

# GET MORE FROM THE ATARI



IAN SINCLAIR

SUITABLE  
ALSO FOR THE  
NEW ATARI  
600XL

**Other books of interest from Granada:**

**The Apple II**

**APPLE II  
PROGRAMMERS  
HANDBOOK**

R. C. Vile  
0 246 12027 4

**ATARI**

**GET MORE FROM  
THE ATARI**

Ian Sinclair  
0 246 12149 1

**THE ATARI  
BOOK OF GAMES**

M. James, S. M. Gee  
and K. Ewbank  
0 246 12277 3

**The BBC Micro**

**INTRODUCING  
THE BBC MICRO**

Ian Sinclair  
0 246 12146 7

**THE BBC MICRO—  
AN EXPERT GUIDE**

Mike James  
0 246 12014 2

**21 GAMES FOR  
THE BBC MICRO**

M. James, S. M. Gee  
and K. Ewbank  
0 246 12103 3

**BBC MICRO  
GRAPHICS AND SOUND**

Steve Money  
0 246 12156 4

**DISCOVERING  
BBC MICRO  
MACHINE CODE**

A. P. Stephenson  
0 246 12160 2

**THE BBC BASIC  
PROGRAMMER**

S. M. Gee and  
M. James  
0 246 12158 0

**The Colour Genie**

**MASTERING THE  
COLOUR GENIE**

Ian Sinclair  
0 246 12190 4

**The Commodore 64**

**COMMODORE 64  
COMPUTING**

Ian Sinclair  
0 246 12030 4

**THE COMMODORE 64  
GAMES BOOK**

Owen Bishop  
0 246 12258 7

**SOFTWARE 64  
Practical Programs for  
the Commodore 64**

Owen Bishop  
0 246 12266 8

**The Dragon 32**

**THE DRAGON 32  
And How to Make  
The Most Of It**

Ian Sinclair  
0 246 12114 9

**THE DRAGON 32  
BOOK OF GAMES**

M. James, S. M. Gee  
and K. Ewbank  
0 246 12102 5

**THE DRAGON  
PROGRAMMER**

S. M. Gee  
0 246 12133 5

**DRAGON GRAPHICS  
AND SOUND**

Steve Money  
0 246 12147 5

**THE DRAGON 32  
How to Use and  
Program**

Ian Sinclair  
0 586 06103 7

**The IBM Personal  
Computer**

**THE IBM PERSONAL  
COMPUTER**

James Aitken  
0 246 12151 3

**The Jupiter Ace**

**THE JUPITER ACE**

Owen Bishop  
0 246 12197 1

**The Lynx**

**LYNX COMPUTING**

Ian Sinclair  
0 246 12131 9

**The NewBrain**

**THE NEWBRAIN  
And How To Make  
The Most Of It**

Francis Samish  
0 246 12232 3

**The ORIC-1**

**THE ORIC-1  
And How To Get  
The Most From It**

Ian Sinclair  
0 246 12130 0

**THE ORIC-1  
BOOK OF GAMES**

M. James, S. M. Gee  
and K. Ewbank  
0 246 12155 6

**THE ORIC-1  
PROGRAMMER**

M. James and S. M. Gee  
0 246 12157 2

**ORIC MACHINE CODE  
HANDBOOK**

Paul Kaufman  
0 246 12150 5

Continued on inside back cover

**Get More From  
The Atari**

# **Get More From The Atari**

**Ian Sinclair**

**GRANADA**

London Toronto Sydney New York

# Contents

<i>Preface</i>	vii
1 Setting Up The Machine	1
2 Words On The Screen	13
3 A Bit Of Variation	23
4 Repetitions And Decisions	36
5 String Up Your Programs	51
6 Filing And Planning	65
7 The Coarser Characters	80
8 More Resolution	98
9 Sounding Out The Atari	124
10 Odds And Ends	136
<i>Appendix A: SAVE and LOAD Problems</i>	141
<i>Appendix B: Useful Addresses</i>	143
<i>Index</i>	145

# Preface

The Atari 400 and 800 models are now well-established and respected personal computers. Recent price cuts, and the provision of 48K of memory on all the 800 models have made these well-designed computers very much more competitive in the UK. This has highlighted the need for a comprehensive guide for the beginner to Atari programming. When the Atari models first appeared, the BASIC language cartridge which enabled the user to program the machine for himself/herself was sold as an extra. Because of the vast array of software which could be bought for the Atari models, there was little incentive for most owners to write their own programs. In addition, some of the most fascinating actions of the computer were not even hinted at in the manual.

This book is aimed at the beginner who has just acquired an Atari 400 or 800, but it should be of considerable service to the established owner of an Atari who has never tried programming. Programming for both models is identical, and the main differences between the 400 and the 800 are the keyboards, and the provision of an extra cartridge slot in the 800. Everything in this book, therefore, refers equally to both the 400 and the 800 computers. I am sure that the text and examples here will provide a welcome source of information for the beginner. I hope also that the more seasoned user will find much of interest, and perhaps a few welcome surprises in these pages.

Ian Sinclair

## Chapter One

# Setting Up The Machine

By the time that you read this, you will have found that the Atari is a very solidly constructed machine, with a strong diecast chassis, and certainly no lightweight chunk of plastic. It comes much better prepared for service than most other machines, so that you can get it going for you very quickly. A computer is a more complicated device than a kettle or a toaster, however, and though the mains plug is already connected to your Atari, computing is not just a matter of plugging in and switching on.

First you must find a space large enough to take the Atari, along with everything you are likely to use with it. That will include a TV receiver and, almost certainly, the Atari program recorder (a form of cassette recorder). Later you may want to add disk drives, a printer and other goodies, so that space may be important. The only way I can manage to have several computers in one room is by using specially made stands, and the type made by Selmor (Fig. 1.1) is the best I have come across. If you are not quite at that stage yet, then make sure that you have a good-sized table or desk to work on.

With that hurdle over, you are almost ready to work some Atari wizardry, but you need the use of a TV receiver. A computer is a device which is arranged so as to send signals to a TV receiver, and unless you connect a TV receiver to the Atari you won't be able to see what the Atari is doing. It will still compute for you just as well, but you won't see what is going on.

Unlike most small computers, the Atari comes with its TV cable already attached and with an aerial plug at the end of the lead. You could, of course, simply plug this lead into the TV receiver, but a better option is to use the type of 2-to-1 adaptor that is illustrated in Fig. 1.2. This allows you to keep an aerial cable plugged in, and to connect or disconnect the Atari as you wish without disturbing the TV receiver. It's useful if you have to share a colour TV with the family. It also saves wear on the aerial connector of the TV receiver

2 Get More From The Atari

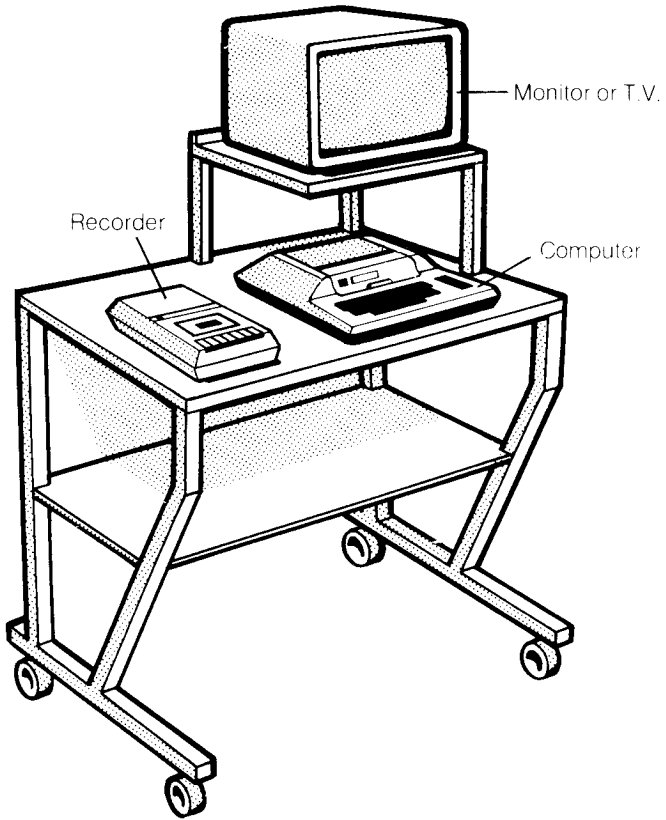


Fig. 1.1. The Selmor computer stand in use.

