NEXT FACE
50 FOR GO=0 TO 1 STEP 0
60 FOR FACE=1 TO 2:POKE 756,INT(CHR(FACE)/256):SOUND 1,400-FACE*100,10,10:FOR W=
1 TO 500:NEXT W:POSITION 7,5
70 POKE 756,226:IF FACE=1 THEN PRINT #6;"FROWN"
80 IF FACE=2 THEN PRINT "          s m i l e":PRINT #6;"SMILE"
90 FOR W=1 TO 500:NEXT W
100 SOUND 1,0,0,0
110 NEXT FACE:NEXT GO
1000 DATA 60,126,90,254,218,102,60,00
1010 DATA 60,126,90,254,230,90,60,00

```

- Line 10 Make space for the CHR-array, select Mode 2, delete cursor, arrange colours.
- Line 20 Arrange starting points in memory for the sets of data representing the two characters.
- Line 30 In each case, take the eight sets of data from the DATA statements and POKE them into the correct part of memory.
- Line 60 POKE into storage site 756 (the character set select one) each face in turn.
- Line 70 POKE into the same storage site the 'alternate memory' in readiness for the 'normal' printing.
- Line 1000/1010 DATA statements, two sets of eight bytes for the two sets of eight lines of the two characters. Note that these DATA statements appear after the end of the program and are never involved in a RUN.

SUMMARY

In this chapter I've been able to do no more than give you a little taste of two more advanced graphics techniques. These are the use of sprites ('player/missile' graphics) and defining your own characters. The work needs a lot of POKEs, and quite a few PEEKs, putting us on the fringe of machine coding. The chance was also taken to give the details of READ...DATA...RESTORE.

No formal 'Do it yourself' section this time; only *you* can decide how much to explore these ideas.

18 Ragbag

In this chapter I would like to look at one or two matters not dealt with elsewhere. I shall also note one important area for the advanced programmer that a basic book like this cannot attempt to touch in depth. It is interfacing the computer with other equipment.

TIMING

In Chapter 2 I briefly described the internal structure of computers like the Ataris. One part of the central processing unit that I did *not* mention is the 'clock'. The clock is in practice very important indeed—the action of a CPU is complex, and it is essential that all the different parts work in step. Thus data must be transferred at the right moment, and the different sections of the system must act on it correctly.

The micro's internal clock ticks very fast—carrying out a cycle of processing in about a millionth of a second. Using BASIC there's no way we can get at that clock. Anyway, even if we could, it would not be easy to use. However, there is a little chunk of memory that holds some kind of record of how many cycles the CPU has gone through since you switched on. We *can* access those storage sites, and we can then use them to tell the time in some kind of way.

The addresses of the three storage sites concerned are 18, 19, and 20. When you switch the micro on, each takes a 0 value. Each fiftieth of a second (sixtieth in North America) after that, 1 is added to the value in site 20. When that reaches the value 255, the micro adds 1 to the value in site 19 and the value in site 20 goes back to 0. This goes on until the value in site 19 reaches 255, when the micro adds 1 to the contents of site 18 and both 19 and 20 take 0 again. If you've ever seen a 'tally-counter', this is a bit like an electronic one, 'incrementing' its tally each fiftieth of a second.

Program 54 consists mainly of two subroutines that you can use in all kinds of programs. I've put them at the start of the program because, as explained on Page 125 when the program calls a subroutine the micro starts hunting for it at the first line. Putting the subroutines that matter at the start of the program means that it won't take the micro more than a few cycles to find what it wants—thus in timing it can be more accurate.

The crucial lines of the two subroutines, lines 20 and 50, are just the same. What they do in each case is to give to the corresponding T a value that represents the number of seconds since switch-on. The value SEC then takes the difference, thus giving the time elapsed between the two subroutine calls. Those two subroutine calls are in lines 110 and 120 of the main part of the program. What you do when you RUN this program, is to press RETURN when you want timing to start, and press it again when you want timing to stop. The program closes by giving you the number of seconds between those two RETURN-presses. The INT structures in the two timing lines allow this result to be correct to one decimal place.

Program 54: BASIC timer

```
10 GOTO 100
11 REM ** Sub-routines at start to save micro time
12 REM ** Start timing - 'TON'
20 LET T0=INT(((PEEK(18)*65536+PEEK(19)*256+PEEK(20))/5+0.5)/10)
30 RETURN
42 REM ** Stop timing - 'TOFF'
50 LET T1=INT(((PEEK(18)*65536+PEEK(19)*256+PEEK(20))/5+0.5)/10)
60 LET SEC=T1-T0
70 RETURN
99 REM ** Main program starts
100 GRAPHICS 2:DIM G$(1):LET TON=20:LET TOFF=50
110 FOR GO=1 TO 5
120 INPUT G$:GOSUB TON
130 INPUT G$:GOSUB TOFF
140 PRINT #6;"      Time elapsed was ";SEC;" seconds."
150 NEXT GO
160 PRINT "      RUN and RETURN to start again!":END
```

You can use this program as a simple reaction timer if you wish, but in fact there's a better one in Appendix 2. Of course there are many ways in which you can use this useful feature. You may wish to try to set up a little digital clock that runs in the top corner of your screen throughout your program. That will need a routine for converting to minutes and seconds, or even hours, minutes and seconds if your program runs a long time. It's also easy enough to put timing into games or 'twenty questions' programs.

If you're sharp you'll have realized that the system can lead to error if, between TON and TOFF, register 18 goes back to zero. Well, you are right, it could happen, but it's going to be just over 93 hours after switch on. Not too much to worry about, is it?

INTERFACING

Here we really think of using the micro to capture information from outside devices and/or to control outside devices. Examples of the two types are photocells, electronic thermometers, and roof-top radar; and lamps, motors, and nuclear power stations.

If this kind of work is likely to become your scene, you'll have to search the magazine adverts for suitable 'interfaces'—'black boxes' that plug into the Atari and in turn have the other devices plugged into them. This is not something that we can discuss in this book. But here's a list of possible uses that may whet your appetite. . . .

- Controlling tape/slide equipment from an interactive program;
- Burglar alarms and decoys;
- Flashing light displays for your disco;
- Central heating control with an array of sensors.

We are in fact closer to the first with the Atari than with most other micros. You may know that your program tape has a spare track on it on which audio material can go. With a great deal of effort, therefore, it is possible to have the cassette contain a recorded program for loading into the computer, which when RUN, thereafter controls the cassette machine to allow the playback of a soundtrack in time with the program action. To do that is a lot of hard work, so you'll have to look for magazine articles that describe it.

The main interfacing needs of Atari users, however, involve letting their micro interact with disc units, printers, and on an even simpler level, joysticks and paddle controls.

If you are rich enough to be able to afford a *disc drive*, you have a fair choice of models which will work with your micro. Each lets you have access to large amounts of information at high speed. Not only can you find the programs or data that you require very fast, but they can be got into the machine quickly too. If you use the Atari a lot, or if your use involves transfer, processing, changing, and storage of large amounts of data, a disc drive is a good buy.

This is not, however, the place to give details of how to use a disc drive. Most likely by the time you get round to getting one, your command of BASIC will allow you to pick up the new features and structures very soon.

The same sort of comment applies to the use of a *printer*. Here too there are many machines on the market, and the Atari micros can (on the whole) be used with a good range of them.

Again it is not likely that you will need to use a printer soon after you have got your Atari. The main uses are if you wish to produce lots of clear and unbugged program listings—as is the case with this book—or when you are running software for data-handling or text-editing purposes.

More needs to be said, however, about using the Atari with a joystick or paddle. This is because you are perhaps more likely to get one or other of these with your new micro, or soon after, and because you may well find there are few instructions for their use. Figure 18.1 shows these devices with a micro. There are many kinds of joystick; I shall describe the simplest type, the sort that the Atari is easy to program for.

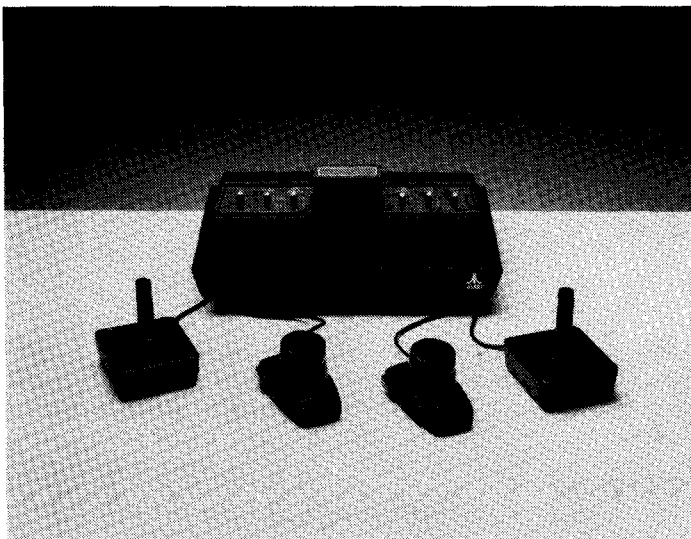


Figure 18.1 Atari joysticks and paddles.

I guess you know what a joystick is. The name comes from the control lever of an aircraft (and I'd love to know why airmen called it that). It consists of a lever in a box, with a circuit that allows a signal to be output that depends on the way the stick is pushed. There is also a trigger ('fire-button') that is simply off or on.

The micro checks the state of the joystick or joysticks that may be connected at the time. It does so 50 times a second (in the United States the figure is 60 times a second). The state of each of the four joysticks you can use shows as a code in the four storage sites. You as a programmer can then access each of those storage sites by using a special joystick function. The function is `STICK(N)` where `N` is 0–3 depending on which joystick you want to read. (Early Ataris could live with four joysticks, modern ones are for users with only two hands.)

Figure 18.2 shows the different ways you can push the joystick and the codes that go into the corresponding storage site. I'll come back to the trigger in a moment. Thus if joystick number one is not in use, or not even connected to the machine, the command `PRINT STICK(0)` returns the value 15. If you connect the joystick and push it forward the value becomes 14. And so on.

A similar function allows you to read the state of the trigger. The function is `STRIG(N)` where again `N` stands for the joystick concerned. The resting (off) value for `STRIG` is 1; when you press the button, the value becomes 0.

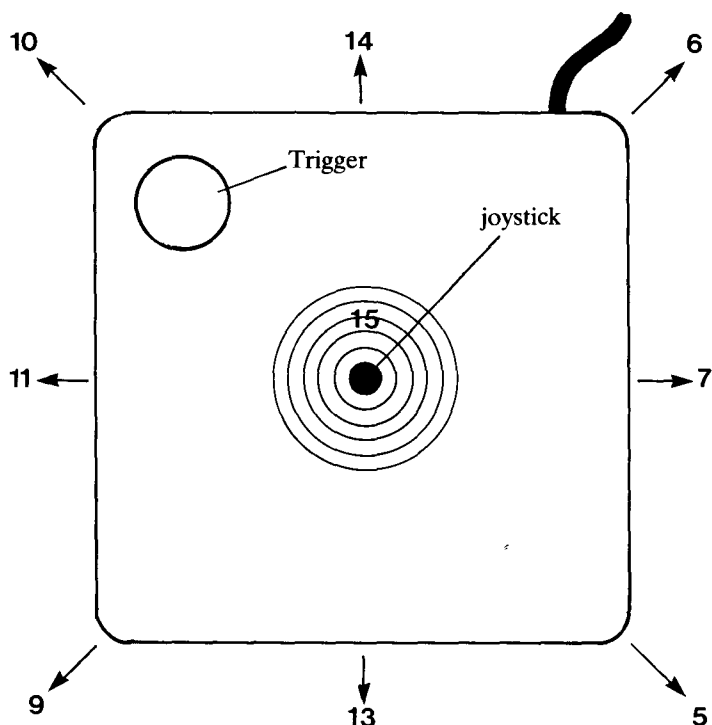


Figure 18.2

Program 55 shows how we can use these two functions. It should be enough to give you an idea of how to program for joysticks. Once you've got that idea I suggest you look at some of the programs in this book and convert them to joystick control rather than keyboard control.

Program 55: Joy forever

```

10 DIM A$(10):GRAPHICS 2
20 FOR G0=0 TO 1 STEP 0
30 PRINT #6:CHR$(125):POSITION 2,5
40 RESTORE 200+RND(0)*8
41 REM ** Note this useful trick; 'Random RESTORE', Appendix 6
50 READ A$,A
51 REM ** Put direction and corresponding STICK value into A$ and A
60 PRINT #6:A$:PRINT "Push stick ";A$;"..."
70 LET FLAG=1:FOR G=1 TO 500:LET S=STICK(0)
71 REM ** FLAG becomes zero when stick moved or button pressed
80 IF S<15 THEN LET G=1001:GOSUB 1000
81 REM ** To sub-routine and out of loop when action taken
90 NEXT G
100 IF FLAG THEN PRINT "Too late!"
110 FOR W=1 TO 500:NEXT W
120 NEXT G0
201 DATA North,14
202 DATA North-east,6
203 DATA East,7
204 DATA South-east,5
205 DATA South,13
206 DATA South-west,9
207 DATA West,11
208 DATA North-west,10
1000 IF STRIG=0 THEN STOP

```

```

1010 LET FLAG=0:IF S=A THEN PRINT "That's right.":RETURN
1020 PRINT "Nope....":RETURN
1021 REM ** Reached if wrong direction tried

```

This program is a simple test of compass directions. It tests whether you push the joystick in the direction stated or not within a couple of seconds. It goes on forever, until you press the trigger button. You can of course improve it in a number of ways in practice, but it's here as a simple demonstration of how to use the joystick functions in a program.

On purpose I didn't give as an example of joystick usage a games program. I would like to note here that the use of joysticks is important not just for gaming, but in many 'serious' areas too. A number of modern micros have what's called a 'mouse' or a 'golf-ball', by rolling which in different directions the user can control the screen display. There are many applications of this kind of effect in office software. And the joystick may be used in just the same kind of way.

For instance, certain programs, called 'spread-sheets', contain a vast array of many rows and columns of data for processing. The arrays are far too large to fit onto the screen, so the user must access the different parts in some way. Mice and joysticks are as good as each other in this context. One can also write a program with large 'menus' (tables of option choices), and then allow the user to select his or her option with the joystick. Perhaps it's a shame that Atari call the joystick and paddle 'game controllers'!

Paddle controls are a different kind of flexible input device. (The name comes from the North American word for table tennis racquet, and refers to the fact that the first video games—from Atari, of course!—were a version of ping-pong in which this kind of control was used.) Again the device has many uses outside gaming.

Coding for the paddle is very much the same as dealing with joysticks. However while a joystick gives eight separate readings (plus trigger output), the paddle has two hundred and twenty eight (plus trigger). The main use of joysticks is to give one of eight choices (I now ignore the trigger), but that does not mean that you use paddles for giving one of 228 choices! Not as such, anyway. The best use of this device is to allow the user to control something that can vary continuously from one extreme to the other.

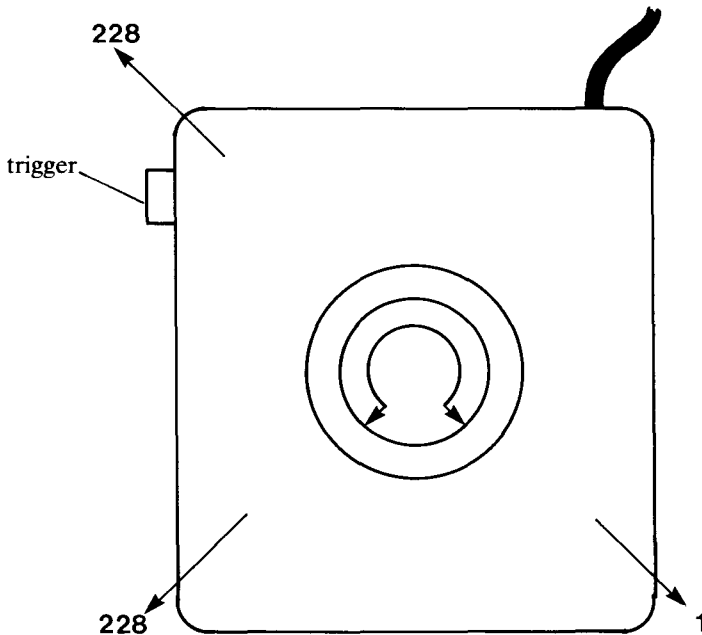


Figure 18.3

Sound, for instance. Program 56 very simply shows how to use the two paddle keywords: PADDLE(N) and PTRIG(N). Here again, N is the code for the paddle to be looked at. I don't think there's any problem here.

Program 56: Up the creek

```
10 FOR G0=0 TO 1 STEP 0
20 LET P=PADDLE(0):REM ** Read first paddle
30 SOUND 1,P,10,10
40 IF PTRIG(0)=0 THEN SOUND 2,P,10,10:REM ** Read first paddle trigger
41 REM ** Or you can use the TRIG to get out with....
50 NEXT G0
```

PADDLE(N) gives an output of 228 if the paddle approached is not connected or has the control turned all the way anti-clockwise. The paddle can go about three quarters of the way round, but you will find that a third or so of that makes no difference to the output—it is not used. See Figure 18.3. PTRIG(N) gives 1 if the trigger is left at peace, but 0 if pressed. Isn't it a shame no one uses paddles any more?

THE SPECIAL FUNCTION KEYS

These are the set over the right of the keyboard, four or five in number depending on the Atari you have. We've dealt with the (SYSTEM)RESET key already (Page 93 for instance). Here I want us to look at the OPTION, SELECT, and START keys that all Atari micros offer.

Every fiftieth of a second, your micro checks to see which, if any, of these keys are pressed. Storage site number 53279 carries a value that depends on the keys used. The table shows the readings in that storage site and you can access it at any time with PEEK(53279).

Table 18.1 PEEKing 53279

Key(s) down	Stored value
None	7
OPTION	3
SELECT	5
START	6
OPTION and SELECT	1
OPTION and START	2
SELECT and START	4
All three	0

If you have an XL Atari, you may have tried the 'Help' routine you get by pressing BYE or by switching on with the OPTION key down. My program 'You takes your choice', is a pale imitation of that. Study it to get some idea of how to read these special keys and how to use the results.

Program 57: You takes your choice

```
10 GRAPHICS 2:FOR W=0 TO 1 STEP 0
20 POSITION 5,5:PRINT #6;"START key"
30 IF PEEK(53279)=6 THEN LET W=2
40 POSITION 5,5:PRINT #6;"      "
50 NEXT W
51 REM ** All above is START routine
60 POSITION 6,0:PRINT #6;"M E N U"
70 PRINT #6:PRINT #6;"1.  this":PRINT #6:PRINT #6;"2.  that":PRINT #6:PRINT #6;"
3.  the other"
```

```

80 PRINT "OPTION key to change line.":PRINT "SELECT key to show choice."
90 LET L=2
100 FOR W=0 TO 1 STEP 0
110 POSITION 16,L:PRINT #6;"<"
120 IF PEEK(53279)=3 THEN POSITION 16,L:PRINT #6;" ":FOR D=1 TO 100:NEXT D:LET L
=L+2:IF L=8 THEN LET L=2
130 IF PEEK(53279)=5 THEN LET W=2
140 NEXT W
150 GRAPHICS 2:GOSUB L+500
151 REM ** All above is OPTION/SELECT routine
160 RUN
161 REM ** Re-start after chosen program
999 REM ** First choice
1000 POSITION 7,5:PRINT #6;"THIS"
1010 FOR W=1 TO 1000:NEXT W
1020 RETURN
1999 REM ** Second choice
2000 POSITION 7,5:PRINT #6;"THAT"
2010 FOR W=1 TO 1000:NEXT W
2020 RETURN
2999 REM ** Third choice
3000 POSITION 5,5:PRINT #6;"THE OTHER"
3010 FOR W=1 TO 1000:NEXT W
3020 RETURN

```

I guess most uses of the special function keys involve your checking whether the user is pressing any one or other of those three (or none at all). You may have some uses for combination key-presses, if so, good luck to you. That last program will show you how to use the results.

The newer Atari micros also have a special HELP key. Its use in programs is much the same as the above—see Page 197.

19 Hip Hip Array

Well, we *can* cheer. This is my last chapter in this book, it deals with the last remaining aspect of BASIC. I've had to use its subject—arrays (the computer equivalent of lists and tables) in a number of programs already. That's because they are *so* useful.

To computer buffs, an array is a list or table of data stored as a block in memory in such a way as still to allow the individual members—called *elements*—to be accessed. Before using an array, we must enter a DIM (= dimension) statement to tell the micro to reserve the correct amount of storage space. As you know, COM will do instead of DIM.

A SIMPLE ARRAY

Let me build up a fairly simple data-handling program. It uses an array to store a number of numbers and lets that list be used by the user. The first part of the program, Program 58a, sets up the array as the user wants it, and then allows the user to input the elements. Let's have the (part)program before any more theory . . .

Program 58a: Entering an array

```
10 LET TEST=1000:DIM T$(4)
20 PRINT CHR$(125);POSITION 5,10:PRINT "How many numbers to store";
30 FOR I=0 TO 1 STEP 0:INPUT T$:GOSUB TEST:IF FLAG=0 THEN LET I=2
40 POSITION 30,10:PRINT " ":POSITION 30,10:NEXT I
50 LET T=VAL(T$):IF T<5 OR T>100 THEN PRINT :PRINT "Please try a more useful number!":FOR W=1 TO 750:NEXT W:RUN
60 DIM A(T):PRINT CHR$(125)
70 FOR B=1 TO T:PRINT "Give number ";B;" ":
80 FOR I=0 TO 1 STEP 0:INPUT T$:GOSUB TEST:IF FLAG=0 THEN LET I=2
90 NEXT I:LET A(B)=VAL(T$):NEXT B
990 STOP
999 REM ** Sub-routine TEST
1000 LET FLAG=0:IF T$="" THEN LET FLAG=1:RETURN
1010 FOR A=1 TO LEN(T$):IF T$(A,A)<"0" OR T$(A,A)>"9" THEN LET FLAG=1
1020 NEXT A:RETURN
```

Do enter that program into your micro now, but don't run it yet—there isn't any point so far!

Note that in this example—what we call a one-dimensional numeric array—I've blocked lists outside the limits of 5–100 elements. That's only for convenience. For the same reason too, the program accepts only whole number element values. You should easily be able to remove the first restriction if you want. But can you find and remove the second?

If you *do* run this program after you have entered it, the part of memory reserved for the array by the DIM statement will contain the T numbers. Remember that we call these the elements of the array—we write each in the form A(S) where we call S the

subscript of that element. Thus if the user inputs the value 7 to T, and the seven numbers entered, in order, are 5, 10, 15, 20, 25, 30, and 35, then A(1) = 5, A(2) = 10, A(3) = 15 . . . and A(7) = 35. Those numbers are the seven elements or members of array A, their subscripts are 1 to 7.

In that part-program I used a loop to let the user input the array elements. This is the normal approach—simple to set up, code and use. Please make sure you can understand what's going on before you look at the next bit. As I've said, there's not much point in having a micro set up an array and get the user to input the values of the elements, unless the data can be used.

SEARCH AND SORT

One important usage of data held in this way is to let the micro 'search' to check if a given value is held or not. Program 58b does this. Once you've entered it on top of 58a, you can RUN the combined program and explore it!

Program 58b: Searching the array

```
200 PRINT CHR$(125):SETCOLOR 1,0,0:SETCOLOR 2,6,12:SETCOLOR 4,6,12
210 POSITION 6,4:PRINT CHR$(253);"S E A R C H   R O U T I N E":PRINT "   =====
=====
220 POSITION 6,10:PRINT "What number to seek";
230 FOR I=0 TO 1 STEP 0:INPUT T$:GOSUB TEST:IF FLAG=0 THEN LET I=2
240 POSITION 25,10:PRINT "          ":POSITION 25,10:NEXT I
250 LET V=VAL(T$):POSITION 6,10:PRINT "Searching for ... ";V;" ... "
260 LET A$=" in the list.":LET FLAG=0:POSITION 6,10
270 FOR B=1 TO T:IF A(B)=V THEN PRINT V;A$:LET FLAG=1:LET B=B+1
280 NEXT B:IF NOT FLAG THEN PRINT V;" not";A$
290 FOR W=1 TO 1000:NEXT W:RUN
```

When you test this, don't forget that I've set the program up to accept only integer (whole number) element values for the array. And to save you too much time in testing, stick to small arrays of five or six elements only. Once we've got the full program, of course, you can try it out with large arrays and see how fast the Atari is in that context.

The micro can also sort the array we have set up very easily. Well, I mean, no problem for the micro. Coding isn't so simple.

What 'sorting' means is putting the array elements into a useful new order. The routine that does that starts at line 300 in the final full version, Program 58c. This also includes a menu (line 110) which uses the special function key routines I introduced to you on Page 172. I think the program is well worth entering, trying out, finishing off, and keeping.

Program 58c: Hip, Hip, Array

```
10 LET TEST=1000:DIM T$(4),A$(13)
20 PRINT CHR$(125):POSITION 5,10:PRINT "How many numbers to store";
30 FOR I=0 TO 1 STEP 0:INPUT T$:GOSUB TEST:IF FLAG=0 THEN LET I=2
40 POSITION 30,10:PRINT "          ":POSITION 30,10:NEXT I
50 LET T=VAL(T$):IF T<5 OR T>100 THEN PRINT :PRINT "Please try a more useful number!":FOR W=1 TO 750:NEXT W:RUN
60 DIM A(T):PRINT CHR$(125)
70 FOR B=1 TO T:PRINT "Give number ";B;"   ";
80 FOR I=0 TO 1 STEP 0:INPUT T$:GOSUB TEST:IF FLAG=0 THEN LET I=2
90 NEXT I:LET A(B)=VAL(T$):NEXT B
100 FOR W=1 TO 500:NEXT W:FOR M=0 TO 1 STEP 0
110 SETCOLOR 1,0,12:SETCOLOR 2,12,2:SETCOLOR 4,12,0:PRINT CHR$(253);CHR$(125):POSITION 13,4:PRINT "*** M E N U ***":POSITION 16,5:PRINT "-----"
120 POKE 82,8:POSITION 8,8:?"Search":?"?:?"Sort ascending":?"?:?"Sort descending":?"?:?"Enter new data":?"?:?"Start new list"
121 REM ** ? for PR. to save space
122 REM ** POKE 82.. to change margin
130 POKE 75,1:POKE 82,2:PRINT :PRINT :PRINT "Use SELECT and OPTION keys to choose."
```

```

140 LET X=31:LET Y=8:REM * Set start point of cursor
150 FOR W=0 TO 1 STEP 0:POSITION X,Y:PRINT "<":IF PEEK(53279)=3 THEN POSITION X,
Y:PRINT " ":LET Y=Y+2:IF Y=18 THEN LET Y=8
160 IF PEEK(53279)=5 THEN LET W=2
170 FOR D=1 TO 100:NEXT D:NEXT W
180 GOSUB (Y/2-2)*100:NEXT M
199 REM ** Search
200 PRINT CHR$(125):SETCOLOR 1,0,0:SETCOLOR 2,6,12:SETCOLOR 4,6,12
210 POSITION 6,4:PRINT CHR$(253);"S E A R C H " R O U T I N E":PRINT " =====
===== "
220 POSITION 6,10:PRINT "What number to seek";
230 FOR I=0 TO 1 STEP 0:INPUT T$:GOSUB TEST:IF FLAG=0 THEN LET I=2
240 POSITION 25,10:PRINT " ":POSITION 25,10:NEXT I
250 LET V=VAL(T$):POSITION 6,10:PRINT "Searching for ... ";V;" ... "
260 LET A$=" in the list.":LET FLAG=0:POSITION 6,18
270 FOR B=1 TO T:IF A(B)=V THEN PRINT V;A$:LET FLAG=1:LET B=B+1
280 NEXT B:IF NOT FLAG THEN PRINT V;" not";A$
290 FOR W=1 TO 1000:NEXT W:RETURN
291 REM ** Changed from last time
299 REM ** Upward sort
300 PRINT CHR$(125):SETCOLOR 1,0,0:SETCOLOR 2,6,12:SETCOLOR 4,6,12
310 POSITION 6,4:PRINT CHR$(253);"U P W A R D " S O R T I N G":PRINT " =====
===== "
320 PRINT :PRINT "The numbers in order are:"
330 FOR C=1 TO T-1:LET FLG=0
340 FOR E=1 TO T-C:IF A(E+1)<A(E) THEN LET F=A(E):LET A(E)=A(E+1):LET A(E+1)=F:
LET FLG=1
350 NEXT E
360 IF FLG THEN NEXT C
370 FOR G=1 TO T:PRINT ,A(G):NEXT G:FOR W=1 TO 1000:NEXT W:RETURN
399 REM ** Sorting downwards
400 REM ** I leave this to you...
410 RETURN
499 REM ** Enter new data
500 PRINT CHR$(125):POP :GOTO 70
501 REM ** To be frowned on, but I wanted to meet POP before I died
599 REM ** Re-start
600 RUN
601 REM ** Again not the best way to use a sub-routine
999 REM ** Sub-routine TEST
1000 LET FLAG=0:IF T$="" THEN LET FLAG=1:RETURN
1010 FOR A=1 TO LEN(T$):IF T$(A,A)<"0" OR T$(A,A)>"9" THEN LET FLAG=1
1020 NEXT A:RETURN

```

Apart from the bit I asked you to try to do yourself (starting at line 400, and very much the same as the subroutine starting 300—so use EDIT), this is a full numeric data-handling program. As it stands, it allows you to enter between 5 and 100 whole numbers, and then to search and sort them. It's also an example of a *menu-driven* program. Here's the structure first:

Line 10	initialize
Line 20	set up and enter data
Lines 100–180	menu and choice handling (using a fairly sophisticated routine)
Lines 200–290	search routine
Lines 300–370	upward sort routine
Lines 400	downward sort routine (<i>your</i> bit!)
Line 500	return to start of data entry section
Line 600	return to start of program
Lines 1000–1020	input check routine

Here is a brief explanation of the upward sort routine, if you are interested. But it is complex!

Lines 300–320	initialize
Lines 330–360	the sorting
Line 370	print out sorted list

The sort has two nested loops C and E, causing the micro to pass time and again through the array elements. On each pass, a pair of elements is compared (line 340). If the second is not greater than the first, the elements are swapped (line 340—check it) and the flag is set to show we are not yet done.

COMPLEX ARRAYS

Let's face it, one-dimensional numeric arrays (simple single-column lists of numbers like the above) are not particularly valuable data structures. The strength of arrays becomes really clear when they have more than one dimension, and/or when they store strings instead of, or as well as, numbers.

Program 59 shows how to use a two-dimensional array. Just for a change it is a program specially for secondary school science teachers—but you should not find it hard to modify the concept, and then the program, for your own special needs if they differ!

The program is for an imaginary class of 15 groups, each finding the density of different samples of some given substance. Each group of values for sample mass and sample volume go into the program, and into the array. The micro finds the corresponding density value in each case, and stores these in the array as well. Then all 45 values are printed out in the form of a table, and the class average is shown.

Program 59: A dense array

```

10 DIM A(15,3),CONT$(1):LET T=0
11 REM ** Initialise
19 REM ** Start data entry
20 FOR GP=1 TO 15:PRINT CHR$(125)
30 PRINT "GROUP ";GP
40 POSITION 4,0:PRINT "Sample MASS m/g";:INPUT M:LET A(GP,1)=M
41 REM ** Insert TEST routine as in 58; but anyway you can't have INPUT of an array element direct
50 POSITION 4,14:PRINT "Sample VOLUME V/cm3";:INPUT V:LET A(GP,2)=V
60 LET A(GP,3)=INT(100*A(GP,1)/A(GP,2)+0.5)/100:LET T=T+A(GP,3)
70 FOR W=1 TO 750:NEXT W:NEXT GP
71 REM ** End data entry
79 REM ** Start display routine
80 SETCOLOR 1,0,0:SETCOLOR 2,5,12:SETCOLOR 4,5,10:PRINT CHR$(125):POSITION 4,10:
PRINT "Ready for display";:INPUT CONT$:PRINT CHR$(125)
90 PRINT "Group / m/g / V/cm3 / Density/g/cm3":FOR D=1 TO 37:PRINT "=:NEXT D:PRINT
100 FOR GP=1 TO 15:LET L=PEEK(84)
101 REM ** 84 holds current line on screen
110 POSITION 6-LEN(STR$(GP)),L:PRINT GP;
111 REM ** Note screen format trix!
120 POSITION 13-LEN(STR$(A(GP,1))),L:PRINT A(GP,1);
130 POSITION 21-LEN(STR$(A(GP,2))),L:PRINT A(GP,2);
140 POSITION 34-LEN(STR$(A(GP,3))),L:PRINT A(GP,3)
150 NEXT GP:FOR D=1 TO 37:PRINT "=:NEXT D
160 PRINT :PRINT "Mean density value: ";INT(T/15*100+0.5)/100;"g/cm3"
170 FOR W=0 TO 1 STEP 0:NEXT W

```

Did you note that I called this a two-dimensional array? You may perhaps have thought that, because it consists of four lists, it should really be a four-dimensional array. Not so! The dimensions of an array are just like dimensions in real life. Our first one was a simple list (Program 58), extending on paper in one dimension only. Our second example, Program 59, is two-dimensional because this is a table that extends on paper in two dimensions. Look again at the DIM statements in each case—the first number, if there is more than one, is the number of rows, while the second gives the number of columns.

On paper you can have books of tables. (You may perhaps remember logarithm tables that are still sometimes used at school!) In computing terms, the data stored in such a book would be held in a three-dimensional array, and you would need to set it up using a three-dimensional DIM statement. For instance, if you wanted to hold twenty tables, each of seven rows and three columns, you would use DIM(20,7,3).

In fact, in the case of the Atari micro you can't go so far—two-dimensional numeric arrays are all you can get. (Some people call such an array a matrix.) Some other micros offer more flexibility in their array structures, and indeed have similar multi-dimensional systems for holding strings. Alas, the Ataris do not have string arrays.

You may ask, then, why DIM is used for strings as well as for numeric arrays. Surely that can't be a coincidence? You are right—it is *not* a coincidence. In fact Atari BASIC views a string as a one-dimensional array of characters. Advanced programmers can utilize that feature to mimic the presence of 'proper' string arrays, putting them as substrings within a single string (character array). To do work like this requires a fair knowledge of string functions as well as of what string arrays can offer. I don't think I ought to go that far here.

I'll close with another array program, offered to you without comment, that lets SOUND statements access array element values. As we have four SOUND channels, I'll use a two-dimensional array with four columns.

Here's where I add a little advanced point. When you use DIM A(5), for instance, the micro in fact reserves *six* storage sites, rather than the five you might expect. The reason is our good friend, the sudden zero. The elements of the array are numbered from 0 to 5. Total is six. I'll just as well use that point in this next program, as our SOUND channels have the same numbers.

Program 60: A sound array