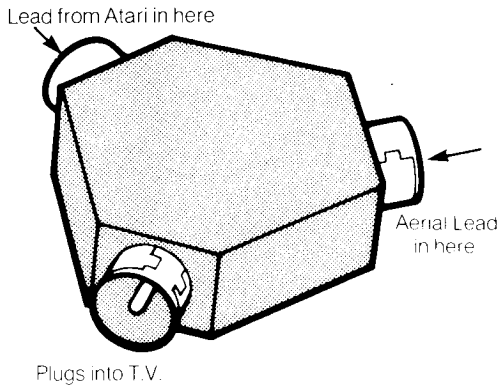


Fig. 1.2. A useful 2-to-1 TV adaptor. One of these is included with your Atari.



itself. This device is packed with your Atari, and its inclusion is typical of the way in which the Atari is made ready for you.

The TV that you use to display the Atari's signals need not be a colour receiver, not to start with at least. The skills of programming an Atari do not require you to see the results in colour until you come to the colour instructions of the Atari in Chapter 7. The signals which appear in colour on a colour TV will appear in shades of grey on a black/white TV so that you need not feel that you are missing anything essential if you have only a portable B/W TV available. Nevertheless, the colour signals that the Atari can produce are so outstandingly good that you should try to have a colour TV available when you come to Chapters 7 and 8 of this book.

### The big switch-on

Now before you plug in everything in sight and switch on, it's a good idea to see how many mains sockets you have around. When you are in full control of your Atari you will need three mains sockets. Two of these will be for the Atari and the TV receiver, but you will need one more for the program recorder. Most houses have desperately few sockets fitted, so you will find it worth while to buy or make up an extension lead that consists of a three- or four-way socket strip with a cable and a plug (Fig. 1.3). This avoids a lot of what the famous advert calls 'spaghetti hanging out the back'. Don't rely on the old-fashioned type of three-way adaptor - they never produce really reliable contacts. The Atari has its own mains switch, but this does not switch the mains supply. The mains plug connects to a power-pack, which converts the high mains voltage into a low (and

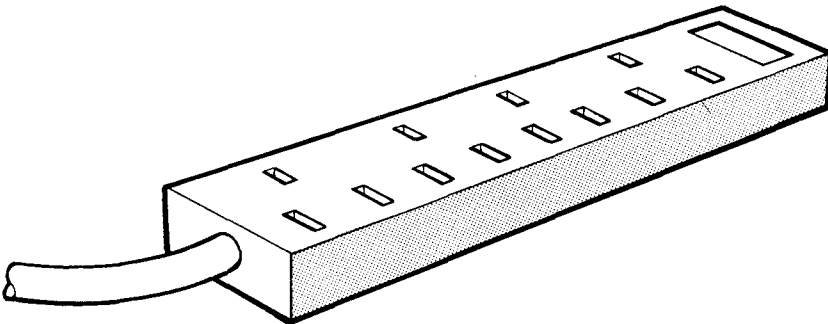


Fig. 1.3. A four-way socket strip which avoids the use of the old-style adaptors.

#### 4 Get More From The Atari

safe) voltage supply for the Atari. This low voltage supply is taken along a lead which has a small tubular plug fitted to it. This plugs into the power input socket at the right-hand side of the Atari. When you switch off the Atari, using the switch at this right-hand side, you do not switch off the mains. It's always a good idea to pull out the mains plug after you have switched off at the machine. In addition, you should make sure that the power pack is resting in a well-ventilated place, because it will become warm while it is working.

Before you can start to do any serious work with programming your Atari, you will need to instal the BASIC cartridge, if this has not already been done by your dealer. At the top edge of the keyboard, you will see a small catch that is marked PULL OPEN. Pull it towards you to release the catch, and you can lift the flap which is just behind it. There are two slots under this flap. The BASIC cartridge plugs into the left-hand slot, with the title of the cartridge facing you (see Fig. 1.4). The next step, now, is to shut the

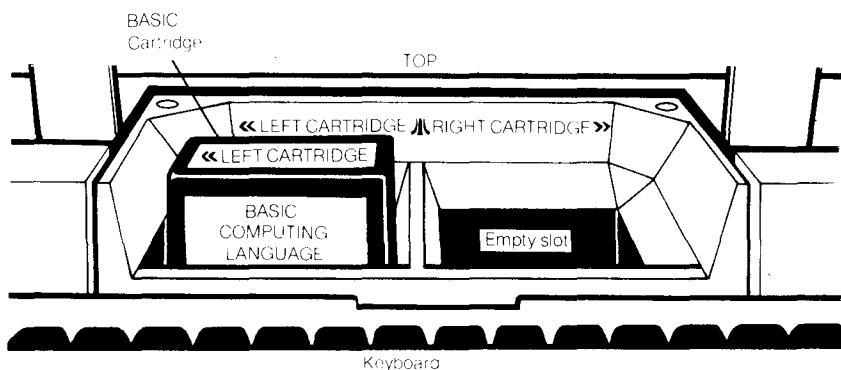


Fig. 1.4. Plugging in the BASIC cartridge. This is illustrated for the Model 800, in which the left-hand cartridge socket must be used.

flap again, and switch on the TV receiver and the Atari. The small built-in loudspeaker of the Atari remains silent as you switch the machine on. It's job is to deliver warning messages when something goes wrong. It doesn't, however, deliver the main sound effects. These are played through the loudspeaker of the TV so that you have full control over the volume.

An ordinary domestic TV is not ideal for viewing the Atari (or any other computer) signals. This is because the signals cannot be sent directly to the TV in the form that would give a clear picture. Instead, they have to be transmitted, using a miniature transmitter

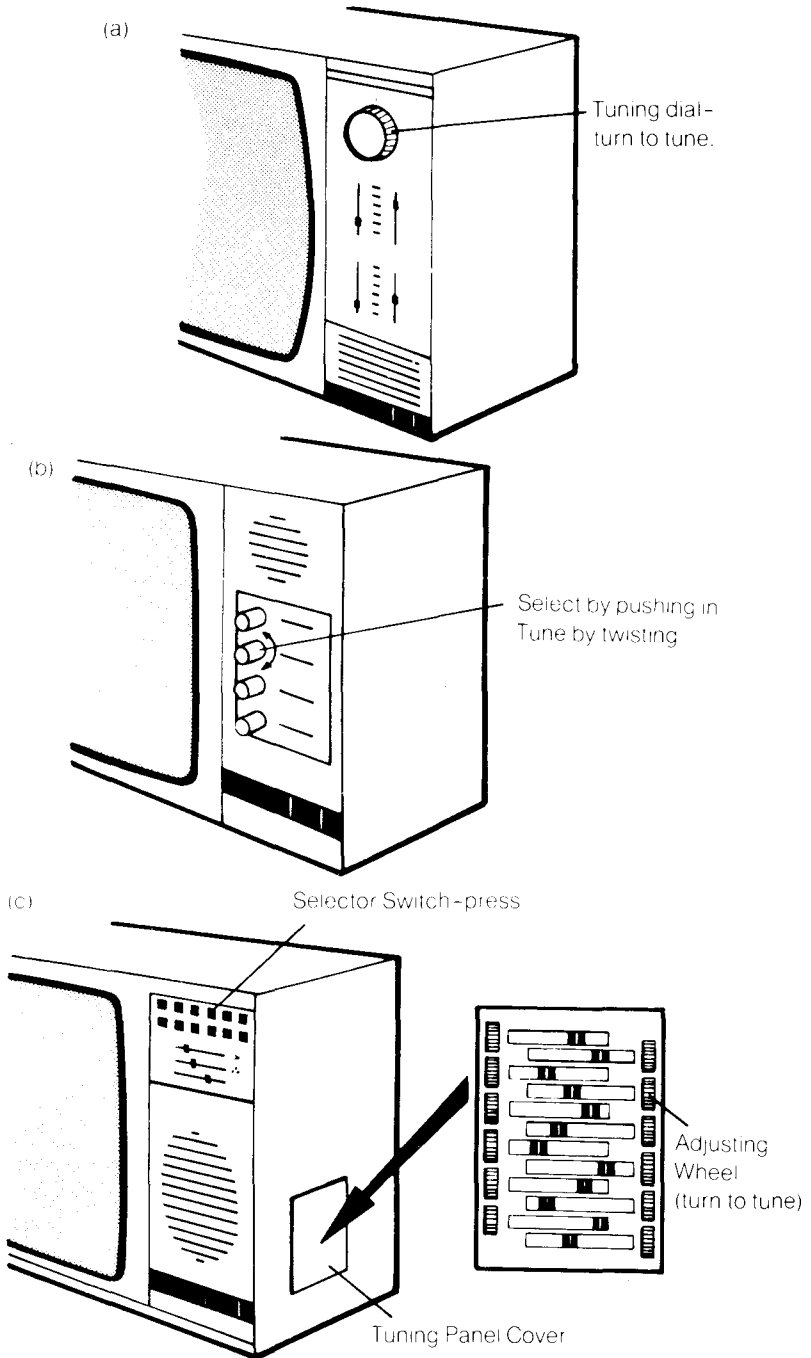
that is called a *modulator*. This is because most TV receivers cannot be connected safely to anything except by the aerial lead. Very much clearer pictures can be obtained by using what is called a 'monitor'. This is a form of stripped-down TV which can't receive broadcast signals (no licence needed!), but which can be connected safely to the Atari, and to a few other types of computers, to show high quality pictures. If you are lucky enough to see a demonstration of Atari signals displayed on a colour monitor you will get some idea of how much is lost when a modulator and an ordinary colour TV has to be used. You can buy a special cable for fitting a monitor to your Atari if you are so fortunate as to have a monitor available.

The second point is that a TV receiver has to be tuned to the signal from the Atari. Unless you have been using a video cassette recorder, and the TV has a tuning button that is marked 'VCR' it's unlikely that you will be able to get the Atari tuning signal to appear on the screen of the TV simply by pressing tuning buttons. The next step, then, is to tune the TV to the Atari's signals. Plug in the mains plugs of the Atari and the TV, making sure that the small plug from the Atari power supply is firmly inserted into its socket. Switch on the TV and turn down the volume control so that you are not distracted by the noise. Switch on the Atari, using the rocker-switch at the right-hand side of the machine.

Figure 1.5 shows the three main methods that are used for tuning TV receivers in this country. The simplest type is the dial tuning system that is illustrated in Fig. 1.5(a). This is the type of tuning system that you find on black/white portables, and you only have to turn the dial to get the Atari's signal on the screen. If the dial is marked with numbers, then you should look for the signal somewhere between numbers 30 and 40. If the dial isn't marked, which is unusual, then start with the dial turned fully anticlockwise as far as it will go, and slowly turn it clockwise until you see the Atari signal appear. If you turn the volume control up slightly so that you can hear the rushing noise of the untuned receiver, you will hear things go quiet as the Atari signal appears. You may find that there is some reduction in the sound level as you tune to a local TV transmission, but you'll notice the difference. The Atari doesn't give you the sound of Coronation Street!

What you are looking for, if the Atari hasn't been touched since you switched it on, is the word **READY** on the screen. When you can see this word, turn the dial carefully, turning slightly in each direction until you find a setting in which the words are really clear. On a TV receiver, particularly a colour TV, the words may never be

## 6 Get More From The Atari



*Fig. 1.5.* TV tuning controls. (a) Single dial, as used on black and white portables, (b) four-button type, (c) the more modern touch-pad or miniature switch type.

particularly clear, but get them steady at least and as clear as possible. If you don't have the BASIC cartridge installed at this stage, all you'll see (on a colour receiver) is a blue background.

The older types of colour and B/W TV receivers used mechanical push-buttons (Fig. 1.5(b)) which engage with a loud clonk when you push them. There are usually four of these buttons, and you'll need to use a spare one which for most of us means the fourth one. Push this one in fully. Tuning is now carried out by rotating this button. Try rotating anticlockwise first of all, and don't be surprised by how many times you can turn the button before it comes to a stop. If you tune to the Atari's signal during this time, you'll see and hear the same signs – the message on the screen and the reduction in the noise from the loudspeaker. If you've turned the button all the way anticlockwise and not seen the tuning signal, then you'll have to turn it in the opposite direction, clockwise, until you do. If you can't find the Atari signal at any setting, check the TV using an aerial in case there is something wrong with the tuning of the TV.

Modern TV receivers are equipped with touch pads or very small push-buttons for selecting transmissions. These are used for selection only, not for tuning. The tuning is carried out by a set of miniature knobs or wheels that are located behind a panel which may be at the side or at the front of the receiver (Fig. 1.5(c)). The buttons or touch pads are usually numbered, and corresponding numbers are marked on the tuning wheels or knobs. Use the highest number available (usually 6 or 12), press the pad or button for this number, and then find the knob or wheel which also carries this number. Tuning is carried out by turning this knob or wheel. Once again, you are looking for a clear picture on the screen and silence from the loudspeaker. On this type of receiver, the picture is usually 'fine-tuned' automatically when you put the cover back on the tuning panel, so don't leave it off. If you do, the receiver's circuits that keep it in tune can't operate, and you will find that the tuning alters, so that you have to keep re-tuning. Figure 1.6 illustrates some of the faults that can be caused by mis-tuning.

## **The Atari attractions**

Once you have achieved a tuned signal from your Atari, the business of mastering the Atari attractions can start. It's important to note that nothing that you can do by pressing keys on the keyboard can possibly damage the Atari – the worst you can do is to lose a



Fig. 1.6. Picture defects caused by faulty tuning.

program that was stored in the memory. You can, however, damage the Atari by spilling coffee all over it, dropping it, or connecting it up to other circuits while the power is switched on. Opening the cartridge cover will not cause damage, because the computer is automatically switched off when you do this. You will, however, lose any program that was stored in the computer when you open this hatch.

It's time now to look at the keyboard, because the keyboard is the way that you pass instructions to the Atari. If we ignore the keys at the left- and right-hand sides, most of the Atari keys look like typewriter keys. The arrangement of letters and numbers is the same as that of a typewriter and if you've ever used a typewriter, particularly an electric typewriter, then you should be able to find your way round the keyboard of the Atari pretty quickly.

There's one very noticeable difference, though. When you use a typewriter, pressing a letter key gives you a small letter (called *lower-case*), and pressing a letter key along with the SHIFT key produces a capital letter (called *upper-case*). On the Atari, you will get upper-case letters whether you have the SHIFT key pressed or not. This is because the instructions that you issue to the machine have to be typed in upper-case letters, so it's set up to give these letters unless you want to use lower-case. To change over to lower-case, you must press the CAPS/LOWER key, which is above the SHIFT key on the right-hand side of the keyboard. After pressing this key, you will get lower-case letters normally, and capitals when you press the SHIFT

key as well, just like a typewriter. To return to normal computer-style capitals, press SHIFT and the CAPS/LOWER keys at the same time.