```

10 DIM S(40,3)
11 REM ** 40x4 elements, not 40x3!
20 FOR N=1 TO 40
30 LET S(N,0)=N
40 LET S(N,1)=N*2
50 LET S(N,2)=N*4
60 LET S(N,3)=N*6
70 NEXT N
71 REM ** All elements given values now
79 REM ** Now let's use those values....
80 FOR 60=0 TO 1 STEP 0:FOR N=40 TO 1 STEP -1
90 GOSUB 200
100 NEXT N
110 FOR N=1 TO 40
120 GOSUB 200
130 NEXT N:NEXT 60
199 REM ** SOUND sub-routine
200 SOUND 0,S(N,0),10,10
210 SOUND 1,S(N,1),10,10
220 SOUND 2,S(N,2),10,10
230 SOUND 3,S(N,3),10,10
240 RETURN

```

I've really just produced there a tune-playing program that works at random (though of course it's not truly random). But I'd like you to think how easy it would be to use READ...DATA with arrays to put the notes of a tune into a form the micro can quickly play.

Well, there are all sorts of ideas in this chapter for you to explore and develop as you please. And not a single new keyword to learn!

20 Onward

Well, it's got to happen sometime—you're on your own. In this book I have tried to make you happy with basic Atari BASIC. There *are* a few keywords left untouched. They all deal with more advanced coding work, or with such things as disc handling, which most readers won't be into straight away. Appendix 3 gives a full list.

If you've worked through the book with some care up to this point, your grasp of Atari BASIC will be quite enough for you to develop fast on your own. We've had sixty programs (and there are some more in Appendix 2)—check you understand how they work, and then develop them for your own purposes. That'll give you a fair software library!

You should now have no trouble in following programs listed in the better magazines and the better books. (I say 'better', because you must always beware of problems with printing errors. . . .)

But I hope that this book is not due to be thrown away now. I have designed it not just as a teaching guide, but as a source of reference for a long time in your future. So although this is the last chapter, there are still a number of appendices you may find useful in that context. And do use the index, when you want to check on something.

So—onward!

Appendix I: Comments on Questions

Many of the chapters in this book close with ‘Do it yourself’ sections. Well, I *mean* that charming name! You cannot learn how to program a micro by reading a book—you *must* start wearing your finger-tips away on the keyboard.

In this Appendix, then, don’t think you’re going to be able to find answers to the D I Y questions. All I put here are the occasional tips, notes, suggestions and further comments.

There’s another thing. Give a coding task to ten programmers and you’ll get ten programs. Each program will do the job as set out, but there are lots of ways to carry out each module. (If you haven’t got to Chapter 6 yet, a module is a sub-task to be handled within a program.) So I’d be very wrong to give you ‘answers’ and imply that *you* are wrong if your code differs from mine. If your code works, you’ve done the job. Of course I have been coding for a good number of years, so maybe my programs are more efficient, more structured and shorter than yours. Or maybe not—good programmers are born not made and you may very well have more flair than I!

CHAPTER 3

2. I refuse to make up a poem for you. “This pretty young girl used Atari . . .”? Once you’ve made it up, use PRINT and POSITION to make it look good on screen, clearing the display after each verse if it’s an epic.
3. The best symbols are the ones on the upper half of the non-letter keys. But some numbers and letters (small or capital) are useful too. Can I jump ahead and tell you that you can display characters dark on light rather than light on dark?—use the key with the space-age Atari symbol (old micros) or the one showing a sort of semaphore flag (newer ones). Bottom right. And if you press letter keys with CONTROL (CTRL) down you get some special shapes designed just for this purpose.
9. I was a rotten cheat. I coded this on a micro that doesn’t complain about errors (more’s the pity), but printed it out on the same printer. Rotten cheat indeed.
13. Program 37 on Page 128 is one answer.

CHAPTER 4

2. For complex shape printing as is needed here, *do* make copies of the screen grids in this book—Figure 4.4, 4.10 and 4.11 in this case. Sketch what you want with care before going to the keyboard. Use pencil. . . .
5. See Appendix 5.

CHAPTER 10

1. If you can't do this, better go back to the start of the chapter. . . .
2. "Better than"—that's easy.

CHAPTER 11

4. The best test in the book!

CHAPTER 12

3. This book is *not* going to make you an expert arcade game programmer. You'd best make sure you can animate objects this way before dreaming of your own Frog Invaders program. . . .

CHAPTER 13

1. In a loop of course, with lots of nice, neat, useful messages. Perhaps subroutines are the best way to handle the different possible functions.
2. That means add question/answer reward/penalty routines. A nice task to program, and it'll help you with your tables too!
3. What you want is for the program to ignore all even numbers for a start. (No even number, except 2, is prime.) Then, on testing a new number, as soon as it finds a factor, jump out of the loop (in the proper way) and try the next one.
4. The crucial lines will be something like:

```
LET SPEED = 10 * T
LET DIST = 5 * T * T           (faster than 5 * T ^ 2)
```

7. With each input number, enter a subroutine to put it in place and add to the sum (total). A more efficient way is to use an array, but, alas, we don't get to arrays till Chapter 19. Then you'll meet Program 58.
8. Same applies. Program 45 touches on this too (and also uses arrays before time).
9. This is for advanced mathematicians only. If *you* are one such, you'll know how to approximate 'pi' to as much accuracy as you want. But, alas, the Atari can't go far with you.

CHAPTER 14

3. You could try to pull out a substring of limits set by the user, for instance. Or change all upper case characters to lower, and vice versa. Well, you can do quite a lot with strings.
4. Maybe Program 42 will give you an idea for GETting input very poshly. Question 8 hints at this too.

CHAPTER 16

2. The big catch here is that screen lines and columns start with number 0. So you mustn't go above 79 in an 80-column line, for instance.
4. Of course, if you're a student, teacher or market researcher, what an opportunity you have!

6. And how about cursor-control with RETURN to fix the position of, for instance, a chair?

Good Lord—is that all? Clearly I haven't asked you enough questions. Still with only half of that you'll have a fair software library!

Appendix 2: Some More Programs

Just in case you feel you haven't had enough—here are some more! These arose as I was playing around with some specific programming idea and got carried away, or I developed a program round some aim and found it didn't make any special coding point.

Anyway I'll just give you the programs without any comments other than the odd REMs. Enjoy them (the programs, I mean).

Program A2.1: Flitout

```
10 SETCOLOR 1,0,0:SETCOLOR 2,5,14:SETCOLOR 4,5,14
20 LET SPLAT=1000:LET HI=0:POKE 752,1:FOR GAME=0 TO 1 STEP 0:PRINT CHR$(125)
21 REM ** HI is high-score
30 POSITION 0,21:PRINT HI:POSITION 0,2:PRINT "++++++":POSITION
  8,18:PRINT "++++++"
40 FOR L=2 TO 18:POSITION 0,L:PRINT "+":POSITION 31,L:PRINT "+":NEXT L
50 LET X=20:LET Y=10
51 REM ** All above is initialisation
59 REM ** Main loop starts now
60 FOR FLIT=1 TO 5000
70 SOUND 1,50,10,10:FOR DLY=1 TO 50:NEXT DLY:SOUND 1,0,0,0:POSITION X-1,Y-1:PRIN
  T " ":POSITION X-1,Y:PRINT " * ":POSITION X-1,Y+1:PRINT " "
80 POSITION 0,0:PRINT FLIT
90 LET R=RND(0):LET X=X+(-1*(R<0.3333))+(1*(R>0.6667))
91 REM ** Important logic trick there
100 LET R=RND(0):LET Y=Y+(-1*(R<0.3333))+(1*(R>0.6667))
110 IF FLIT>HI THEN LET HI=FLIT
111 REM ** Update high score
120 IF X<0 OR X>31 OR Y<2 OR Y>18 THEN GOSUB SPLAT:LET FLIT=5000
130 NEXT FLIT
140 FOR D=1 TO 250:NEXT D
150 NEXT GAME
999 REM ** Sub-routine SPLAT
1000 SOUND 1,250,0,15
1010 POSITION X-1,Y-1:PRINT "###":POSITION X-1,Y:PRINT "###":POSITION X-1,Y+1:PR
  INT "###":POSITION 16,10:PRINT "SP"
1020 RETURN
```

A pretty weedy game (*you* can't join in, I mean) that's fairly advanced in its animation routine.

Program A2.2: Cat's cradle

```
10 SETCOLOR 1,0,12:SETCOLOR 2,4,2:SETCOLOR 4,4,0:POKE 82,0:OPEN #1,4,0,"K":POKE
  752,1:LET SBSTRNG=500:LET OPEN=600
20 DIM A$(20),B$(20)
30 PRINT CHR$(125):FOR I=0 TO 1 STEP 0:POSITION 0,10:PRINT "Ready when you are.."
  "
40 FOR W=1 TO 400:IF PEEK(764)=255 THEN NEXT W
```

```

41 REM ** Waits a couple of seconds for a key-press
50 IF PEEK(764)=255 THEN POSITION 8,12:PRINT "Oh, DO hurry up!":NEXT I
60 PRINT CHR$(125):POSITION 10,10:PRINT "And about time too."
70 POKE 764,255
71 REM ** Clears input buffer
80 FOR W=1 TO 250:NEXT W
90 LET A$="cat's cradle":GOSUB SBSTRNG:GOSUB OPEN
100 FOR W=1 TO 400:IF PEEK(764)=255 THEN NEXT W
101 REM ** Wait for key-press again
110 GRAPHICS 18:SETCOLOR 4,12,6:POSITION 6,6:PRINT #6;"PHEW!!!":FOR W=1 TO 500:
NEXT W
120 GRAPHICS 0:POSITION 0,10:PRINT "What's your name, by the way?";INPUT A$:POSI
TION 10,14:PRINT "Oh":FOR W=1 TO 500:NEXT W
130 GOSUB SBSTRNG:GOSUB OPEN
140 FOR W=1 TO 400:IF PEEK(764)=255 THEN NEXT W
150 GRAPHICS 0:POSITION 0,10:PRINT "Gimme your date of birth - like 050468":INPU
T A$:POSITION 10,14:PRINT "Ta":FOR W=1 TO 500:NEXT W
160 GOSUB SBSTRNG:GOSUB OPEN
170 GRAPHICS 17:PRINT #6;CHR$(125):LET V=VAL(A$)
171 REM ** YOU should mug-trap that date entry
180 FOR A=1 TO 12:POSITION 0,(A-1)*2:PRINT #6;A;" x ";V;" = ";A*V:NEXT A:FOR W=1
TO 1000:NEXT W
190 RUN
499 REM ** Sub-routine SUBSTRNG
500 GRAPHICS 17:FOR A=0 TO 23:POSITION RND(0)*10,A:PRINT #6;A$(1,RND(0)*LEN(A$(1
,LEN(A$)-1))+1):NEXT A:FOR W=1 TO 1000:NEXT W
510 PRINT #6;CHR$(125)
520 FOR A=0 TO 23:LET X=INT(RND(0)*(LEN(A$)-2))+1:POSITION RND(0)*10,A:PRINT #6;
A$(X,X+RND(0)*(LEN(A$)-X-1)+1):NEXT A:FOR W=1 TO 1000:NEXT W
530 PRINT #6;CHR$(125)
540 FOR A=0 TO 23:LET X=INT(RND(0)*(LEN(A$)-2))+1:POSITION RND(0)*10,A:PRINT #6;
A$(X,LEN(A$)):NEXT A:FOR W=1 TO 1000:NEXT W
550 PRINT #6;CHR$(125):RETURN
599 REM ** Sub-routine OPEN
600 IF INT(LEN(A$)/2)*2=LEN(A$) THEN LET A$(LEN(A$)+1)="!"
601 REM ** Make A$ have odd number of characters
610 FOR A=1 TO 20:POKE 712,RND(0)*255:POKE 708,RND(0)*255:POKE 709,RND(0)*255:SO
UND 0,250-A*10,10,10:PRINT #6;CHR$(125)
611 REM ** Those POKEs deal with colour of #6 background, #6 capitals, and #6 s
mall letters
620 FOR B=1 TO LEN(A$) STEP 2:LET B$=A$(LEN(A$)/2+1-B*0.5,LEN(A$)/2+B*0.5):POSIT
ION 10-LEN(B$)/2,B:PRINT #6;B$:NEXT B
630 FOR W=1 TO 200:NEXT W:SOUND 1,50+A*10,10,10:NEXT A:SOUND 0,0,0,0:SOUND 1,0,0
,0:RETURN

```

Maybe this could have gone near the end of Chapter 14—this is a string-handling effort. “Cat’s cradle” is a string-handling effort, see? No? Oh, well.

Program A2.3: Circle

```

10 GRAPHICS 8:SETCOLOR 1,0,15:SETCOLOR 2,3,0:SETCOLOR 4,3,0:COLOR 1:DEG :LET DRA
W=500:LET TEST=600
11 REM ** Initialise, including setting angle work in degrees
20 LET CENX=159:LET CENY=79:LET RAD=75
21 REM ** Set centre-point and radius
30 GOSUB DRAW
40 FOR CCL=0 TO 1 STEP 0:FOR GO=0 TO 1 STEP 0:PRINT "Centre (x,y)";:INPUT CENX,C
ENY:PRINT "Radius";:INPUT RAD:GOSUB TEST:IF FLG THEN NEXT GO
50 GOSUB DRAW:NEXT CCL
499 REM ** Sub-routine DRAW
500 PRINT :PRINT :PRINT "Centre: ";CENX;",";CENY,"Radius: ";RAD
510 PLOT CENX,CENY:FOR ANG=1 TO 360 STEP 5:LET X=CENX+RAD*SIN(ANG):LET Y=CENY+RA
D*COS(ANG)
511 REM ** Change STEP value to change speed and resolution
520 DRAWTO X,Y
530 NEXT ANG:RETURN
599 REM ** TEST: can circle fit on screen?
600 LET FLG=0:IF CENX-RAD<0 OR CENX+RAD>319 OR CENY-RAD<0 OR CENY+RAD>190 THEN L
ET FLG=1
601 REM ** Grafix allowed within 0<x<319 and 0<y<191
610 RETURN

```

If you had your life again and got a Sinclair Spectrum micro, you'd have CIRCLE as a standard command/statement. As it is, with the Atari we have to work harder to get a round on screen. But you may need the idea in routines of your own.

Program A2.4: Railway Terrace Plate

```

5 REM * This is for my race-loving daughter Rebecca!
10 LET PLACE=500:LET WIN=600:POKE 755,0:SETCOLOR 1,0,0:SETCOLOR 2,12,12:SETCOLOR
  4,12,12:PRINT CHR$(125)
20 FOR LINE=1 TO 21:POSITION 2,LINE:PRINT "!":POSITION 36,LINE:PRINT "!";:NEXT L
  INE
21 REM * Draw 'course'
30 LET H1=3:LET H2=7:LET H3=11:LET H4=15:LET H5=19
31 REM * Horses' tracks
40 LET X1=0:LET X2=0:LET X3=0:LET X4=0:LET X5=0
41 REM * Horses' starting places
50 LET H=H1:GOSUB PLACE
60 LET H=H2:GOSUB PLACE
70 LET H=H3:GOSUB PLACE
80 LET H=H4:GOSUB PLACE
90 LET H=H5:GOSUB PLACE
100 POSITION 1,23:PRINT "Place your bets and then press RETURN.";
110 OPEN #1,4,0,"K":GET #1,1:POSITION 1,23:PRINT "They're OFF!!!";
  ;
120 FOR GO=1 TO 99
130 IF INT(60/2)=60/2 THEN POSITION 1,23:PRINT "They're OFF!!!";
140 IF INT(60/2)<>60/2 AND 60<10 THEN POSITION 1,23:PRINT "They're OFF!!!";
141 REM * Flashy flashing text effect
150 LET H=H1:LET X1=X1+INT(RND(0)+0.5):GOSUB PLACE:IF X1>33 THEN LET WINNER=H:LE
  T 60=100:NEXT GO:GOSUB WIN:RUN
151 REM * Luvverly chance for editing there!
160 LET H=H2:LET X2=X2+INT(RND(0)+0.5):GOSUB PLACE:IF X2>33 THEN LET WINNER=H:LE
  T 60=100:NEXT GO:GOSUB WIN:RUN
170 LET H=H3:LET X3=X3+INT(RND(0)+0.5):GOSUB PLACE:IF X3>33 THEN LET WINNER=H:LE
  T 60=100:NEXT GO:GOSUB WIN:RUN
180 LET H=H4:LET X4=X4+INT(RND(0)+0.5):GOSUB PLACE:IF X4>33 THEN LET WINNER=H:LE
  T 60=100:NEXT GO:GOSUB WIN:RUN
190 LET H=H5:LET X5=X5+INT(RND(0)+0.5):GOSUB PLACE:IF X5>33 THEN LET WINNER=H:LE
  T 60=100:NEXT GO:GOSUB WIN:RUN
200 NEXT GO
499 REM * Sub-routine PLACE
500 SOUND 1,250-H*10,10,8
510 POSITION X1*(H=H1)+X2*(H=H2)+X3*(H=H3)+X4*(H=H4)+X5*(H=H5),H:PRINT " #0":FOR
  W=1 TO 10:NEXT W
520 SOUND 1,0,0,0:RETURN
599 REM * Sub-routine WIN
600 FOR A=1 TO 100:SOUND 1,250-RND(0)*2*A,10,15:LET X=RND(0)*15:SETCOLOR 1,0,X:S
  ETCOLOR 2,12,15-X:NEXT A
601 REM * Typical Deeson flashy finish
610 POSITION 1,23:PRINT "The winner is horse number ";(H+1)/4;"!";
620 FOR A=1 TO 100:SOUND 1,250-RND(0)*2*A,10,15:LET X=RND(0)*15:SETCOLOR 1,0,X:S
  ETCOLOR 2,12,15-X:NEXT A
630 FOR W=1 TO 500:NEXT W:RETURN

```

Horse-racing with the Atari. You can bet on which horse wins. And one day you may be able to define characters to make better horses.

Program A2.5: Ticker

```

10 DIM MESSAGE$(120),BANNER$(150),BORDER$(20):LET BORDER$="#####"
:POKE 82,0
20 PRINT CHR$(125);CHR$(253);"Please type in your message and RETURN."
30 POSITION 0,10:INPUT MESSAGE$
40 LET BANNER$="          ** "
50 LET BANNER$(LEN(BANNER$)+1)=MESSAGE$:LET BANNER$(LEN(BANNER$)+1)=" **
  ;
51 REM * Concatenation: makes the banner string = intro symbols+message+closing
  symbols
60 GRAPHICS 18:SETCOLOR 0,8,0:SETCOLOR 1,0,14:SETCOLOR 2,0,0:SETCOLOR 3,2,10:SET
  COLOR 4,2,6

```

```

61 REM * 'Best' colour settings for all #6 print styles
70 POSITION 0,2:PRINT #6;BORDER$:POSITION 0,8:PRINT #6;BORDER$
80 FOR GO=0 TO 1 STEP 0
90 FOR CHAR=1 TO LEN(BANNER$)-20:IF PEEK(764)<>255 THEN POKE 764,255:RUN
91 REM * Stop display with any key-press
100 POSITION 0,5:PRINT #6;BANNER$(CHAR,CHAR+19)::SOUND 1,200,10,14:FOR W=1 TO 10
0:NEXT W:SOUND 1,0,0,0
110 NEXT CHAR
120 NEXT GO

```

As in ticker-tape rather than the old heart. Quite nice for a display when you're rich enough to have a shop-window.

Program A2.6: Timer

```

10 LET RNDEL=500:LET TMEAS=600:LET SHOW=700:POKE 82,0:PRINT CHR$(125):SETCOLOR 1
,0,14
20 PRINT:PRINT "R E F L E X   T I M E R":PRINT "=====
30 PRINT:PRINT:PRINT "Please press the number of your choice!"
31 REM * Menu
40 PRINT:PRINT:PRINT "1. Simple - screen only":PRINT:PRINT "2. Harder - sou
nd":PRINT:PRINT "3. Harder still - both:PR:PR:PR."
41 REM * POSITION is not always simplest in practice
50 OPEN #1,4,0,"K"
60 FOR C=0 TO 1 STEP 0:GET #1,I:LET I=I-48:IF I>0 AND I<4 THEN LET C=2
70 NEXT C
71 REM * Mug-trap
80 LET BESTT=1000:LET SUM=0:GRAPHICS 18:GOSUB 100*1:FOR W=1 TO 1000:NEXT W
81 REM * NB Computed sub-routine address
82 REM * NB too the simple maths tricks with BESTT and MEANT
90 RUN
91 REM * Re-start
99 REM * TEST sub-routines
100 FOR GO=1 TO 10:PRINT #6;CHR$(125):PRINT #6:PRINT #6;"T E S T   1":PRINT #6:P
RINT #6;"GO ";GO:";"
110 POSITION 2,8:PRINT #6;"PRESS return WHEN THE SCREEN CLEARS!"
120 GOSUB RNDEL:PRINT #6;CHR$(125):LET T0=INT((65536*PEEK(18))+(256*PEEK(19))+PE
EK(20)):POKE 764,255:GOSUB TMEAS
121 REM * POKE is to stop cheating
130 NEXT GO:RETURN
200 FOR GO=1 TO 10:PRINT #6;CHR$(125):PRINT #6:PRINT #6;"T E S T   2":PRINT #6:
PRINT #6;"GO ";GO:";"
210 POSITION 2,8:PRINT #6;"PRESS return WHEN THE BUZZER CLICKS!"
220 GOSUB RNDEL:SOUND 1,250,0,10:LET T0=INT((65536*PEEK(18))+(256*PEEK(19))+PEEK
(20)):POKE 764,255:SOUND 1,0,0,0:GOSUB TMEAS
221 REM * ?CHR$(125) won't work in this Mode; that buzz is too long anyway
230 NEXT GO:RETURN
300 FOR GO=1 TO 10:PRINT #6;CHR$(125):PRINT #6:PRINT #6;"T E S T   3":PRINT #6:P
RINT #6;"GO ";GO:";"
310 POSITION 2,8:PRINT #6;"PRESS return WHEN THE BUZZER CLICKS AFTER THE SCR
EEN CLEARS!"
320 GOSUB RNDEL:LET R=1:PRINT #6;CHR$(125):GOSUB RNDEL:LET R=0:GOSUB RNDEL:SOUND
1,250,0,10
330 LET T0=INT((65536*PEEK(18))+(256*PEEK(19))+PEEK(20)):POKE 764,255:SOUND 1,0,
0,0:GOSUB TMEAS
340 NEXT GO:RETURN
499 REM * Other sub-routines
500 FOR W=1 TO (10*(I=1)+10*R)+(RND(0)*75)*(1+(I=1))*(1+(10*( NOT R)))
501 REM * NB logical expression; can you follow it and then see why I use it?
510 NEXT W:RETURN
600 FOR W=0 TO 1 STEP 0:IF PEEK(764)<>255 THEN LET W=2
601 REM * Wait for key-press technique
610 NEXT W:LET T1=INT((65536*PEEK(18))+(256*PEEK(19))+PEEK(20)):LET T=(T1-T0)/50
611 REM * In North America use 60 not 50
620 LET SUM=SUM+T:LET MEANT=SUM/GO:IF T<BESTT THEN LET BESTT=T
621 REM * Check arithmetic techniques!
630 GOSUB SHOW:RETURN
700 PRINT #6;CHR$(125);"R E S U L T S":PRINT #6;"-----"
710 POSITION 4,3:PRINT #6;"time ";T:"sec"
720 POSITION 4,5:PRINT #6;"best ";BESTT:"sec"
730 POSITION 4,7:PRINT #6;"mean ";INT(100*MEANT+0.5)/100;"sec"

```

```

740 FOR W=1 TO 1000:NEXT W:POSITION 0,9:PRINT #6;"RETURN to go on..."
750 POKE 764,255:GET #1,J:RETURN
751 REM * Wait for key-press to go on

```

Has computing improved your reflexes? Likely not, though Missile Command (the Atari game I most love to hate) may help. TIME is of course the essence.

Program A2.7: Wordguess

```

10 LET ODPS=500:DIM TARGET$(10),6$(12):GRAPHICS 2:SETCOLOR 4,5,6:POKE 708,14:POK
E 709,248,
11 REM * POKes handle #6 colours
20 DIM F$(20):LET F$="" ":FOR WORD=1 TO 5:READ TARGET$:LET T=L
EN(TARGET$)
21 REM * Limit depends on word-store
30 FOR GO=1 TO 99:PRINT #6;CHR$(125)
40 POSITION 3,1:PRINT #6;"guess the word"
50 LET P=9-T/2:FOR X=P TO P+T-1:POSITION X,5:PRINT #6;"-":NEXT X
60 LET CHECK=0
70 FOR G=0 TO 1 STEP 0:POSITION 0,7:PRINT #6;"type guess + RETURN":POSITION P,6:
INPUT G$:IF LEN(G$)=T THEN LET G=2
80 IF LEN(G$)<>T THEN GOSUB ODPS
90 NEXT G
100 FOR CHAR=1 TO T:IF G$(CHAR,CHAR)<TARGET$(CHAR,CHAR) THEN POSITION P+CHAR-1,5
:PRINT #6;"<"
101 REM * Now check each character in turn
110 IF G$(CHAR,CHAR)>TARGET$(CHAR,CHAR) THEN POSITION P+CHAR-1,5:PRINT #6;">"
120 IF G$(CHAR,CHAR)=TARGET$(CHAR,CHAR) THEN POSITION P+CHAR-1,5:PRINT #6;G$(CHA
R,CHAR):LET CHECK=CHECK+1
150 NEXT CHAR:FOR W=1 TO 1000:NEXT W:IF CHECK=T THEN LET SC=60:LET GO=100
160 NEXT GO:POSITION 0,7:PRINT #6;F$:POSITION 3,7:PRINT #6;"got it in ";SC;"!":F
OR W=1 TO 1000:NEXT W:NEXT WORD:RUN
161 REM * RUN includes RESTORE
200 DATA COMPUTER,ATARI,WORD,BUZZ,GUESS
499 REM * ODPS
500 PRINT #6;"wrong length!":FOR W=1 TO 500:NEXT W:POSITION 0,6:PRINT #6;F$:RETU
RN

```

Quite a nice word-game, using complex screen-handling. Each dash on screen is a letter in the target word. Type your guess at the word (with RETURN). For each letter, the display shows '<' if your letter is earlier in the alphabet than the correct one, or '>' if it is later. When you get the word right, along comes a new word to guess.

Program A2.8: Wallpaper