As well as the ordinary typewriter keys, there are a number of special keys which are not found on any typewriter. Among these are the keys which are marked ESC (Escape) and CTRL (Control). These are used in special ways that will be explained later. The ESC key is always used by pressing it, releasing it, and then pressing another key. The CTRL key is always used by being held down while another key is being pressed. Another important key is the BREAK key, at the top right-hand side of the keyboard. This is a 'panic button' which when pressed will return the control of the Atari to you if it appears to have 'locked up' and refuses to obey instructions. Pressing this key will not cause you to lose a program that is stored in the memory of the computer. The most important of these special keys, however, as far as we are concerned at the moment, is the key that is marked RETURN. This is in the position of the 'carriage return' key of an electric typewriter, but its action is not the same in all respects. Pressing the RETURN key is a signal to the computer that you have completed typing an instruction and that you now want the computer to obey it.

If you are accustomed to using an electric typewriter, you will have to change some of your habits as far as this key is concerned. During the use of a typewriter, you would press the 'carriage return' key each time you wanted to select a new line, with typing starting at the left-hand side of the new line. The RETURN key of the computer does rather more than this. If the material that you are typing into the Atari takes more than one line on the screen, the machine will automatically select the next screen line for you. The RETURN key must not be used for this purpose. The RETURN key is used only when you want the machine to carry out a command or store an instruction, not simply when you want to use a new line. It will always provide a new line for you, however, and select a position at the left-hand side. The position where a letter or other character will appear when you press a key is indicated by a flashing block on the screen. This flashing block is called the 'cursor', and it acts as a sort of signpost for you, as we'll see later.

## **Store it away**

You can get a lot of enjoyment from a computer system that consists

only of the machine and a TV receiver. Each time that you switch the machine off, however, all the program and other information that has been stored in the memory of the computer will be lost. Since it might take several hours to enter a program into the machine by typing instructions on the keyboard, this waste just has to be avoided. We avoid the loss of programs by recording them on tape.

The computer has circuits which will convert the instructions of a program into musical tones, which can then be recorded on an ordinary cassette recorder. When these notes are replayed, another set of circuits will convert the signals back into the form of a program. In this way, the use of a cassette recorder allows you to record your programs on tape and to replay them. Before you tackle the rest of this book, then, it's important to check now that you can record and replay programs.

Some computers can make use of ordinary cassette recorders, but such machines were never designed for use with programs, and troubles with recorders are very common when such machines are used. The Atari requires its own special recorder. This is purpose-built for program recording, and you cannot use an ordinary unmodified recorder in its place. The program recorder requires another power pack, which is also fitted with a three-pin plug for mains supply, and a small plug for engaging into the power socket of the recorder.

Start work by switching everything off. Now find the cassette lead of the Atari. This is attached to the program recorder at one end, and has a large plug at the other end. The plug fits into a socket at the right-hand side of the Atari - it's marked 'PERIPHERAL'. Be careful how you push this plug in. It should fit only one way round, so don't force it.

Once you have made this connection, the program recorder is ready for use. The next thing that you have to sort out is a supply of blank cassettes. There's nothing wrong with using reputable brands of C90 length cassettes (ordinary 'ferric' tape, not the hi-fi CrO<sub>2</sub> type), but you'll find that the short lengths of tape that are sold as C5, C10 or C15 in computer shops and in most branches of W. H. Smith's, Boots, and Currys are much more useful.

Put a fresh cassette into the machine, with the I or A side uppermost. The first part of the cassette consists of a 'leader' which is plain, not recording, tape. This has to be wound on before you can record. Reset the counter of the program recorder to zero, and then fast-wind the cassette to a count of 5.

Now before you can make a recording to test the system, you need



```

10 REM
20 REM
30 REM
40 REM

```

Fig. 1.7. A program for testing the cassette recording and replaying actions.

a program to record, and this involves some typing. This is easy if you have just switched the Atari on, but if you have been pressing keys at random, then it's a good idea to switch off again, then on. This gets the machine cleared, and all ready for you to start.

Type the number 10 (1 and then 0), and then the word REM. Check that this looks correct, and then press the RETURN key. The effect of this is to place the instruction line 10 REM into the memory of the Atari. Now type the rest of the lines, as illustrated in Fig. 1.7, remembering to press the RETURN key after you have completed typing each line. The numbers are called 'line numbers', and they are there for two reasons. One is to remind the computer that this is a program; the other is to guide it, because the computer will normally carry out instructions in the same order as the line numbers.

Check that your program looks on the screen like the printed version in Fig. 1.7, and make sure that the recorder is ready. Now type CSAVE. The C stands for cassette, and CSAVE is the instruction to the computer meaning that you want to save (record) a program on a cassette. Now press RETURN, and you'll hear the built-in loudspeaker of the Atari buzz twice at you. Now start the recorder by pressing its PLAY and RECORD keys. Press them firmly so that they lock in place. Nothing will happen until you press RETURN once more, and you will then see the reels of the cassette turning. If you turn up the volume control of the TV receiver, you will hear the sounds that are being recorded. After a rather long time, the cursor of the Atari will reappear on the screen with a READY message. This lets you know that the program has been recorded, and you can press the STOP key of the recorder. The motor will have been stopped automatically, but it's not good for the recorder to leave the PLAY key depressed for long periods when the motor is not moving. That's all.

Now comes the crunch. You have to be sure that the recording was O.K. Wind back the tape again. Type NEW and press RETURN. This should have wiped your program from the memory. Now type LIST and press RETURN. Nothing should appear - LIST means put a list of the program instructions on the screen, and there shouldn't be any!

You can now load the instructions in from the tape. Type

CLOAD and press RETURN. The loudspeaker will honk at you again, once only. You can rewind the tape, if you forgot to do so earlier. Now press the PLAY key of the recorder and press the RETURN key of the computer again. The motor will start, and the program will be entered into the memory of the Atari again. Once more, the word READY will appear to tell you that the loading process has finished, and the motor of the program recorder will stop. Type LIST now, then press the RETURN key. You should see your program appear on the screen.

Once you can reliably save programs on tape, and re-load them, you can confidently start computing. When you have spent an hour or more typing a program on to the keyboard, it's good to know that a few minutes' more work will save your effort on tape so that you won't have to type it again. It's a remarkably foolproof system compared to the capers that other computer owners have to endure, and you should find no problems. If you cannot load or save programs, ask your Atari dealer to sort it all out for you - there's nothing that you can do for yourself. If you find that some programs do not record satisfactorily, but others do, then there's advice for you in Appendix A. For the moment, however, you should encounter no problems while you stick to simple programming.

One final point. If you leave your Atari switched on for a long time without typing anything on the keyboard, you will see the picture on the screen change colour at intervals. This is intentional. It reminds you that the machine is switched on, and it prevents possible marking of the TV screen due to displaying one unchanged colour for a long time. It's just another example of thoughtful Atari design.

## Chapter Two

# Words On The Screen

Chapter 1 will have broken you in to the idea that the Atari, like practically all computers, takes its orders from you when you type them on the keyboard. You will also have found that an order is obeyed when the RETURN key is pressed. You will by now have used the command NEW which clears out a program from the memory; and LIST which prints your program instructions on to the screen. You will also have found that the CLEAR key at the top right-hand side of the keyboard has the effect of wiping the screen clear when it is pressed at the same time as the SHIFT key.

Now there are two ways in which you can use a computer. One way is called *direct mode*. 'Direct mode' means that you type a command, press RETURN, and the command is carried out at once. This can be useful, but the more important way of using a computer is in what is called *program mode*. In 'program mode' the computer is issued with a set of instructions, with a guide to the order in which they are to be carried out. A set of instructions like this is called a *program*.

The difference is important, because the instructions of a program can be repeated as many times as you like with very little effort on your part. A direct command, by contrast, will be repeated only if you type the whole command again, and then press RETURN.

Let's take a look at the difference. If you want the computer to carry out the direct command to add two numbers, 2.6 and 4.4, then you have to type:

PRINT 2.6 + 4.4 (and then press RETURN)

You have to start with PRINT because a computer is a dumb machine, and it obeys only a few set instructions. Unless you use the word PRINT, the computer has no way of telling that what you want is to see the answer on the screen. It doesn't recognise instructions like 'TELL ME' or 'WHAT IS', only a few words that

we call its 'reserved words' or 'instruction words'. PRINT is one of these words.

When you press RETURN after typing PRINT 2.6 + 4.4, the screen shows the answer, 7, under the command, and the word READY appears under this answer. The READY is a 'prompt', a reminder that the computer is ready for another command. Once this command has been carried out, however, it's finished.

A program does not work in the same way. A program is typed in, but the instructions of the program are not carried out when you press RETURN. Instead, the instructions are stored in the memory, ready to be carried out as and when you want. The computer needs some way of recognising the difference between your commands and your program instructions. On computers that use the 'language' called BASIC (Beginners' All-purpose Symbolic Instruction Code), this is done by starting each program instruction with a number which is called a 'line number'. This must be a positive whole number, the type of number that is called a 'positive integer'. This is why you can't expect the computer to understand an instruction like  $5.6 + 3 =$  ; it takes the 5 as being a line number, and the rest doesn't make sense.

Let's start programming, then, with the arithmetic actions of add, subtract, multiply and divide. Computers aren't used all that much for calculation, but it's useful to be able to carry out calculations now and again. Figure 2.1 shows a four-line program which will print some arithmetic results.

Take a close look at this, because there's a lot to get used to in these four lines. To start with, the line numbers are 10, 20, 30, 40 rather than 1, 2, 3, 4. This is to allow space for second thoughts. If you decide that you want to have another instruction between line 10 and line 20, then you can type the line number 15, or 11 or 12 or any other whole number between 10 and 20, and follow it with your new instruction. Even though you have entered this line out of order, the computer will automatically place it in order between lines 10 and 20. If you number your lines 1, 2, 3 then there's no room for these second thoughts.

The next thing to notice is how the number zero is slashed across

```
10 PRINT 2.6+5.5
20 PRINT 3.2-1.4
30 PRINT 2.4*4.6
40 PRINT 7.3/2.1
```

*Fig. 2.1.* A four-line arithmetic program.

as it appears on the screen and also in the printed program. This is to distinguish it from the letter O. The computer simply won't accept the  $\emptyset$  in place of O, nor the O in place of  $\emptyset$ , and the slashing makes this difference more obvious to you so that you are less likely to make mistakes. Some magazines, unfortunately, reprint computer programs with the slash-marks removed, so that it's very easy to make mistakes. There is no slash-mark across the  $\emptyset$  on the Atari keyboard, because its position makes it less easy to confuse with the letter O.

Now to more important points. The star or asterisk symbol in line 3 $\emptyset$  is the symbol that the Atari uses as a multiply sign. Once again, we can't use the  $\times$  that you might normally use for writing multiplication because this is a letter. There's no divide sign on the keyboard, so the Atari, like all other small computers, uses the backslash (/) sign in its place. This backslash sign is on the ? key, so don't confuse it with the other slash symbol which is on the + key. The Atari keyboard is arranged so that the four arithmetic symbols, +, -, \* and / can be typed without pressing the SHIFT key.

So far, so good. The program is entered by typing it, just as you see it. You don't need to leave any space between the line number and the P of PRINT, because the Atari will put one in for you when it displays the program on the screen. The space that shows on the screen when you type LIST and press RETURN does not get stored in the memory, so we save memory by missing this space out. You will have to press the RETURN key when you have completed each instruction line, before you type the next line number. You should end up with the program looking as it does in the illustration when you LIST it.

When you have entered the program by typing it, it's stored in the memory of the computer in the form of a set of code numbers. You already know how to check that the program is in the memory, by using the LIST command. What you need to know now is how to make the machine carry out the instructions of the program. Type RUN, then press the RETURN key, and you will see the instructions carried out.

When you follow the instruction word PRINT with a piece of arithmetic like 2.4\*4.6, then what is printed is the result of working out that piece of arithmetic. The program doesn't print 2.4\*4.6, just the result of the action 2.4\*4.6.

Now this is useful, but it's not always handy to get a set of answers on the screen, especially if you have forgotten what the questions were. The Atari allows you a way of printing anything that you like

## 16 Get More From The Atari

```
10 ? "2+2= ";2+2
20 ? "2.5*3.6= ";2.5*3.6
30 ? "9.6-2.7= ";9.6-2.7
40 ? "24.2/4.7= ";24.2/4.7
```

Fig. 2.2. Using quote marks. In this and other examples, the ? has been used in place of the PRINT instruction word.

on the screen, exactly as you type it, by the use of what is called a *string*. In addition, you can cut down the wear and tear of your typing finger(s) by typing ? in place of the word PRINT!

Figure 2.2 illustrates these principles. In each line, some of the typing is enclosed between quotes (inverted commas) and some is not. Enter this short program and run it. Can you see how very differently the computer has treated the instructions? Whatever was enclosed between quotes has been printed exactly as you typed it. Whatever was not between quotes is worked out, so that the first line, for example, gives the unsurprising result:

2+2=4

Now there's nothing automatic about this. If you type a new line:

```
15 PRINT "2+2= ";5*1.5
```

then you'll get the daft reply, when you RUN this, of:

2+2=7.5

The computer does as it's told and that's what you told it to do. One of the hardest things to appreciate about computers is that they do exactly what you instruct them, no more, no less. A computer is a tool, like an electric drill. As for computers taking over the world, well, electric drills didn't, though most households have one and use it. Note also that the spaces in the program of Fig. 2.2 between the = and the " are useful - just see what happens if you miss them out!

With all of this accumulated wisdom behind us, we can now start to look at some other printing actions. PRINT or its abbreviation ?, as far as the Atari is concerned, always means 'print on to the TV screen'. For activating a paper printer ('hard copy', it's called) there's a separate instruction LPRINT (and LIST"P:" for program listings). It's not an indication of Welsh design - the L once meant 'line' in the days when printers for computers were huge pieces of

```
10 ? "THIS IS"
20 ? "THE REMARKABLE"
30 ? "ATARI COMPUTER"
```

Fig. 2.3. Using the PRINT instruction (in its ? form) to place words on the screen.

machinery that printed a whole line at a time. You need not use these instructions unless you have a printer connected and switched on.

Now try the program in Fig. 2.3. You can try typing the lines in any order that you like, to establish the point that they will be in line number order when you list the program. When you RUN the program, the words appear in twos, with two words on each line. This is because the instruction PRINT doesn't just mean 'print on the screen'. It also means 'take a new line, and start at the left-hand side!'

Now this isn't always convenient, and we can change the action by using punctuation marks that we call *print modifiers*. Start this time by acquiring a new habit. Type NEW and then press the RETURN key. This clears the old program out. If you don't do this, there's a chance that you will find lines of old programs getting in the way of new ones. Each time you type a line, you delete any line that had the same line number in an older program, but if there is a line number that you don't use in the new program it will remain stored.

```
10 ? "THIS IS ";
20 ? "THE REMARKABLE ";
30 ? "ATARI"
```

Fig. 2.4. The effect of semicolons.

Now try the program in Fig. 2.4. There's a very important difference between Fig. 2.4 and Fig. 2.3, as you'll see when you RUN it. The effect of a semicolon following the last quote in a line is to prevent the next piece of printing starting on a new line at the left-hand side. When you RUN this program, all of the words appear in one line. It would have been a lot easier to do this by just having one line of program that read:

```
10 PRINT "THIS IS THE REMARKABLE ATARI"
```

but there are times when you have to use the semicolon to force two different print items on to the same line. We'll look at examples of that sort of thing later.