```

10 GRAPHICS 11:DEG
11 REM * Multicolor mode: DEG cos I find RAD tuff
20 FOR GO=0 TO 1 STEP 0:SETCOLOR 4,0,8:FOR LOOP=0 TO 3
21 REM * SE.4... handles brightness in this mode
30 LET CENX=20+RND(0)*40:LET CENY=20+RND(0)*150:LET ST=RND(0)*360
31 REM * Set centre and starting angle
40 FOR SPK=1 TO 360 STEP 5:LET COL=RND(0)*15:PLOT CENX,CENY:COLOR COL:SOUND LOOP
,LOOP+250-SPK/2,10,SPK/25
41 REM * Check RND in the colour and sound bits
50 DRAWTO CENX+SPK/20*SIN(SPK*ST),CENY+SPK/20*COS(SPK*ST)
51 REM * Uniformly rising angle and radius
60 NEXT SPK:NEXT LOOP:FOR LUM=0 TO 15:SOUND INT(LUM/4),0,0,0:SETCOLOR 4,0,LUM:FO
R W=1 TO 50:NEXT W:NEXT LUM:NEXT GO
61 REM * Can you follow the SOUND control there?

```

We all need a relax sometimes. If you still need your micro to be working when you take a break, try this so-called restful routine. IF you have an XL Atari . . .

Appendix 3: BASIC Survey

I give here very brief notes on all Atari BASIC keywords. They're for you to refer to rather than to learn from. After all, I want you to keep this book for a long time as a useful work of reference!

I use these symbols to make the notes even more brief:

N	numeric constant (e.g., 4), variable (e.g., TOTAL) or expression (e.g., L/2+1)
N\$	string constant ("Shiva"), variable (T\$), expression (T\$(4,7))
NC	numeric constant
NC\$	string constant
NV	numeric variable
NV\$	string variable

I'll also tell you about short forms if it's worth it.

ABS(N) Function: returns the absolute value of N, ignoring sign. LET B = ABS(A).

ADR(NV\$) Function: returns the decimal address of the storage site at which the start of the argument is or will be kept. ?ADR(ES\$)

AND Logical operator: joins two conditions, and is TRUE (value 1) only if both are true. IF A AND B THEN... (= IF A <> 0 AND B <> 0 THEN...)

ASC(N\$) Function: gives the Atari code (similar to ASCII) of the first character of the string argument, or 'code' 44 if the string is empty (LEN = 0). ?ASC(A\$)

ATN(N) Function: returns the angle whose tangent is the argument, in radians unless you've set DEG mode. ?ATN(-100)

BREAK Interrupt key: stops the micro in its tracks whatever it is doing (unless it's LPRINTing), and restores command mode.

BYE (B.) Command/statement: causes the micro to leave BASIC, clearing all storage sites (in new machines only), and enter the operating system (old versions) or the memory test routine (new machines).

CHR\$(N) Function: returns the character or control action with code N. Values of N outside the range 0-255 repeat those in it as long as you stay within the range 0-65535. Early Ataris do not allow a CHR\$ term on both sides of an inequality. ?CHR\$(130)

CLOAD Command/statement: causes a NEW, then attempts to load from cassette the next program found saved with CSAVE or SAVE"C: ...".

CLOG(N) Function: returns the common (base 10) logarithm of the argument; this must be positive. ?CLOG(DANCE)

CLOSE(CL.) # N Command/statement: closes an input/output channel from further use. If a channel is open to one device, you must close it before you open it to a second. N values from 1–7 only are allowed. CL. # X + 3

CLR Command/statement: clears the contents of all storage sites holding numeric and string variables; un-dimensions all arrays and strings; does a RESTORE.

COLOR(C.)N Command/statement: controls the format of PLOT and DRAWTO work. In Mode 0 (GR.0) selects the character to plot/draw with. In Modes 1 and 2 the same, but colour too. In Modes 3–7 the colour register to be used. In Mode 8 the brightness. In other modes similarly. N may not be negative. C.3

COM NV(\$) (...) Command/statement: just the same as DIM. Why both? I dunno. COM A(6,3)

CONT Command (ignored if a statement, and pointless then anyway): instructs the micro to go on from next program line after an error crash or after BREAK, RESET, STOP or END. It may fail in the second case.

COS(N) Function: returns the cosine of the argument, assumed to be an angle in radians. ?COS(-1)

CSAVE(CS.) Command/statement: attempts to control the cassette machine, and to copy the program in store onto tape in a form that only CLOAD can re-read.

DATA(D.)NC(\$)... Statement: signals the start of a series of data items, separated by commas. Data strings cannot therefore contain commas, though no other character is blocked. DATA 4,Y, Listen!

DEG Command/statement: instructs the micro to work with degrees rather than radians for angle unit.

DIM NV(\$) (...) Command/statement: reserves storage space for numeric arrays and for strings (character arrays). Once you've dimensioned you cannot re-dimension except after CLR. DIM PL\$(15), ARRAY(14,10)

DOS Command: turns the micro over to the disc operating system. If you haven't a disc, DOS BYEs you instead.

DRAWTO (DR.)N1,N2 Command/statement: after a GR. instruction, draws a line, as straight as can be, from the last point used by PLOT or DRAWTO (but not by POS.). The colour is that of the background unless you first use a COLOR instruction. In Modes 0–2, the line drawn consists of characters (set by COLOR) rather than little squares. DR.15,3

END Command/statement: makes a running program halt as if it had reached its normal end. Programs need not end with END, nor need END be the last statement. END.

ENTER(E)"..." Command/statement: causes loading of a program saved with LIST "...", from cassette ("C:...") or disc ("D:..."). The new program will 'merge' with one in memory rather than NEWing that first, to produce a combination of the two. If both programs have a line number the same, however, the new version will replace the old one. ENTER "C:SUB27"

EXP(N) Function: returns the number 'e' (2.718. . .) raised to the power of the argument. ?EXP(0) (though *that* may give a slightly wrong result!)

FOR(F.)NV... Command/statement: signals the start of a loop. FOR GO = 1 TO 50

FRE(N) Function: returns the amount of unused storage in bytes. N can take any value: 0 is best. ?FRE(N)

GET # N,NV Command/statement: read a byte from an open channel N and store its value in NV. If the channel is the keyboard, NV stores the code of the character got. GET # 3, CHAR

GOSUB(GOS.)N Command/statement: sends control to a closed subroutine starting at line N (ERROR if doesn't exist), storing in a stack the current position to allow later return. GOSUB TEST

GO TO(GO.)N or **GOTO N** Command/statement: sends control to line N (if it exists) without the possibility of return. Well-structured programs do not need GO TOs, even if only implied. GO TO MAINPROG

GRAPHICS (GR.)N Command/statement: puts the display into Mode N, perhaps clearing the screen. In this book I've looked at the full official range (0-56), but other positive values of N produce effects worth an explore. GR.4 + 16

HELP Key: when pressed, this puts a special value into storage that can be read and acted upon. (New Ataris only.) See Page 197.

IF... Command/statement: expresses the condition(s) in which the statement that follows is to be carried out. The condition or set of conditions must take the logical value 1 (= TRUE) for the rest of the line to be used. The logical value of a FALSE statement is 0. IF SCORE AND ANSWER\$ = "Y" THEN...

INPUT (I.) NV(\$) Command/statement: waits for keyboard entry with RETURN and assigns the data entered to NV(\$). (May also be used with other open channels in the form I. # N,NV(\$).) INPUT A,B\$

INT(N) Function: returns the argument rounded off to the whole number left of it on the number line, unless it is already a whole number. ?INT(5.6)

LEN(N\$) Function: counts the number of characters in N\$ *as assigned*, including any control characters. ?LEN(LION\$)

LIST (L.) Command/statement: lists the program, or the lines specified, to the screen (editor) or to the channel specified. Thus LIST "C:..." sends the program to the cassette recorder, saving it in the form loadable by ENTER. And LIST "P:" sends a listing to the printer. LIST 200,2050.

LOAD (LO.) "..." Command/statement: attempts to read into store a program on cassette ("C:...") or disc ("D:...") saved with SAVE "...". LOAD "C:PROG 6"

LOCATE (LOC.),N1, N2, NV Command/statement: returns to NV the code for the character printed or point plotted at screen site N1,N2. The code is that used by COLOR. This is a graphics instruction, so you must use GR. before. LOC.5,0, CODE

LOG(N) Function: Returns the natural logarithm (base "e") of the argument. N must be positive. ?LOG(JAM)

LPRINT(LP.) Command/statement: follows the rules of PRINT, but sends the items to the printer rather than the screen. LPRINT "End of test "; TEST; ".,"

NEW Command/statement: clears the current program and most variables from access in memory.

NEXT (N.)NV Command/statement: marks the end of the FOR... loop using NV as loop counter. NEXT BOUNCE

NOT Unary logical operator: returns 1 if the expression that follows is FALSE, and 0 if it is TRUE—inverts its logical value in other words. ?NOT FLAG

NOTE (NO.) # N,NV1,NV2 Command/statement: with newer disc operating systems, this reads the current disc file pointer and sets its sector and byte values to NV1 and NV2. N is the channel open to the pointer. NOTE # 3,5,B

ON NV... Command/statement: sends control to one of several subroutines or program lines depending on the value of NV. ON CHOICE GOS. TEST, REPEAT, GAME

OPEN (O.) # N1,N2,N3, "..." Command/statement: opens channel 1 to the device in speech marks (such as "K:" or "P:"); N2 and N3 select the details. OPEN # 6,4,128,"C:"

OPTION Key: when pressed, this puts a value into store that can be checked and acted upon.

OR Logical operator: gives TRUE if either or both the expressions next to it are TRUE, and FALSE if both are FALSE. IF SCORE = 10 OR BONUS > 5 THEN...

PADDLE(N) Function: returns a value that depends on the current state of the paddle control specified by N. ?PADDLE(1)

PEEK(N) Function: returns the value in storage site N, in decimal form. ?PEEK(87)

PLOT(PL.)N1,N2 Command/statement: puts a character (specified by COLOR) in Modes 0–2, or a 'dot' in Modes 3 upwards, on screen at site N1 across and N2 down. PLOT 10,14

POINT (P.) # N1,N2,N3 Command/statement: with newer disc operating systems, resets the pointer for the opened channel to sector N2, byte N3. POINT # 3,17,110.

POKE N1,N2 Command/statement: copies N2 into storage site N1, if that site exists in RAM. POKE 83,37

POP Command/statement: clears the latest entry to the stack, so turns a GOSUB into a GO TO in effect, or cancels the current FOR...NEXT loop. Best avoided!

POSITION (POS.)N1,N2 Command/statement: prepares to move the cursor to the screen site specified ready for GET, INPUT, LOCATE, PRINT or PUT. POSITION X,Y + 3

PRINT (?,PR.)... Command/statement: displays the print items that follow on Mode 0 screen or through the channel specified with #, that channel being open. PRINT # 6; "Something"

PTRIG(N) Function: returns 0 if you press the trigger of paddle N, 1 otherwise. ?PTRIG(0)

PUT # N1,N2 Command/statement: sends a byte, value N2, through channel N1 if open. PUT # 7,48

RAD Command/statement: instructs the micro to work with the radian as the unit of angle rather than degree.

READ NV(\$) Command/statement: assigns the next free DATA item to NV(\$).
READ PI

REM (. or R.) Command/statement: tells the micro to ignore the rest of the line, it being a message to the programmer. **REM **** Start subroutines

RESET Key: stops the micro in its tracks, with some loss of data and a return to Mode 0.

RESTORE (RES.) Command/statement: returns the DATA pointer to the start of the DATA list(s), from line N if specified. **RESTORE BLOCK2**

RETURN Key: shows the end of the current data entry.

RETURN (RET.) Command/statement: causes control to go back to the statement after the last GOSUB met. **RETURN**

RND(N) Function: returns a pseudo-random number from 0.000... to 0.999... N can take any value; 0 is best. ?RND(0)*10

RUN Command/statement: causes the current stored program, or that read from a device, such as cassette "C:", to start execution. **RUN "C:"** can handle only programs saved with **SAVE "C:..."**. **RUN "C: CHECKBOOK"**

SAVE (S.) "..." Command/statement: sends to cassette ("C:") or to disc ("D:") the program in store, with an optional name in the former case. **SAVE "C:Masterpiece"**

SELECT Key: when pressed, puts a value into a storage site that can be used.

SETCOLOR (SE.) N1,N2,N3 Command/statement: puts into register N1 (0-4), numbers for colour (N2) and brightness (N3). If the command is valid, the micro carries it out at once over the whole screen. **SETCOLOR 4,6,8**

SGN(N) Function: returns -1 if N is negative, 0 if 0, and 1 if it's positive. ?SGN(BORD)

SIN(N) Function: returns the sine of angle N, assumed to be radians unless you've put in DEG. ?SIN(TAX)

SOUND (SO.) N1,N2,N3,N4 Command/statement: turns channel N1 on or off, with pitch related to N2, volume to N4, and 'distortion' to N3. **SOUND CHA,P,T,A,**

SQR(N) Function: returns the square root of N if not negative. ?SQR(UL)

STATUS (ST.) # N,NV Command/statement: puts into NV a report on what last happened in channel N. **STATUS # 7, ERROR**

STEP N Command/statement: part of FOR... giving the increment by which the loop counter should change each time round. **FOR W = 0 TO 1 STEP 0**

STICK(N) Function: gets output between 0 and 15 depending on what's going on with joystick N. ?STICK(INSECT)

STOP Statement: halts the running of a program under its own control, with a stopped message.

STRIG(N) Function: returns 0 if the trigger of joystick N is pressed, otherwise 1.
?STRIG(IL)

STR\$(N) Function: returns a string whose characters are the digits of N; older Ataris can't have *STR\$* on both sides of a comparison. You do not need to DIM *STR\$* strings!
LET DATE\$ = STR\$(DATE)

SYSTEM RESET See *RESET*

THEN Command/statement: result part of *IF...*

IF NOW = BEDTIME THEN PRINT "Night-night!"

TO Command/statement: closing part of *FOR...*

FOR WINK = 1 TO 40: PRINT "Snore": NEXT WINK

TRAP(T.)N Command/statement: sends control to line N if an *ERROR* appears.
TRAP ERRORSUB

USR(N) Command/statement: calls a machine code subroutine starting from memory site N. *?USR(500000)*

VAL(N\$) Function: returns a number whose digits are the (numeric) characters of *N\$*. If *N\$(1,1)* is not a number, an *ERROR* appears. Otherwise *VAL* copes somehow.

LET V = VAL("2-stroke")

XIO(X.)N1, #N2,N3,N4, "..." Command/statement: general input/output command, action being *N1*, channel being *N2* (defined in "..."), and details from *N3* and *N4*.
XIO 18, # 6,1,1,"S:"

Appendix 4: PEEK-a-boo

BASIC is what we call a fairly high-level programming language. Programs written using it are changed by the micro into machine code, which consists of no more than strings of 0s and 1s.

As compared with machine code, a high-level language is (supposed to be) one which

- consists of instructions and symbols which closely relate to human usage;
- requires no knowledge of how the micro really works;
- requires no knowledge of how and where data is held in store;
- offers user-designed labels ('names') for statements, data items and constants;
- lets one instruction represent many at machine code level.

A BASIC program line like "~~100~~ PRINT SQR(CAT*DOG)" is fairly high-level—it's fairly easy for humans to follow, and converts to dozens of machine code instructions.

A really high-level programming language would allow instructions like "Put these numbers in order, find their mean, and plot a graph" or "Correct all the spelling mistakes in this chapter". Alas, we're still far from having software able to handle that sort of thing.

Meanwhile we have BASIC, or rather the dozens (hundreds?) of forms of it. Dialects, we call them of course. The forms range from the original, given to the world in 1964, long before micros had appeared, to such creatures as Sinclair SuperBASIC and COMAL, which really differ so much that they are new species. Modern BASICs handle complex features like colour and sound and high-resolution screen graphics that BASIC's founders never heard of.

Atari BASIC does all that too of course, but it *does* date back a few years, so is not always as flexible as other dialects. To use Atari BASIC with flair often means that you need to dip down into machine level work sometimes. PEEK, POKE and USR are the BASIC keywords for that purpose. I've used PEEK and POKE quite a lot in this book. USR lets you run a machine code subroutine (yours or one of Atari's in-built ones) if it can do things better than a BASIC one. But I'm not going to delve into USR here.

In this appendix I want to list the more useful PEEK/POKE techniques. First a reminder of what the keywords do. 'Peek' is old English/modern North American for 'peep', meaning 'take a little look'. PEEK(N) gives you a little look into storage site N in the micro, so you can find out the value of the byte (decimal 0–255) held there at that moment. And 'poke'—I guess you know what that means—lets you change the contents: with POKE N, BYTE you replace the current value stored in N with your own BYTE. Figure A4.1 outlines the uses for the various blocks of the 64K storage units possible. You can PEEK anywhere you like. If the site at address N isn't used, you'll get a value of 0 back. You can POKE anywhere you like, but it won't get you anywhere if the site poked is unused or in ROM. (ROM = Read Only Memory. Yes?)

Sometimes of course, there's a need for a value to be greater than a byte can hold. We often use numbers greater than 255—so does the micro. Some storage sites are double size (or even larger), with two (or more) addresses. Thus sites 88 and 89 hold the start

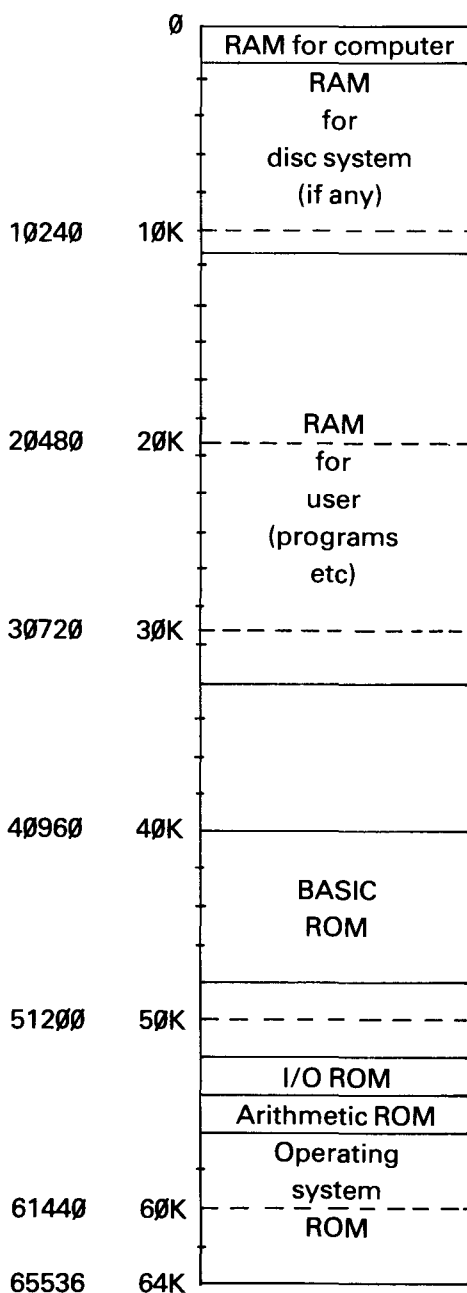


Figure A4 Outline Atari memory map.

address of where the screen contents appear in memory. You'd have to PEEK out that info with `PRINT PEEK(88) + PEEK(89)*256`. In general if N1 (lower) and N2 (higher) are the two addresses concerned, PEEK with

`PRINT PEEK(N1) + PEEK(N2)*256`

and POKE in a value V with

`POKE N1,V-INT(V/256)*256: POKE N2, INT(V/256)*256`

Here's my table of the main storage site addresses worth noting. The note (2) after an address means double byte—the address quoted is N1 as above, with N2 straight after.

- 14(2) Holds the address just below the start of screen display—the highest one usable for programs and program data.
- 17 128, unless BREAK has just been used, when the value becomes 0.
- 18(3) Time since switch-on, in fiftieths (US—sixtieths) of a second.
- 65 POKE 65, 0 turns off speaker sound during loading/saving.
- 66 POKE 66, 1 stops the micro keeping full keyboard watch, so gives an increase in processing speed. Only a small increase, though. POKE 66,0 to return to normal.
- 77 Deals with 'attract', the screen flash effect that starts after a few minutes to stop the screen 'burning'. Attractive? POKE 77,255 to save you waiting those few minutes. Pressing any main key puts the figure back to 0. (It then steps up each five seconds or so back towards 255.)
- 82 Carries the left margin offset. This is 2 unless you POKE some other value in. POKE 82,39 has a nice effect. More than 39 locks out the keyboard rather than the screen. . . .
- 83 The same for the right margin.
- 84 Carries the screen line on which the next action will take place.
- 85(2) The same for screen columns.
- 87 Lets you PEEK the current mode number. A POKE can be upsetting, though.
- 88(2) The address of the first byte of screen memory, the top left-hand corner.
- 90 As 84, but for DRAWTO and XIO 18 statements.
- 91(2) As 85 for DR. and XIO 18.
- 93 The code for the character at the cursor position. Not the standard Atari code, though.
- 94(2) The address held here holds the cursor position.
- 106 'RAMTOP'—the address of the end ('top') of RAM. ?PEEK(106)/4 gives the number of K bytes of RAM you have in your Atari. You can POKE in a lower value to fool the micro into thinking there's less RAM—thus giving you total control over a block of storage sites for your own purposes. You must be able to divide the value you poke in by 4.
- 136(2) Holds the address of the start of your program.
- 140(2) Holds the address of the end of the program.
- 142(2) Holds the address of the end of the program's strings and arrays.
- 144(2) Holds the address of the end of the storage space used by the program as a whole.
- 186(2) The line number where your program was when you stopped with STOP or BREAK, or where *it* stopped with ERROR.
- 195 Briefly carries the number of the last ERROR.
- 201 The number of columns per TAB (the comma kind). Standard value is 10.
- 251 Holds 0 if you're in RAD mode, and 6 for DEG.
- 559 Controls 'direct memory access' for use with sprites. If the value here is 0 you can't use sprites. Other values effect the sprite style. POKE 559,0 in cutting down screen output also lets the micro work rather faster; use the instruction then when your program comes to a major chunk of complex processing.
- 564(2) Hold the position of the light pen, if used.
- 580 POKE 580,1 causes (SYSTEM)RESET to have the effect of switching off and on again.
- 621 (XLs only) POKE 621,1 to lock out the keyboard; return to life with 621,0. The former is a good trick to play on a mate—but for serious uses the instructions need to come in a program. Otherwise RESET is the answer.
- 622 (XLs only) POKE 622,1 gives 'fine scrolling'—a slower and thus less jerky scroll than you get normally, or after POKE 622,0.
- 623 Values here fix the priority of sprites—which goes in which screen 'layer'.
- 656 As 84 but for text window.
- 657 As 85 similarly.
- 659 POKEs give some interesting text window effects.

660(2) The address of the start of the text window.
665 As 93 for the text window.
675(15) A 120-bit long map of the stops for TAB.
694 If 0—normal characters; other values give interesting codes. POKE 694,32 switches on lower case mode; use 64 for a CONTROL switch, and 128 for an inverse display switch.
702 Values concern keyboard locks. 0 is normal, and 128 is CONTROL lock for instance.
703 POKE this with 4 to prevent printing on screen. Useful for programs using printer as well as video, or as part of TRAP routine.
704(1) Various colour registers, 708–712 being the SETCOLOR ones.
729 (XLS only) POKE 729,X to give a delay of X fiftieths of a second (sixtieths in North America) before the key repeat action starts. The norm is 40, so POKE 729,10 gives you a good value for speedy typing. POKE 729,0 cuts out the effect entirely. This is worth doing for preventing repeating problems with youngsters. . . .
730 (XLS only) As for 729 but looking after the key repeat rate. The normal value is 5.
731 (XLS only) You can turn off the key click with POKE 731,1. I prefer it on myself—get it back with POKE 731,0.
732 (XLS only) This storage site contains a value that relates to the use of the HELP key. Read it with PEEK (732), then clear it with POKE 732,0. The values are 17 if HELP has been pressed, 81 if it's SHIFT and HELP, and 145 for CONTROL and HELP. That means you can offer three levels of help by IF PEEK(732) = 17 THEN GOSUB HELP1 and so on.