## Rows and columns

Neat printing is a matter of arranging your words and numbers into rows and columns, so we'll take a closer look at this particular art now. To start with, we know already that the instruction PRINT or

## 18 *Get More From The Atari*

```
10 ? CHR$(125):? "THIS IS THE ATARI"  
20 ? :?  
30 ? "READY TO OBEY YOU"
```

*Fig. 2.5.* Clearing the screen with CHR\$(125), and using multistatement lines.

its abbreviated form ? will cause a new line to be selected, so the action of Fig. 2.5 should not come as too much of a surprise. Line 10 contains two novelty items, though, one in the form of two instructions in one line. The instructions are separated by a colon (:), and you can, if you like, have several instructions following one line number in this way, taking several screen lines. So long as the number of characters in the 'line' does not exceed 114 (and at 40 characters per screen line that's almost three screen lines), you can put instructions together in this way. The Atari will remind you of the limit by sounding a warning bleep after 107 characters and at each character beyond this number. In a 'multistatement' line, the Atari will deal with the different instructions in a left-to-right order.

The other point about line 10 is that the instruction ?CHR\$(125) causes the screen to clear. This is the same action as you get when you press the SHIFT key and the CLEAR key at the same time. It doesn't affect what is stored in the memory, simply what you see on the screen. We'll look at another way of programming this instruction shortly. Moving on, line 20 causes the lines to be spaced apart. The two ? instructions, with nothing to be printed, each cause a blank line to be taken. There are other ways of doing this, as we'll see, but as a simple way of creating a space, it's very handy.

Figure 2.6 deals with columns. Line 10 is a PRINT instruction that acts on the numbers 1,2,3 and 4. When these appear on the screen, though, they appear spaced out just as if the screen had been divided into four columns. The mark which causes this effect is the comma, and the action is completely automatic. As line 20 shows, you can't get five columns. Anything that you try to get into a fifth column will actually appear on the first column of the next line down. It will also be displaced in from the left-hand side. The action works for words as well as for numbers, as line 30 illustrates. When words are being printed in this way, though, you have to remember

```
10 ? 1,2,3,4  
20 ? 1,2,3,4,5  
30 ? "ONE","TWO","THREE","FOUR"  
40 ? "THIS IS TOO LONG!","TWO","THREE","  
FOUR"
```

*Fig. 2.6.* How the comma causes words to be placed in columns.



that the commas must be placed outside the quotes. Any commas that are placed inside the quotes will be printed just as they are and won't cause any spacing effect. You will also find that if you attempt to put into columns something that is too large to fit, the long phrases will spill over to the next column, and the next item to be printed will be at the start of the next column along. Line 40 illustrates this – the first phrase spills over from column 1 into column 2, and the word TWO is printed starting at column 3. Once again, the word which 'spills over' is 'indented' – i.e. printed several spaces in from the left-hand side.

Commas are useful when we want a simple way of creating four columns. A much more flexible method of placing words along a line exists, however. It uses the tabulation feature of the Atari. This uses the TAB key which is located on the left-hand side of the keyboard. This acts like the TAB key of a typewriter, and its action is to shift the cursor along to one of a set of fixed places on the screen. Imagine your screen divided into 40 column positions, evenly spaced across. Of these, only columns 2 to 38 are normally available. When you switch your Atari on, the TAB positions are fixed at the left margin (which is the second column of the screen), at position 7,15,23, and so on every eight columns. Pressing the TAB key will advance the cursor to the position of the next TAB stop on the right, wherever the cursor happens to be.

The use of these TAB stops can be placed into a program by making use of the ESC key. This key is used to convert into program form an action which otherwise would be immediate. The way the ESC key is used is to press it, and release it, and then press the key which you want to use along with ESC. Figure 2.7 is a program which I have had to have type-set rather than use a print-out. The reason is that the effect of using the ESC and TAB keys does not show up on my printer. Where you see the > mark in a line, this is the result of pressing the ESC key followed by the TAB key. It appears on the screen as a solid arrowhead. When you RUN this program, you will see the effect of the standard TAB settings.

```

10 ?"START"
20 ?" ►FIRST TAB"
30 ?" ►►SECOND TAB"
40 ?" ►►►THIRD TAB"

```

*Note:* the ► mark is produced on the screen by pressing the TAB key.

*Fig. 2.7.* Using the tabulation key of the Atari. This program has been type-set, because it cannot be reproduced by my printer.

You can clear or set these TAB stops for yourself. Press the TAB key so as to move the cursor to its next TAB position, then press CTRL TAB. This means that the CTRL key and the TAB key are to be pressed at the same time. The effect of this is to clear this TAB, so that the cursor will not stop at this place again. To place a TAB stop at another position, move the cursor to where you want it and then press SHIFT TAB (SHIFT and TAB together). To move the cursor you can use either the spacebar (which allows only left-to-right movement) or press the CTRL key along with one of the four keys which have arrows on them (the keys next to RETURN and CAPS/LOWER). As you might expect, you can carry out this rearrangement of the TAB positions within a program also. The actions are the same, but you have to press ESC before you press the CTRL TAB or SHIFT TAB key combinations. The use of 'tabulation', as this is called, can make the appearance of printing on the screen much smarter.

Meantime, there's another very important print modifier to look at. The POSITION instruction allows us to place the cursor, invisibly, at any place we like on the screen. By 'invisibly', I mean that you won't see the cursor appear at its new position until a PRINT or ? instruction is carried out.

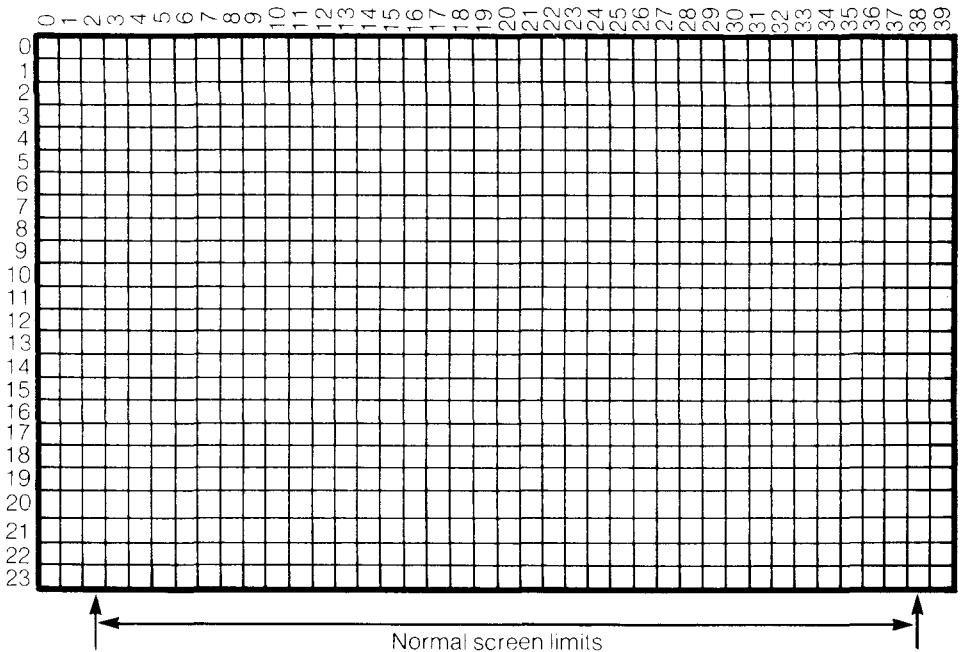


Fig. 2.8. The numbers that are used with the POSITION instruction.

For the purpose of using POSITION, we imagine the screen divided into a grid of 40 columns across and 24 rows down. These are numbered, counting 0 as the number for the left-hand column of the screen and for the top row. The column at the right-hand side is numbered 39, and the bottom row is numbered 23, as Fig. 2.8 shows. We can specify the position of the cursor by two numbers, the column number and the row number. For example, if we use the numbers 10,12, this will mean column number 10 and row number 12. The numbers are always given in this column, then row, order. What you have to remember, however, is that column numbers 0, 1, and 39 are not normally available. This can be overcome if you want to use all of the screen by the command:

```
POKE 82,0:POKE83,39
```

followed by pressing RETURN. This allows you to use the whole width of the screen.

Figure 2.9 shows how POSITION is used. The screen is cleared in

```
10 ? CHR$(125):POSITION 17,2:? "TITLE"
20 ? :?
30 POSITION 2,5:? "LOOKS A LOT SMARTER"
```

Fig. 2.9. Tabulation in a program by means of the POSITION instruction.

line 10. The next part of line 10 is POSITION 17,2. This selects the column number 17, and row number 2 (the third row, remember, because counting starts with 0). Following this is the ?"TITLE" instruction which will print the word TITLE starting at the desired

1. Count number of characters in the title, including spaces.
2. Divide by two, ignoring any remainder.
3. Subtract this result from 18, if you are using the normal screen pattern. If you have extended the screen width to 40 characters, then subtract from 20.
4. Use the result as the column number in the POSITION instruction, or as the number of spaces placed before the title. Alternatively, you can set the TAB key to this number of spaces.

#### Example

Title: ATARI MAGIC!.....12 characters.

$12/2=6$ , and  $18 - 6 = 12$ . Therefore, use POSITION 12,2 to centre on line 2.

Fig. 2.10. The formula for centring a title.

position. This achieves centring of the word TITLE, and Fig. 2.10 shows how the column number has to be calculated to achieve this. Finally, in Fig. 2.9, line 30 prints starting at column 2. This will be the left-hand side of the normal screen, but will appear set in if you have shifted the margins by using the POKE instructions.

When you use POSITION you don't have to print in the order of left to right or top to bottom either, because POSITION allows you complete freedom to print wherever you want. If your choice of POSITION places a new word over an old one, then the new letters will simply replace the old ones.

```

10 ? "]"
20 POSITION 5,20:?"FIRST ITEM"
30 POSITION 5,10:?"SECOND ITEM"
40 POSITION 5,1:?"THIRD ITEM"

```

*Fig. 2.11.* How POSITION is used to place words anywhere on the screen.

Finally, then, try the program of Fig. 2.11 as an illustration of the use of POSITION to place words anywhere on the screen. The screen is cleared in line 10, this time using a different-looking method. What has been done here is to press the ? mark (for PRINT), then the quote mark, and then ESC and SHIFT CLEAR in that order. Remember that this means pressing ESC, releasing it, and then pressing SHIFT and CLEAR together. The result on the screen looks like a bent arrow, but on my printer the curly bracket appears. Wherever you see this sequence of question mark, quote mark and curly bracket (then another quote mark) in this book, it's an indication of the ESC SHIFT CLEAR keys being used. The effect in a program is to clear the screen, as you might expect.

Line 20 then prints FIRST ITEM near the bottom of the screen. The words in lines 30 are printed above this first phrase, showing that you are not bound by the normal left-to-right or top-to-bottom order when you make use of POSITION. This is an exceptionally useful feature. Its equivalent on other computers is the PRINTAT or PRINT@ instruction.

## Chapter Three

# A Bit Of Variation

So far, our computing has been confined to printing numbers and words on the screen. That's one of the main aims of computing, but we have to look now at some of the actions that go on before anything is printed. One of these is called *assignment*. Take a look at the program in Fig. 3.1. Type it in, run it, and contrast what you see on the screen with what appears in the program. The first line that is printed is line 20. What appears on the screen is:

```
2 TIMES 25 IS 50
```

but the numbers 25 and 50 don't appear in line 20! This is because of the way we have used the letter X as a kind of code for the number 25. The official name for this type of code is a *variable name*.

```
5 ? ";"  
10 X=25  
20 ? "2 TIMES ";X;" IS ";2*X  
30 X=5  
40 ? "X IS NOW ";X  
50 ? "AND 2 TIMES ";X;" IS ";2*X
```

*Fig. 3.1.* Assignment in action. The letter X has been used in place of number

Line 10 assigns the variable name X, giving it the value of 25. 'Assigns' means that wherever we use X, not enclosed by quotes, the computer will operate with the number 25. Since X is a single character and 25 has two digits, that's a saving of space. It would have been an even greater saving if we had assigned X differently, perhaps as X=2174.3256, for example. Line 20 then proves that X is taken to be 25, because wherever X appears, not between quotes, 25 is printed, and the 'expression' 2\*X is printed as 50. We're not stuck with X as representing 25 for ever, though. Line 30 assigns X as being 5, and lines 40 and 50 prove that this change has been made.

That's why we call X a 'variable' – we can vary whatever it is we want it to represent. Until we do change it, though, X stays assigned.

Even after you have run the program of Fig. 3.1, providing you haven't added new lines or deleted any part of it, you can type ?X, and pressing RETURN will show the value of X on the screen.

This very useful way to handle numbers in code form can use a 'name' which must start with a letter. You can add to that as many more letters or digits as you like. For example, you could use D, MYNUMBER or R2D2 as your codes for numbers. You could also type the line:

```
1Ø 4D=5.67
```

but though the line would be accepted by the computer, you would get an error message if you typed ?4D or tried to use this value. Just to make it even more useful, you can use similar 'names' to represent words and phrases also. The difference is that you have to add a dollar sign (\$) to the variable name. If N is a variable name for a number, then N\$ (pronounced 'en-string' or 'en-dollar') is a variable name for a word or phrase. The computer treats these two, N and N\$, as being entirely separate and different.

### Tying it up

Figure 3.2 illustrates 'string variables', meaning the use of variable names for words and phrases. This is not quite so straightforward as the use of number variables. Lines 1Ø and 2Ø carry out the assignment operations, and line 3Ø shows how these variable names can be used. Notice that you can mix a variable name, which doesn't

```
1Ø DIM N$(5), AT$(5):? "":N$="NAME"  
2Ø AT$="ATARI"  
3Ø ? AT$;" IS THE ";N$;" TO REMEMBER"
```

Fig. 3.2. Using string variables. These are distinguished by the dollar sign. They must also be 'dimensioned'.

need quotes around it, with ordinary text, which must be surrounded by quotes. The important difference is illustrated in line 1Ø, however. The instruction word DIM means DIMENSION, and what it does is to prepare the computer's memory for a string variable. The reason is that the computer stores strings in its memory in a way that is completely different from the way it stores numbers. Any number value, whether large or small, will be stored in the same number of units of memory. We call these units of memory *bytes*, and a number is stored in 7 bytes. A string of letters, however, will

need one byte for each letter, so that the computer cannot prepare space for a string until the length of the string is known. The Atari gets round this problem by requiring you to declare in advance how many letters will be needed for each string variable name that you use. If you type, for example:

```
10 DIM NAMES(25)
```

then you can use up to 25 letters in anything you assign to NAMES\$. You aren't forced to use 25 letters, but you must not exceed 25. If you do, the excessive letters will simply be lost. If you try to use a string variable name which has not been dimensioned at all, you will get an error message - ERROR 9.

Now before you go wild on this use of variable names, a word of warning. There's nothing to stop you from using variable names of as many characters as you like. Nothing, except the fact that it uses up precious memory, though that's less of a worry if you are using the 48K (49,152 bytes) Atari 800. The computer will not be fooled by names that look alike. Take a look at Fig. 3.3. When you run this

```
10 DIM NURSE$(7),NUT$(7),NUMB$(5),NUDE$(
8)
20 NURSE$="LINDSAY"
30 NUT$="PEANUTS"
40 NUMB$="GOOFY"
50 NUDE$="STARKERS"
60 ? NURSE$
70 ? NUT$
80 ? NUMB$
90 ? NUDE$
```

*Fig. 3.3.* Illustrating long variable names. The Atari regards all of these as separate. Many computers would treat all of these variables as being NU\$.

one, lines 60 to 90 all produce different words. The Atari computer is not confused because all of these variable names have started with the letters NU. This is something you can't do on a lot of other computers!