740 True RAMTOP (you can't POKE this to fool the micro, unlike 106).
741(2) 'MEMTOP'—hold the address of the highest byte of free storage.
743(2) Do the same for the bottom.
752 POKE 1 here to make the cursor invisible. Only takes effect when you next use PRINT, though.
755 Normal value is 2—giving a visible but see-through cursor. 0 hides the cursor, 1 also makes non-see-through, 3 the same except the cursor is invisible. Add 4 to each of these for the same effects and upside-down characters! Neat!
756 Its values select character set. 204 in Mode 1 gives the international set for instance, the one with £. Some nice effects follow POKES here. . . . POKE 756,224 to get back to real life.
763 Contains the Atari ASCII code for the last character entered or copied, or the value of the last point plotted.
764 Contains the code of the last key-press, or 255 if none. POKE 764,255 will delete the last key-press. Sometimes useful before INPUT. PEEK(764) acts rather like GET (but can be better).
765 To be poked with the colour value you want used with XIO 18... (FILL).
767 If the value here is 0, display can scroll. If 255, not. Gives you scroll control in program, rather than with CONTROL and 1.

Appendix 5: The Atari Character Set

























Well, maybe I should say sets? It takes a lot of effort to get into using any but the Mode 0 characters (and even they aren't all simple!). By the time you explore Modes 1 and 2 and the international set (newer Ataris only)—phew!


































The table that follows gives the 256 CHR\$(X) results in Mode 0, Modes 1 and 2 (standard and alternative versions), and when you try going international—which UK readers will have to do to get the pound (£) into their programs. The work will POKE you to death, even if the results are COLORful. Here are some notes to help you.


Notes

























































- 1. A dagger (†) shows a function/character duality—?CHR\$(X) will cause the function to occur; ?“(ESCAPE)X-key” will do the same in a program; and ?CHR\$(27); CHR\$(X) will give you the symbol described.
- 2. Standard Mode 1/2 characters appear with ? #6;CHR\$(X). Alternatively enter the COLOR register you want—the number in brackets in column 3, and ? # 6;... the corresponding character from column 2.
- 3. Alternative Mode 1/2 characters come when you do all that with a POKE 756, 226 first (and POKE 756,224 after). The COLOR codes are as in column 3 still.
- 4. The ‘international’ character set, available only on new Ataris, comes with POKE 756,204 (Mode 0) and POKE 756,206 (Modes 1 and 2). Column 5 shows what you get in Mode 0. In Modes 1 and 2 you get the main 64 characters in each of the four colours. Again then the colour register numbers in column 3 apply.






Table A5.1: Effects of CHR\$ codes




Code number	Mode 0	Mode 1/2 standard		Mode 1/2 alternative	‘International’
0	heart	space	(1)	heart	á
1		!	(1)		ù
2		”	(1)		Ñ
3			(1)		É
4		\$	(1)		S
5		%	(1)		Ô
6	bold slash	&	(1)	bold slash	Ò
7	bold back-slash	,	(1)	bold back-slash	ì
8		((1)		£
9	)	(1)		ï
10		*	(1)		ü
11		+	(1)		ä
12		,	(1)		Ö
13		-	(1)		ú
14		.	(1)		ó

Code number	Mode 0	Mode 1/2 standard	Mode 1/2 alternative	'International'
15		/ (1)		ö
16	club	Ø (1)	club	Ü
17		1 (1)		â
18	bold dash	2 (1)	bold dash	û
19	bold plus	3 (1)	bold plus	î
20	blob	4 (1)	blob	é
21		5 (1)		è
22		6 (1)		ñ
23		7 (1)		ê
24		8 (1)		â
25		9 (1)		à
26	:	(1)	:	À
27	ESCAPE symbol†	;	ESCAPE symbol	ESCAPE†
28	up-arrow†	< (1)	up-arrow	up-arrow†
29	down-arrow†	= (1)	down-arrow	down-arrow†
30	left-arrow†	> (1)	left-arrow	left-arrow†
31	right-arrow†	? (1)	right-arrow	right-arrow†
32	space	space (0)	heart	space
33	!	! (0)		!
34 (0)		..
35	#	# (0)		#
36	\$	\$ (0)		\$
37	%	% (0)		%
38	&	& (0)	bold slash	&
39	,	, (0)	bold back-slash	,
40	(((0)		(
41)) (0)	)
42	*	* (0)		*
43	+	+ (0)		+
44	.	. (0)		.
45	-	- (0)		-
46	.	. (0)		.
47	/	/ (0)		/
48	Ø	Ø (0)	club	0
49	1	1 (0)		1
50	2	2 (0)	bold dash	2
51	3	3 (0)	bold plus	3
52	4	4 (0)	blob	4
53	5	5 (0)		5
54	6	6 (0)		6
55	7	7 (0)		7
56	8	8 (0)		8
57	9	9 (0)		9
58	:	: (0)	:	:
59	;	; (0)	ESCAPE symbol	;
60	<	< (0)	up-arrow	<
61	=	= (0)	down-arrow	=
62	>	> (0)	left-arrow	>
63	?	? (0)	right-arrow	?
64	●	● (0)	diamond	●
65	A	A (0)	a	A
66	B	B (0)	b	B
67	C	C (0)	c	C
68	D	D (0)	d	D
69	E	E (0)	e	E
70	F	F (0)	f	F

Code number	Mode 0	Mode 1/2 standard	Mode 1/2 alternative	'International'
71	G	G (0)	g	G
72	H	H (0)	h	H
73	I	I (0)	i	I
74	J	J (0)	j	J
75	K	K (0)	k	K
76	L	L (0)	l	L
77	M	M (0)	m	M
78	N	N (0)	n	N
79	O	O (0)	o	O
80	P	P (0)	p	P
81	Q	Q (0)	q	Q
82	R	R (0)	r	R
83	S	S (0)	s	S
84	T	T (0)	t	T
85	U	U (0)	u	U
86	V	V (0)	v	V
87	W	W (0)	w	W
88	X	X (0)	x	X
89	Y	Y (0)	y	Y
90	Z	Z (0)	z	Z
91	[[(0)	spade	[
92	\	\ (0)	vertical line	\
93]] (0)	left-bend arrow]
94	^	^ (0)		^
95	-	- (0)		-
96	diamond	⦿ (1)	diamond	⦿
97	a	A (1)	a	a
98	b	B (1)	b	b
99	c	C (1)	c	c
100	d	D (1)	d	d
101	e	E (1)	e	e
102	f	F (1)	f	f
103	g	G (1)	g	g
104	h	H (1)	h	h
105	i	I (1)	i	i
106	j	J (1)	j	j
107	k	K (1)	k	k
108	l	L (1)	l	l
109	m	M (1)	m	m
110	n	N (1)	n	n
111	o	O (1)	o	o
112	p	P (1)	p	p
113	q	Q (1)	q	q
114	r	R (1)	r	r
115	s	S (1)	s	s
116	t	T (1)	t	t
117	u	U (1)	u	u
118	v	V (1)	v	v
119	w	W (1)	w	w
120	x	X (1)	x	x
121	y	Y (1)	y	y
122	z	Z (1)	z	z
123	spade	[(1)	spade	⦿
124	vertical line	(CLS) [†] (CLS) [†]	vertical line	vertical line
125	bent arrow	(CLS) [†] (CLS) [†]	(not used)	bent arrow (CLS) [†]
126	◀ (BS) [†]	^ (1)	◀	◀ (BS) [†]

Code number	Mode 0	Mode 1/2 standard	Mode 1/2 alternative	'International'
127	 (TAB) [†]	—	 (TAB) [†]	 (TAB) [†]
128	inverse heart	Space	heart	inverse á
129		!		inverse ù
130		"		inverse Ñ
131		#		inverse Ê
132		\$		inverse ç
133		%		inverse ô
134	inverse bold slash	&	bold slash	inverse ò
135	inverse bold back slash	,	bold back slash	inverse ì
136		(	inverse í
137	)		inverse î
138		*		inverse û
139		+		inverse ä
140		,		inverse Ö
141		-		inverse ú
142		.		inverse ó
143		/		inverse ö
144	inverse club	Ø	club	inverse ü
145		1		inverse â
146	inverse bold dash	2	bold dash	inverse û
147	inverse bold plus	3	bold plus	inverse ï
148	inverse blob	4	blob	inverse é
149		5		inverse è
150		6		inverse ñ
151		7		inverse ê
152		8		inverse á
153		9		inverse à
154		:		inverse Ä
155	end of line	space	heart	end of line
156	inverse up-arrow [†]	<	up-arrow	inverse up-arrow [†]
157	inverse down-arrow [†]	=	down-arrow	inverse down-arrow [†]
158	inverse left-arrow [†]	>	left-arrow	inverse left-arrow [†]
159	inverse right-arrow [†]	?	right-arrow	inverse right-arrow [†]
160	inverse space	space	heart	inverse space
161	inverse !	!		inverse !
162	inverse "	"		inverse "
163	inverse #	#		inverse #
164	inverse \$	\$		inverse \$
165	inverse %	%		inverse %
166	inverse &	&	bold slash	inverse &
167	inverse ,	,	bold back slash	inverse ,
168	inverse ((	inverse (
169	inverse))		inverse)
170	inverse *	*		inverse *
171	inverse +	+		inverse +
172	inverse ,	,		inverse ,
173	inverse -	-		inverse -
174	inverse .	.		inverse .
175	inverse /	/		inverse /
176	inverse Ø	Ø	club	inverse Ø
177	inverse 1	1		inverse 1

Code number	Mode 0	Mode 1/2 standard	Mode 1/2 alternative	'International'
178	inverse 2	2	(2) bold dash	inverse 2
179	inverse 3	3	(2) bold plus	inverse 3
180	inverse 4	4	(2) blob	inverse 4
181	inverse 5	5	(2)	inverse 5
182	inverse 6	6	(2) 	inverse 6
183	inverse 7	7	(2) 	inverse 7
184	inverse 8	8	(2) 	inverse 8
185	inverse 9	9	(2) 	inverse 9
186	inverse :	:	(2)	inverse :
187	inverse ;	;	(2) ESCAPE symbol	inverse ;
188	inverse <	<	(2) up-arrow	inverse <
189	inverse =	=	(2) down-arrow	inverse =
190	inverse >	>	(2) left-arrow	inverse >
191	inverse ?	?	(2) right-arrow	inverse ?
192	inverse @	@	(2) diamond	inverse @
193	inverse A	A	(2) a	inverse A
194	inverse B	B	(2) b	inverse B
195	inverse C	C	(2) c	inverse C
196	inverse D	D	(2) d	inverse D
197	inverse E	E	(2) e	inverse E
198	inverse F	F	(2) f	inverse F
199	inverse G	G	(2) g	inverse G
200	inverse H	H	(2) h	inverse H
201	inverse I	I	(2) i	inverse I
202	inverse J	J	(2) j	inverse J
203	inverse K	K	(2) k	inverse K
204	inverse L	L	(2) l	inverse L
205	inverse M	M	(2) m	inverse M
206	inverse N	N	(2) n	inverse N
207	inverse O	O	(2) o	inverse O
208	inverse P	P	(2) p	inverse P
209	inverse Q	Q	(2) q	inverse Q
210	inverse R	R	(2) r	inverse R
211	inverse S	S	(2) s	inverse S
212	inverse T	T	(2) t	inverse T
213	inverse U	U	(2) u	inverse U
214	inverse V	V	(2) v	inverse V
215	inverse W	W	(2) w	inverse W
216	inverse X	X	(2) x	inverse X
217	inverse Y	Y	(2) y	inverse Y
218	inverse Z	Z	(2) z	inverse Z
219	inverse [[(2) spade	inverse [
220	inverse \	\	(2) vertical line	inverse \
221	inverse]]	(2) bent arrow	inverse]
222	inverse ^	^	(2) 	inverse ^
223	inverse -	-	(2)	inverse -
224	inverse diamond	@	(3) diamond	inverse @
225	inverse a	A	(3) a	inverse a
226	inverse b	B	(3) b	inverse b
227	inverse c	C	(3) c	inverse c
228	inverse d	D	(3) d	inverse d
229	inverse e	E	(3) e	inverse e
230	inverse f	F	(3) f	inverse f
231	inverse g	G	(3) g	inverse g
232	inverse h	H	(3) h	inverse h
233	inverse i	I	(3) i	inverse i

Code number	Mode 0	Mode 1/2 standard	Mode 1/2 alternative	'International'
234	inverse j	J (3)	j	inverse j
235	inverse k	K (3)	k	inverse k
236	inverse l	L (3)	l	inverse l
237	inverse m	M (3)	m	inverse m
238	inverse n	N (3)	n	inverse n
239	inverse o	O (3)	o	inverse o
240	inverse p	P (3)	p	inverse p
241	inverse q	Q (3)	q	inverse q
242	inverse r	R (3)	r	inverse r
243	inverse s	S (3)	s	inverse s
244	inverse t	T (3)	t	inverse t
245	inverse u	U (3)	u	inverse u
246	inverse v	V (3)	v	inverse v
247	inverse w	W (3)	w	inverse w
248	inverse x	X (3)	x	inverse x
249	inverse y	Y (3)	y	inverse y
250	inverse z	Z (3)	z	inverse z
251	inverse spade	[spade	inverse Å
252	inverse vertical line	\ (3)	vertical line	inverse vertical line
253	inverse bent arrow (beep) [†]]	bent arrow	inverse bent arrow (beep) [†]
254		^		
255		-		

Appendix 6: To Err is Human

Perfect programmers never make mistakes, and the programs they write never fail, no matter what the user does. Perfect programmers don't exist. We mortals must do our best, but. . . .

If you *do* do your best with your programs, you'll plan and code action for all user actions there can be. At least, then, your programs' users will never cause failure and an incomprehensible ERROR report.

But *you* will meet ERROR reports and gradually you'll get to know them, even if not to love them. Well, you'll never get to know them all, unless your memory is inhumanly superb—so here's a list of the reports, their meanings, and, perhaps, some action you can take.

Error reports may appear when a command can't be carried out, or when the program comes to a statement it can't carry out. STATUS (ST.) gives you the reports too, but it's fiddly. You need to know what channel to use; I'll call it C—then enter ST.# C,R: ?R(R). I guess you won't often need STATUS.

Note that syntax errors (such as PRONT for PRINT) don't arise here—the micro kindly tells you off at once with an 'ERROR' display, and a cursor somewhere near where it thinks you went off the rails. Error reports signal only so-called RUN time failures. Anyway, here's the list. Number, meaning, comment. . . .

Code 2: You've run out of storage space

As in the program 1 GOSUB 2/2 GOSUB 1—there's not enough room for your program and its variables, or you've nested loops or closed subroutines too much.

ACTION—reduce messages; cut variable names; reduce the number of variable names; remove spaces; use multi-statement lines; don't have so many nested loops/routines. Or buy more RAM.

Code 3: An argument or such is off limits

As in ? SQR(−1)—many statements and functions may work with numbers only in certain ranges.

ACTION—watch out for this if those numbers are variables—ensure that the values stay within limits.

Code 4: You've run out of variables

You can't use more than 128. Well, who needs to?

ACTION—duplicate variable names (i.e, use the same names again later in the program); keeping variable name lists helps (the micro has to); use arrays more.

Code 5: You've treated a string as longer than it is

As in ?A\$(6,1) when A\$'s DIM is only 5. Or you may have tried to pull out a character sited before the start of a string. Or used a string before you dimensioned and/or assigned a value to it. Or things like that.

ACTION—watch out when you're taking substrings using variable names for the characters; just avoid the other faults.

Code 6: You ran out of DATA

You tried to READ when the DATA pointer had reached the end of the list. Or you hadn't put the DATA in. Or you pressed RETURN when the cursor was on the READY message—the silly micro thought you meant READ Y!

ACTION—check your READ and DATA lists and loops. Don't forget RESTORE.

Code 7: You used a number bigger than 32767

You can't always. Line numbers, for instance, mustn't exceed that limit.

ACTION—Learn by experience when this barrier exists.

Code 8: You tried to assign a string value to a numeric variable

For instance READ X found the pointer at a non-number, or INPUT X was met with a non-number (or (R) alone).

ACTION—check READ/DATA lines again (including looking for double commas in the latter). And mugtrap numeric INPUT statements.

Code 9: Something wrong with DIM somewhere

Perhaps you actually forgot to DIM a string or array before using it? More likely your program met a DIM so-and-so when so-and-so was already dimensioned. Some Code 5 errors appear under 9 too.

ACTION—Do all DIMs in initialization lines at the start of the program; if any depend on user input, ensure the program doesn't loop back over that section.

Code 10: You beat Atari with your argument

Functions can work on pretty complex arguments, but if they get too tough the micro can't cope.

ACTION—Work the expression out in a couple of steps before making it an argument.

Code 11: You tried to use a number bigger than 10⁹⁸

Unless you're an astronomer, this is probably because the micro had to divide by zero.

ACTION—don't force the poor thing to divide by zero, even if *you* can.

Code 12: The micro couldn't find the line number referred to

If you use numbers with GOSUBs, or use GOTOs, even if only implied—well, I note in this book it's bad practice. The Atari won't like it if the line number you sent it to doesn't exist. (Strangely, you can use TRAP N, where N is not in the listing with newer Ataris—the micro just ignores it.)

ACTION—Avoid GO TOs like whisky when you're coding late at night. Triple-check ON... lines and if you renumber program lines change GO addresses concerned too. Best name all such addresses then assign the names at the start of the program.

Code 13: The micro found a NEXT, but couldn't find the FOR

Most likely you've nested loops badly (nested loops must be fully inside each other like Russian dolls). But if you use POP inside a loop you may get ERROR 13. Unlucky for some.

ACTION—Take care when nesting loops. And I reckon you shouldn't POP anyway.

Code 14: The command/program line is too long

Or too complex for the poor thing. Commands/program lines shouldn't exceed three screen lines when entered.

ACTION—Don't overdo multi-statement lines; compress them; or POKE 82,0.

Code 15: The program's lost a GOSUB or FOR statement

Maybe you deleted it? Sometimes you get this when you use GOSUB N as a direct command.

ACTION—check for the missing GOSUB or FOR.

Code 16: The program can't think where to RETURN to

This is the subroutine's equivalent of ERROR 13. Most likely you forgot to protect your subroutine section with END (or STOP) before its start.

ACTION—Avoid entering subroutines except on purpose.

Code 17: The micro can't make head or tail of what it found

Either it came to a line starting ERROR (you forgot to correct a syntax slip on entry), or it found a CONTROL character where there shouldn't be one. But if you've been messing with POKes or machine code subroutines—either can degrade the listing in storage. If you can put hand on heart and deny all that, you've likely got a faulty RAM.

ACTION—try all else before putting hand on heart.

Code 18: VAL failed to carry out her task

As with ?VAL("LEN").

ACTION—Don't try to use VAL if the string argument could be non-numeric without mugtrapping first.

Code 19: LOADING program is too long for your RAM

ACTION—Swap it for a smaller program or buy more RAM.

Code 20: Use of an invalid channel number

Your program tried to access a channel outside the range 1–7.

ACTION—Don't let it.

Code 21: LOADING unacceptable material

Maybe you're trying to LOAD software saved with CSAVE or LIST? Or perhaps the recording wasn't at the beginning.

ACTION—Keep good records of cassette-counter readings and SAVE methods.

Code 128: BREAK used

This report shows the use of BREAK while the program was trying to output or input data.

ACTION—Well, these things happen.

Code 129: Program tried to open a channel open already

ACTION—Avoid this!

Code 130: The program couldn't figure out what channel to use

You may have used LIST "B:" or DRAWTO in a non-graphics mode.

ACTION—If not, check peripherals as for ERROR 138.

Code 131: Attempt to read from a write-only channel

You tried to use GET, INPUT or LOCATE from a channel used only for output.

Code 132: ERROR in XIO statement

ACTION—Check.

Code 133: Attempt to use a closed channel

As in ? # 6;... when you're not in program mode.

ACTION—Get into good habits at OPENing time.

Code 134: Use of an invalid channel number

Pretty much the same as ERROR 20.

Code 135: Attempt to write to a read-only channel

The converse of ERROR 131.

Code 136: The program tried to read data after getting end-of-file
More likely you pressed CONTROL + 3 by mistake.

Code 137: The program met a data block longer than 256 bytes
It shouldn't happen, but it's advanced work anyway.

Code 138: The micro gave up waiting for a peripheral
Perhaps you tried to CSAVE when the cassette machine wasn't ready.
ACTION—Check you've got the right peripheral joined up, switched on and ready for action. Rewind the cassette as well if that's in use.

Code 139: The peripheral didn't respond in the proper way
Could be a version of 138, or the data transfer is failing somewhere.
ACTION—As for 138.

Code 140: The micro can't understand the data
I'm afraid you have a dodgy record on cassette or disc.
ACTION—Always make back-up copies.

Code 141: You tried INPUT # 6;...
ACTION—If not, refer to ERROR 3.

Code 142: Version of 140

Code 143: Ditto
Or you may have started the cassette at the wrong place, in which case . . .
ACTION—Rewind

Code 144: The micro can't send data to or get data from the disc
Have you taped over the 'write-protect' hole? Otherwise, as for ERROR 140.

Code 145: The micro failed to verify the disc record
ACTION—Try SAVE again.

Code 146: The micro reckoned you've made a slip
The sort I mean is trying to send data to the keyboard, or get data from the printer.
ACTION—Don't.

Code 147: Your RAM is too small for your mode
ACTION—Use a less costly mode.

Code 150: Similar to 133

Code 151: So's this

Code 152: Buffer problem
Leave this to the engine-driver. Rare, anyway.

Code 153: Too much input/output going on
But you're not likely to meet this.

Code 154: Much the same

Code 160: You tried to use too many disc drives
You can access only four disc drives.

Code 161: You tried to keep too many data files open
The normal limit is three.

Code 162: You've filled the disc

ACTION—Obvious I guess.

Code 163: A catch-all for data errors the micro can't understand

Code 164: You tried to move the disc file pointer to a place you shouldn't

Code 165: Your pretty file name doesn't turn the micro on

ACTION—Follow the rules for file-names.

Code 166: Much the same as 164

Code 167: You tried to mess about with a locked file

But such disc software is heavily protected.

Code 168: XIO error

ACTION—Check your XIOing.

Code 169: You've run out of room in your disc directory

This can hold only 64 names.

ACTION—Minimize use of short files.

Code 170: The micro can't find a file with the name you quoted

ACTION—Check the directory. Maybe you got the name wrong, or put the wrong disc in.

Code 171: Version of 164

Code 172: You tried to append a DOS 1.0 file using DOS 2.0

(the 1.0 and 2.0 are versions of the Atari disc operating system).

ACTION—Copy the DOS 1.0 material onto a DOS 2.0 disc first.

Code 173: Media Fault

The disc operating system found surface problems when trying to format a disc.

ACTION—If this happens with several new discs, have your disc drive checked.

Code 218: Something wrong with your LOAD

ACTION—Act like Bruce's spider.

That's a huge list. If the force is with you, you'll meet very few of these—but you wouldn't like me to have missed any out, would you? I hope I haven't.

Appendix 7: Tipples

OK, the proper word for little tips is ‘wrinkles’ but who wants to show their age? Here’s a small collection of oddments you might find of use in your programs. No pattern to them, but all worth an explore.

BREAK-AWAY

To make your software more ‘robust’ (stop-proof) take these final steps. I say ‘final’ because until your program’s finished, *you* will want to be able to stop it, won’t you?

TRAP deals with action on error during a run. Don’t add TRAP until you’re certain you’ve covered all possible errors in the coding. Then TRAP will pick up the user’s attempts to halt the program as well as genuine mistakes he or she may make. If the TRAP subroutine closes with RUN, thus setting the trap again, you should be pretty safe.

Making the BREAK key stop working needs a double POKE. Use POKE 16,64 and POKE 53774,64 at the start and after each GRAPHICS statement.

In the same kind of way you can block use of the (SYSTEM) RESET button—POKE 580,1 causes this to produce a complete reset—just as if you switched the machine off and on again.

If you want to give the user the option of stopping the program but still have all those fences up to protect you, use a ‘Press ‘S’ to stop.’ routine. Then IF S\$ = “S” THEN NEW.

ANTIPODES

POKE 755,6 makes all the characters on screen upside-down, while POKE 755,2 brings you back to normal. Other values affect the cursor too—see Page 196.

I guess upside-down screen printing may have a value in certain games, but as the effect covers the whole screen it’s not that much use, surely. (The effect is the same in other modes, so you can get large upside-down characters if that turns you on.)

The trouble is, the characters in a string are upside-down but *not* back to front. Using PRINT # 6; “ollah” does *not* say “hallo” when you stand on your head. So all POKE 755,6 gives you really is yet another set of sets of characters to make patterns from.

HIDDEN INPUT

When I see things on screen like ‘Please type your name?’ I really squirm. (I’m afraid I’m a grammar pedant.) But it’s a bind having the question-mark prompt each time you use

an INPUT statement. And sometimes you don't want to display what the user types either. Open Channel 1 to the keyboard is the answer. Like this:

```
OPEN # 1,4,0; "K:"
```

Then when you want a hidden input, the trick is

```
PRINT "Please type your name!"  
INPUT # 1,I (or I$)
```

No prompt, no display of input data—you use the value of I, or I\$, when and where *you* want.

NICE OLD BUFFER

RUN this command, and while the micro's chasing electrons around its insides, press a few keys. No effect, of course, while the command's in action—but, when it stops, hey presto, there on screen is the character of the last key you pressed. Here's the command:

```
FOR A = 1 TO 2000: NEXT A
```

The Atari has a one-byte 'input buffer', a tiny little special storage site that holds the last key-press value. (A buffer is a store for holding data on the way between a peripheral and the processor or vice versa.)

One byte of store isn't great. (A dozen would be better.) But it has some uses. Press CTRL and 2 after you've RETURNed from a SAVE or LOAD command, for instance, and when the data transfer is done, the buzzer buzzes.

Now I think of it, that's just about the only single key-press you can have on the Atari that does anything worthwhile. A dozen or so input buffer bytes *would* be better.

NO JOY

Games listed in books or magazines often expect you to have a joystick to hand. If you haven't, you'll have to use the keyboard instead. Choose a group of nine keys, such as these:

Q	W	E
A	S	D
Z	X	C

(if you're left-handed)

or

I	O	P
K	L	;
,	.	/

(if you're on the right)

Assign the eight round the central L (or S) to the eight STICK directions, and make a space-bar give the trigger effect. Then re-write the printed code thus:
Add

```
LET I = PEEK(764): POKE 764,255
```

at the start of the main read-stick loop.
Replace

```
IF STICK(0) = 5... with IF I = 38... (left-hand 18)  
6 10 42  
7 2 58  
9 32 23  
10 13 47  
11 5 63  
13 34 22  
14 8 46  
15 255 255  
and IF STRIG(0) = 0 33 33
```

I'll leave you to use this method to convert paddle programs to use with the keyboard—but it's not so likely to be of value of course.

RANDOM RESTORE

As described on Page 170, the statement `RESTORE n` puts the data pointer at the first DATA item after line `n`. That leads us to the chance to pose the questions of a simple test at random. The program fragment below shows this in essence. Of course you can develop this into two or more DATA items (question/answer pairs) in each DATA line and build up nice long tests if you want!

If you want to block questions from second coming, you'll need to use subroutines or arrays to hold the data instead of DATA.

```
10 DIM A$(75),B$(30),R$(30)
50 FOR Q=1 TO 100:REM ** If you like long tests....
60 RESTORE 199+INT(RND(0)*5+0.5)
70 PRINT CHR$(125):READ A$,B$:PRINT "QUESTION ";Q:PRINT :PRINT :PRINT :PRINT A$;
:INPUT R$:POSITION 5,10
80 IF R$=B$ THEN PRINT "Correct!"
90 IF R$<>B$ THEN PRINT "Nope! Try ";B$
100 FOR W=1 TO 1000:NEXT W:NEXT Q
200 DATA What's my name,Tara
201 DATA Why smile,End of book
202 DATA On what day did Suleiman the Tata first enter Istanbul,Dunno
203 DATA Why,Why not?
204 DATA What is NaCl,salt
205 DATA What is 2+2,4
```

READING MUSIC

Here's a quick method of getting little tunes to beep out. All you need to do is to define the pitch and time of each note to go to the DATA block. Express the data for each note in order. Then READ and play them in order in a loop like this, the whole forming a subroutine.

```
10 LET TUNE1=1000:REM etc
999 REM ** Tune 1
1000 RESTORE TUNE1
1010 FOR N=1 TO NUMBEROFNOTES
1020 READ PITCH,TIME:SOUND 1,PITCH,10,10:FOR L=1 TO TIME*50:NEXT L:SOUND 1,0,0,0
:FOR W=1 TO 50:NEXT W
1030 NEXT N:RETURN
1100 DATA PITCH1,TIME1,PITCH2,TIME2,PITCH3,TIME3,etc
```

REPEAT AND WHILE

Some micros offer `REPEAT... UNTIL` and `WHILE... DO... ENDWHILE` loop structures in their BASICs as well as the simple `FOR... TO... NEXT` the Atari has. However, we can readily convert such structures into Atari BASIC.

1. `REPEAT... UNTIL` is used like this:

LET X = 1	or	REPEAT
REPEAT		INPUT ANSWER
LET X = X + 1		UNTIL ANSWER = RESULT
(set of instructions)		
UNTIL X = 100		

With the Atari we can use

```
FOR X = 1 TO 100      or  FOR A = 0 TO 1 STEP 0
(instructions)         INPUT ANSWER
NEXT X                 IF ANSWER = RESULT THEN LET A = 2
                        NEXT A
```

2. WHILE... DO appears comme ça:

```
WHILE Y > 0 DO
(set of instructions)
ENDWHILE
```

We can put it like this, to *almost* the same effect:

```
FOR A = 0 TO 1 STEP 0
(instructions)
IF Y > 0 THEN NEXT A
```

‘RIGHT’ AND ‘WRONG’ MESSAGES

It's somewhat dreary for a user to keep on getting the same micro-reactions to correct or incorrect answers. Randomize them a bit in a subroutine, like this:

```
LET YES = 1000
.
.
.
IF ANSWER = CORRECT THEN GOSUB YES
.
.
.
LET Y = RND(0)* 4
POSITION 10, 20
IF Y = 0 THEN PRINT "Yes,□"; NAME$; "!"
IF Y = 1 THEN PRINT "Well done. . . ."
IF Y = 2 THEN PRINT "Right,□"; NAME$; "."
IF Y = 3 THEN PRINT "Correct!"
RETURN
```

TABULATION

We often need to be able to tabulate things neatly in a vertical line—decimal points or equals signs, for instance. In the former, to put all the decimal points in column 25, use this routine, where NUMBER is the number concerned:

```
POSITION 25-LEN (STR$(INT(NUMBER)) ) , LINE: PRINT NUMBER
```

A rather similar, exceedingly useful, trick allows you to centre a string on the screen. If the string is X\$, use:

```
POSITION 19-LEN(X$)/2, LINE: PRINT X$
```

This won't work so well if LEN (X\$) is greater than 38 of course, but you shouldn't use it for that anyway!

WEEDY ARRAYS

Here's a useful use for DIM Y\$(1) where Y\$ is the Y/N response to some query:

```
DIM Y$(1)
.
.
.
FOR Y = 0 TO 1 STEP 0
PRINT "Again (Y or N)"; INPUT Y$
IF Y$ = "N" THEN STOP
IF Y$ < > "Y" THEN NEXT Y      (mugtrap)
RUN
```

Suitably amended as a subroutine, this is a lot simpler than many published ways of checking such responses.

FUNKING FUNCTIONS

The Atari follows traditional BASIC in offering only SIN, COS and ATN for its trigonometrical functions. You may need more, and while you may know you can get TAN with SIN/COS, others may need more research. I've done the research, so here are some more common trig functions you can build up.

$ACS(X)$ from $ATN(X/SQR(-X*X+1)) + 1.5708$

The angle in radians whose cosine is X; 'arccosine'.
X must lie between -1 and +1.

$ASN(X)$ from $ATN(X/SQR(-X*X+1))$

The angle in radians whose sine is X; 'arcsine'.
X must lie between -1 and +1.

$COT(X)$ from $COS(X)/SIN(X)$

The reciprocal of TAN(X); 'cotangent'.
X must not be zero.

$CSC(X)$ from $1/SIN(X)$

The reciprocal of SIN(X); 'cosecant'.
X must not be zero.

SEC(X) from $1/\text{COS}(X)$

The reciprocal of $\text{COS}(X)$; 'secant'.
 X must not be $\pi/2$ radians (180°).

TAN(X) from $\text{SIN}(X)/\text{COS}(X)$

The tangent of angle X .
 X must not be zero.

Appendix 8: Resources

America seems to dominate Britain in the Atari field, but then it's the other way round with other machines! You'll find that the vast majority of books, software and other resources have their origins in the US of A. Me, I believe in free trade—but this does mean that Brits find the extras rather pricy as well as fairly hard to find. The same is true in all countries outside North America.

Wherever you live and plug in your Atari, you'll need to search if you want extra help. In Britain, try Lasky's and W H Smith's in particular. Elsewhere scour the larger bookshops and personal computer suppliers.

MAGAZINES

The main British Atari users' magazine is *Input/Output*. This suffers from being infrequent and not independent—Atari bring it out quarterly. However its quality and value are increasing rapidly. It gives details of user groups—great.

There have been attempts to get independent magazines going, but none have really succeeded at the time of writing. *Page Six*, which appears each couple of months, is the best so far. There is a big enough number of users in this country surely. Meanwhile keep an eye on the main general home computing mags. Most nod on occasions in the Atari direction; *Your Computer*, *Personal Computer News* and *Personal Computer World* give the best coverage.

The USA has several magazines specific to Atari micros. *Antic* and *Analog* (both sounding more like science fiction!) have the best reputation, but are not easy to get hold of in other countries. Take a look at *Compute!* too if you get the chance—it covers several major US micros but gives a fair treatment of Atari.

BOOKS

I can recommend the following:

Albrecht, B. et al *Atari BASIC* (Wiley, 1979)—a fairly good teach-yourself guide.

Atari BASIC reference manual (Atari, 1983)—the sort of book other micro makers include with the machine!

Atari De re Atari (Atari, 1982)—advanced handbook.

Bunn, Paul *Making the most of your Atari* (Interface, 1983)—sometimes a bit superficial, but with lots of tips and listings.

Carlson, Edward *Kids and the Atari* (Prentice/Hall, 1983)—a pot-boiler, but still pretty good in being usable by youngsters learning Atari programming.

Carris, William *Inside Atari BASIC* (Prentice/Hall, 1983)—a very useful guide, especially for younger programmers.

Compute! *Second book of Atari* (Small System Services, 1982)—the best of their several collations of material from the magazine.

Goode, Peter *The Atari 600XL program book* (Pheonix, 1983)—loads of listings, games and more serious programs.

James, M. et al *The Atari book of games* (Granada, 1983)—lots more listings.

Kohl, H. et al *Atari games and recreations* (Prentice/Hall, 1982)—a very good BASIC guide.

Poole, L. et al *Your Atari computer* (McGraw-Hill), 1982)—generally excellent, but not for the beginner.

Sinclair, Ian *Get more from the Atari* (Granada, 1983)—a rather hasty introduction to BASIC.

Stanton, J. et al *Atari software* (Addison-Wesley, 1983)—a massive collection of reviews of North American products—games, business, education, utilities (and hardware add-ons).

Computer Bookshop (30 Lincoln Road Olton Warwickshire) is able to supply most, if not all, of the publications mentioned here. They send books out of Britain too. In case of difficulty in any aspect of Atari work, don't forget Atari themselves. Their customer services department (Railway Terrace, Slough, Bucks) is doubtless busy—but always friendly and helpful. The company has a number of duplicated handouts, on themes such as file-handling.

So—when you've finished this book, there's plenty more to help you get the most from your micro. But don't throw *me* away!

Index

Note that some of the references here are to structures and concepts that appear inside program listings. Note too that the cryptic code 'ff' means 'and the next page(s)'.

- abbreviations, 98, 188ff
- ABS, 133, 188
- accuracy, 130, 134
- ACS, 213
- ADA, 22
- ADR, 188
- algorithms, 115, 121
- ALO, 12, 128, 137
- amplitude, 75, 78
- AND, 91, 98, 106, 188
- animation, 57, 126, 183, 185
- ANTIC, 15
- argument, 132, 204, 205
- arithmetic, 19, 34, 42, 128ff, 186
- arrays, 151, 153, 171, 174ff, 181, 204, 211, 213
- ASC, 141, 188
- ASCII, 56, 139, 188, 197, 198ff
- ASN, 213
- assembly language, 22, 24
- assignment, 29, 31, 161, 190
- Atari Inc., 1, 4, 216
- ATN, 133, 188
- attract mode, 196
-
- Babbage, 42
- BACKSPACE key—see DELETE
- backing store, 13, 62
- back-up, 70
- bar-chart, 151
- BASIC, 5, 10, 21, 22, 100, 188ff, 194
- BBC, 22, 24
- beep, 41, 43, 55, 84
- binary numbers, 21, 45, 128, 130
- bits, 45
- books, 23, 179, 215ff
- brackets, 31, 106, 129
- BREAK key, 32, 33, 34, 39, 93, 94, 188, 189, 196, 206, 209
- brightness, 17, 46
- buffer, 210
- bugs, 72, 95, 130, 144ff
- BYE, 8, 94, 98, 172, 188
- byte, 45
-
- calculator, 11, 19, 24, 36, 128
- capitals, see upper case
- CAPS key, 9, 18, 55
- care, 70
- cartridge, 9, 13, 16
- case, see upper or lower
- case studies, 24ff
- cassettes, 13, 62ff, 67ff, 70, 168, 196, 206, 207, 210
- channel, screen, 58, 60, 206
- channel, sound, 75, 76
- characters, 9, 56, 137, 160, 161, 165, 180, 197, 198ff, 209
- chips, 14, 45
- chords, 75, 76
- CHRS, 17, 21, 41, 48, 50, 56, 60, 98, 141, 188, 198
- circles, 184
- cleaning, 63, 70
- CLEAR, 51, 52
-
- clearing screen, 11, 17, 20, 21, 27, 35, 43, 48, 52, 55, 94, 128, 151
- click, 78
- CLOAD, 65ff, 98, 188
- clock, 42, 167
- CLOG, 133, 188
- CLOSE, 189
- CLR, 138, 189
- COBOL, 22
- COLOUR, 79, 83, 86, 89, 90, 98, 156, 157, 189, 191, 198
- colour, 17, 44, 46ff, 59, 79, 83, 86ff, 91, 107, 155ff, 184, 186, 197
- COM, 138, 174, 189
- COMAL, 23, 194
- command, 19, 20, 93, 97
- computed address, 69, 108, 124, 186
- concatenation, 139, 140ff, 142
- constants, 29, 31, 137, 188
- CONT, 33, 34, 78, 93, 94, 95, 98, 189
- contrast, 20
- CONTROL key, 17, 28, 36, 42, 54ff, 60, 94, 96, 159, 197, 206, 207, 210
- control unit, 12
- copyright, 70
- COS, 133, 189
- COT, 213
- counting, 31, 37ff
- CPRINT, 71
- CSC, 213
- cursor, 9, 11, 16, 18, 43, 50, 55, 98, 196, 197
- cursor control, 96, 182
-
- DATA, 56, 137, 161ff, 165, 189, 192, 205, 211
- data processing, 12, 24ff, 28, 31, 128, 168, 169, 171, 174ff
- debugging, 144ff
- decisions, 102
- DEG, 133, 135, 189, 196
- delay, 30, 41, 43, 53, 123, 197
- deleting lines, 20, 38, 96
- DELETE key, 9, 11, 20, 55, 97, 100
- design, 158
- development, program, 72, 110ff, 144ff
- dialect, 23, 194
- digital, 12, 21
- DIM, 27, 30, 31, 32, 54, 98, 138, 151, 174, 178, 189, 193, 204, 205
- direct mode, 19
- discs, 62, 69, 168, 191, 195, 207ff
- display, see TV
- distortion, 77
- documentation, 67ff, 118
- DOS, 189
- DRAWTO, 84ff, 87, 98, 154ff, 189, 196, 206
- duration, sound, 75
- dust, 70
-
- editing, 96ff, 99, 117
- effects, sound, 81, 187
- efficiency, 72
- electronics, 12, 21

element, 174
 English, 22
 END, 75, 76, 98, 125, 189, 206
 ENTER, 65ff, 98, 189, 190
 ERROR, 9, 11, 16, 29, 30, 40, 58, 64, 66, 85, 98, 117, 145, 193, 196, 204ff
 ESC key, 51, 52, 198
 execution, 20
 Exp, 133, 190
 exponentiation, 42, 128, 135
 expressions, 29, 31, 129, 132, 205

 FALSE, 108
 filing, 216
 filling, 155ff
 flag, 106, 109, 151, 153, 154, 177
 flowchart, 33, 38, 103, 110ff, 115, 119, 120, 121, 146
 FOR, 38ff, 53, 98, 102, 190, 205
 form-filling, 141, 171
 format, 27, 35, 38, 42, 47, 48, 52, 54, 69, 95, 107, 116, 128, 131, 139, 142, 174, 187, 196, 212
 ForTran, 22, 23
 FRE, 100, 135, 190
 frequency, 74
 front panel, 21
 function keys, 171, 176
 functions, 31, 53, 130, 132ff, 141ff, 204, 213ff

 general input/output, see XIO
 GET, 79, 91, 98, 141, 184, 186, 190, 197, 206, 210
 GOSUB, 41, 67, 69, 98, 122ff, 190, 205
 GOTO, 11, 27, 33, 37, 38, 61, 69, 95, 98, 100, 103, 104, 108, 117, 124, 125, 145, 162, 190, 205
 graphics, 22, 23, 34, 36, 43, 51, 55, 57, 60, 61, 79, 83ff, 126, 150ff, 159ff, 184, 187, 190, 198ff
 GRAPHICS, 43, 85ff, 98, 150, 190, 209
 GTIA, 14
 guarantee, 6, 14

 HELP key 190, 197
 high-level, 22, 24, 194
 histogram, 151
 history, 1, 4, 12, 21, 42

 IF, 33, 34, 41, 98, 102, 104ff, 190
 immediate mode, 19
 increment, 31, 39
 indexing, 68ff
 inequalities, 33
 initialization, 116, 205
 input, 13, 25ff
 INPUT, 27, 28, 29, 31, 32, 98, 99, 128ff, 137, 141, 177, 184, 190, 205, 206, 207, 209
 INSERT, 92
 interfacing, 168ff
 INT, 32, 79, 84, 98, 106, 117, 131, 132, 133ff, 190
 interference, 14, 78
 interval, 77
 inverse, 11, 36, 50, 55, 60, 73, 149, 180, 197
 italic, 11
 item, print, 48

 joystick, 169ff, 192, 210

 K, 45
 keyboard, 9, 13, 70, 196, 197, 210
 keyword, 21, 23, 45, 98
 Kilobyte, 43

 languages, 21, 24, 194
 layout, screen, see format
 layout, system, 70
 LEN, 53, 98, 140, 141, 190, 213
 LET, 27, 31
 library, 63, 66, 67, 98
 light pen, 196
 line length, 41, 99, 205
 line numbers, 19, 20, 33, 67, 100, 102, 115, 124, 137, 189, 205
 lists, 174
 LIST, 11, 20, 21, 41, 42, 45, 63ff, 66ff, 94, 98, 149, 189, 190, 206
 LOAD, 65ff, 69, 98, 190, 206, 208
 LOCATE, 190, 206
 LOG, 134, 190
 logarithms, 133, 134

logic, 34, 106ff, 131, 137, 183, 186, 190, 191
 looping, 37ff, 53, 61, 102ff, 175, 190, 211
 loudness, 74
 lower case, 9, 18, 60
 LPRINT, 188, 190

 machine code, 22, 23, 24, 159, 194, 206
 magazines, 23, 179, 215
 mag-fields, 70
 mainframe, 12, 13
 margin, 51, 52, 99, 196, 205
 mark sensing, 26ff
 matrix, 178
 memory and saving it, 30, 43, 44ff, 65, 69, 98ff, 107, 118, 123, 138, 196, 204
 memory map, 115, 124
 menu, 176
 merging, 66, 189
 messages, 18, 19, 30, 47, 204, 212
 microelectronics, 12
 modes, 37, 42ff, 45ff, 57ff, 61, 87ff, 99, 128, 150, 155, 156ff, 189, 190, 191, 196, 198
 module, 22, 113, 115, 121, 149, 180
 monitor, 74
 mug-traps, 53, 69, 98, 117, 123, 131, 186, 204, 205, 213
 multi-statements, 41, 98ff, 104, 149, 204, 205
 music, 75, 80, 211

 names, 29ff, 63, 64, 124, 126, 138, 157, 204, 208
 nesting, 40, 123, 204, 205
 NEW, 11, 20, 21, 65, 75, 94, 98, 133, 191, 209
 NEXT, 38ff, 98, 191
 nibble, 45
 noise, 78
 NOT, 106, 109, 191
 NOTE, 191
 numbers, 129, 205
 numeric data, 28ff, 99

 office, 171
 ON, 124, 191, 205
 OPEN, 79, 91, 98, 141, 191, 206, 210
 operators, 34, 106, 139
 OPTION key, 171ff, 191
 OR, 54, 98, 106, 140, 191
 output, 13, 25ff

 paddle, 171, 191, 211
 PADDLE, 171, 191
 parameter, 17
 pattern, 54, 143, 209
 pause, see delay
 PEEK, 56, 149, 153, 159ff, 171, 174, 191, 194ff
 peripherals, 13, 63, 94, 206, 207
 permanent set-up, 70
 picture, 34, 54, 61, 158
 pitch, 75, 76
 pixel, 91
 planning, 72, 110
 player/missile graphics, 57, 159ff
 PLOT, 84ff, 87, 95, 98, 153, 155, 189, 191
 POINT, 191
 POKE, 52, 53, 61, 79, 81, 96, 98, 99, 123, 155, 157, 159ff, 184, 191, 194ff, 198, 209
 POKEY, 15
 POP, 125, 126, 191, 205
 POSITION, 9, 18, 21, 27, 45ff, 52, 53, 58, 59, 98, 155ff, 186, 191
 power, 7, 94
 powers, 19, 42, 129, 135
 PRINT, 9, 10, 17, 18, 19, 21, 24, 31, 37ff, 47ff, 52ff, 95, 98, 146, 191, 209
 printer, 4, 13, 25, 34, 66, 169, 190, 197
 print item, 47
 priority, 31, 34, 129ff
 processor, 12
 program, 10, 12, 19, 20
 programming, 2, 21, 34, 72, 110ff, 204
 prompt, 28, 29, 30, 141
 PTRIG, 171, 191
 PUT, 191

 quality, 74
 quote marks, see speech marks

RAD. 133, 192, 196
 radian, 133
 RAM. 45, 195, 196, 197, 206
 random access, 45
 random numbers, *see* RND
 READ, 31, 56, 160, 161ff, 178, 192, 205
 READY, 46, 58, 64, 98
 registers, 47, 59, 86, 89, 157
 recording, *see* LOAD and SAVE
 records, *see* documentation
 REM, 27, 34, 56, 96, 98, 116, 117, 118, 125, 149, 192
 renumbering, 205
 repeat, key, 100, 197
 REPEAT, 211
 RESET key, 9, 11, 35, 43, 52, 75, 93, 94, 133, 189, 192, 196, 209
 resolution, 44, 87, 153, 160
 RESTORE, 161, 170, 187, 189, 192, 211
 RETURN, 41, 67, 98, 116, 122ff, 192, 206
 RETURN key, 9, 11, 16, 20, 21, 54, 97, 141, 192
 RND, 32, 53, 79, 83, 98, 135, 146, 149, 192, 212
 rogue value, 114
 ROM, 45, 194ff
 rounding, 118, 130, 133ff
 routine, 72
 RUN, 11, 20, 33, 35, 65ff, 75, 95, 98, 116, 133, 162, 192, 209

 SAVE, 63ff, 98, 192
 screen, *see* modes
 screen clear, *see* clearing screen
 scroll, 28, 42, 46, 55, 94, 97, 196, 197
 searching, 175
 SEC, 214
 secondary tones, 77
 SELECT key, 9, 171ff, 192
 Selmor Engineering, 70
 SETCOLOR, 9, 11, 16, 17, 20, 21, 27, 46ff, 53, 60, 79, 84, 86, 89, 96, 98, 157, 192, 197
 SGN, 134, 192
 SHIFT keys, 9
 SIN, 134, 192
 Sinclair, 22, 24, 194
 sound, 8, 42, 74ff, 196, 197, 210, 211
 SOUND, 10, 55, 74ff, 84, 98, 178, 187, 192
 sound effects, 81, 107
 sound track, 168
 sorting, 175ff
 spaghetti, 113
 specification, 113
 speech marks, 18, 28, 47, 51, 137, 162
 speed, 12
 spreadsheet, 171
 sprites, 57, 159ff, 196
 SQR, 132, 134, 192
 stack, 39, 123
 START key, 125, 171
 statements, 20
 STATUS, 192, 204
 STEP, 39, 42, 98
 stop, 154
 STOP, 93, 95, 98, 116, 123, 125, 145, 189, 192, 196
 storage, 12, 19, 20, 44ff

STR\$, 141, 193, 212
 STRIG, 169, 193, 210
 string data, 18, 19, 28ff, 52, 137ff, 162, 178, 183ff, 185ff, 187, 188, 204, 213
 structure, 33, 99ff, 103, 113, 190
 subroutines, 41, 66, 72, 80, 115, 121ff, 142, 206
 subscript, 175
 substrings, 139, 140ff, 181, 204
 switch, 94
 syntax, 204
 synthesizing, 75, 79
 SYSTEM key, *see* RESET

TAB and its key, 50, 52, 55, 97, 196, 197, 212
 tables, 174, 177
 TAN, 214
 testing, 9, 73, 93, 104, 115, 118, 125, 146ff
 text modes, 43, 45ff, 57ff
 THEN, 33, 98, 104ff, 193
 timbre, 75, 77
 timing, 41, 167ff, 186, 196
 TO, 38ff, 98, 193
 top-down, 72, 110, 113ff, 123, 125
 tracing, 95, 146
 TRAP, 117, 129, 132, 149, 193, 197, 205, 209
 triggers, 169, 171
 TRUE, 108
 TV, 6, 7, 13, 17, 46, 70, 71, 74, 157

upper case, 9, 18, 21, 29, 54, 60, 73, 138
 user-friendliness, 30, 33, 48, 54, 73, 107, 114, 118, 128
 USR, 193, 194

VAL, 69, 117, 131, 141, 151, 193, 206
 variables, 29, 31, 47, 94, 138, 204
 verifying, 64, 207
 voice, 25

WHILE, 211
 windows, 43, 57, 60, 85, 90, 92, 196ff
 word, 45, 56
 word-processing, 97, 100, 138, 169
 write protecting, 65

XIO, 155ff, 193, 196, 197, 206, 208

zero, 129
 zones, 48, 51

£ 59
 \$ 29, 30, 138
 + 19, 29
 , 48, 50, 51, 52, 99, 128, 196
 / 19, 29
 : 41
 ; 32, 48, 52
 < 33
 = 108, 109
 > 33
 < > 33
 ^ 19, 31, 42, 128

Other titles of interest

Easy Programming for the Commodore 64 £6.95
Ian Stewart & Robin Jones

Easy Programming for the ZX Spectrum £5.95
Ian Stewart & Robin Jones

'... will take you a long way into the mysteries of the Spectrum: is written with a consistent and humorous hand: and shares the affection the authors feel for the computer'—*ZX Computing*

Programming for REAL Beginners: Stage 1 £3.95
Philip Crookall

Programming for REAL Beginners: Stage 2 £3.95
Philip Crookall

Brainteasers for BASIC Computers £4.95
Gordon Lee

'Just the job for a wet afternoon with the computing class'—*Education Equipment*

Computing: A Bug's Eye View £2.95
Cosgrove

A collection of the humorous cartoons that have appeared in 'Shiva's Friendly Micro Series'.

Easy Programming for the BBC Micro £5.95
Eric Deeson

'A beginners guide in the true sense'—*Educational Computing*

Easy Programming for the Electron £5.95
Eric Deeson

Easy Programming for the Oric-1 £5.95
Ian Stewart & Robin Jones

Easy Programming for the Dragon 32 £5.95
Ian Stewart & Robin Jones

Please write for our complete catalogue of computer publications and software

Attention all Atari owners!

Are you an 'apprentice' at programming your micro?
Let Eric Deeson be your guide through the BASIC maze.

From opening the box to writing, saving and running your
own programs, this book will tell you all you need to know,
including:

- setting up
- printing
- recording
- planning
- sound
- graphics

There are over 60 programs complete and ready to RUN
throughout the book, with ideas for do-it-yourself projects
at the end of most chapters.

A number of Appendices cover BASIC keywords, hints
on debugging, the Atari character set, PEEKing the Atari
memory map and further sources of information.

*This volume is suitable for the 600XL, 800XL and 1450XLD
models.*



Shiva Publishing Limited

GB £ NET +006.95

ISBN 1-85014-022-7



9 781850 140221