## Strings and things

Because the name of a string variable is marked by the use of the \$ sign, a variable like A\$ is not confused with a number variable like A. We can, in fact, use both on the same program knowing that the computer at least will not be confused. Figure 3.4 illustrates that the

```

10 A=2:B=3
20 DIM A$(2),B$(2):A$="2":B$="3"
30 ? "}"
40 ? A;" TIMES ";B;" IS ";A*B
45 ? A$;" TIMES ";B$;" IS ?"
50 ? " BUT A*$*B$ IS IMPOSSIBLE - THE ATA
RI":? "WILL NOT ACCEPT SUCH A LINE."

```

Fig. 3.4. String and number variables might look alike when they are printed, but they are different!

difference is a bit more than skin deep, though. Lines 10 and 20 assign number variables A and B, and string variables A\$ and B\$. The essential dimensioning of A\$ and B\$ is also carried out in line 20. When these variables are printed in lines 40 and 45, you can't tell the difference between A and A\$ or between B and B\$. The difference would appear if you had tried to enter the line:

```
45 A$;" TIMES ";B$;" IS ";A*$B$
```

The computer will not accept such a line. It can multiply two number variables, because numbers can be multiplied, but it can't multiply string variables. The reason is simple. A string variable can be anything. We have assigned A\$ as '2', but we could just as easily have assigned it as '2 TRUMPET DRIVE'. You can multiply 2 by 3, but you can't multiply 2 TRUMPET DRIVE by 3 TROMBONE AVENUE. The computer therefore refuses to carry out multiplication, division, addition, subtraction or any other arithmetic operation on strings. Attempting to do a forbidden operation like this causes an error message to appear whenever you press RETURN. The line simply doesn't get entered into the memory. Later on, we'll see that there are operations that we can carry out on strings that we can't carry out on numbers, and attempts to do these operations on numbers will also cause an error message. The difference is an important one and you have to be aware of it.

Now while we are on the subject of strings and string variables, there's another string action which is very useful. Figure 3.5 illustrates this action of joining strings, which is often called *concatenation*. Concatenation is a very useful way of obtaining very long strings which can't be typed because of the limit of 114 characters that you can type after each line number. The Atari, in fact, allows you to use strings which contain as many characters as you like. The only limitations are the dimensions you have allowed, and the amount of memory that is available. Take a close look at Fig. 3.5. This defines strings A\$ and B\$ in lines 20 and 30, but we have dimensioned another longer string, X\$. In line 40, we assign X\$



```

10 DIM X$(20),A$(10),B$(10)
20 A$="SMITH"
30 B$="JONES"
40 X$=A$
50 X$(6)=B$
60 ? X$

```

*Fig. 3.5.* Concatenating or joining strings. The method that the Atari uses for programming this action is unusual.

to A\$, meaning that X\$ is now SMITH. Line 50 then adds B\$ to the end of A\$! This is done because the code X\$(6) means 'X\$ from position 6 onwards'. The number 6 is the letter count. Since there are only 5 letters in SMITH, position 6 is the blank space after the H of SMITH. By instructing: X\$(6)=B\$, we copy B\$ into the space that starts at position 6 in X\$. Line 60 then prints this concatenated string, showing you that SMITH and JONES have been united.

```

10 DIM A$(3),B$(3),AT$(5),N$(20)
20 A$="***":B$="###":AT$="ATARI"
30 ? "}"
40 N$=A$:N$(4)=B$:N$(7)=AT$:N$(12)=B$:N$(15)=A$
50 ? N$

```

*Fig. 3.6.* Using concatenation to make a frame for a title.

Figure 3.6 shows this technique being used in a different way. This time, we are going to print the name ATARI with asterisks and hashmarks (#) on each side. Line 10 dimensions the strings – note that we need only one DIM in this line, with commas used to separate the items. The strings are joined up in line 40, and line 50 prints the result.

```

10 ? "}:DIM NM$(50)
20 ? "WHAT IS YOUR NAME"
30 INPUT NM$
40 ? "}:? :?
50 ? NM$;" - THIS IS YOUR LIFE!!"

```

*Fig. 3.7.* Using the INPUT instruction. The name that you type is put into the phrase in line 50. Note the 'clear-screen' instruction which is typed by using '?', then CTRL, SHIFT CLEAR, and another quote.

## Putting it in

So far, everything that has been printed on the screen by a program has had to be placed in the program before it is run. We don't have to be stuck with restrictions like this, however, because the computer allows us another way of putting information, number or name, into

a program while it is running. A step of this type is called an INPUT and the BASIC instruction word that is used to cause this to happen is also INPUT.

Figure 3.7 illustrates this with a program that prints your name. Now I don't know your name, so I can't put it into the program beforehand. What happens when you run this is that the words:

WHAT IS YOUR NAME

are printed on the screen. On the line below this you will see a question mark. The computer is now waiting for you to type something, and then press RETURN. Until the RETURN key is pressed, the program will hang up at line 30, waiting for you. If you're honest, you will type your own name and then press RETURN. You don't have to put quotes around your name, simply type it in the form that you want to see printed. When you press RETURN, your name is assigned to the variable NM\$. The program can then continue, so that line 40 clears the screen and spaces down by two lines. Line 50 then prints the famous phrase with your name at the start.

You could, of course, have answered MICKEY MOUSE or DONALD DUCK or anything else that you pleased. The computer has no way of knowing that either of these is not your true name. The string variable NM\$ must, as usual, be dimensioned. If the name that you enter is longer than you have dimensioned for, then the extra is simply ignored. You're all right if your name is SMITH, but if you're called WILLOUGHBY-FORTESCUE-JONES then you will have to be generous with your dimensioning. The example uses DIMNM\$(50), which is pretty safe!

We aren't confined to using string variables along with INPUT. Figure 3.8 illustrates an INPUT step which uses a number variable N. The same procedure is used, but of course we don't need to dimension. When the program hangs up with the question mark appearing, you can type a number and then press the RETURN key. The action of pressing RETURN will assign your number to N, and allow the program to continue. Line 40 then proves that the program is dealing with the number that you entered. When you use

```
10 ? "ENTER A NUMBER"
20 INPUT N
30 ?
40 ? ."TWICE ";N;" IS ";2*N
```

*Fig. 3.8.* An INPUT to a number variable. The quantity that you type must be a number.

a number variable in an INPUT step, then what you have typed when you press RETURN must be a number. If you attempt to enter a string, the computer will refuse to accept it. It will stop running its program, with an ERROR 8 message showing. If your INPUT step uses a string variable then anything that you type will be accepted when you press RETURN, provided that you have dimensioned enough space.

The way in which INPUT can be placed in programs can be used to make it look as if the computer is paying some attention to what you type. Figure 3.9 shows an example – but with a small

```

10 ? "}:":DIM NM$(30)
20 ? "TYPE YOUR NAME, PLEASE ";;INPUT NM
$
30 ?
40 ? "VERY PLEASD TO MEET YOU, ";NM$

```

Fig. 3.9. Using INPUT to make it appear as if the computer knows you!

improvement in the way the question is asked and answered. In this example, a PRINT line is used to make the request, and it ends in a semicolon. This is then followed by a colon and the INPUT NMS\$ part of the program. The effect of the semicolon is to prevent the question mark of the INPUT from appearing in the next line. In this way, the question mark appears in its more natural position at the end of the question.

The use of INPUT isn't confined to a single name or number. We can use INPUT with two or more variables, and we can mix variable types in one INPUT line. Figure 3.10, for example, shows two variables being used after one INPUT. One of the variables is a string variable NMS\$, the other is the number variable N. Now when the computer comes to line 20, it will print the message and then wait for you to enter both of these quantities, a name and then a number. This has to be done in the correct way. The correct way, as far as strings, or mixtures of strings and numbers, are concerned is to enter each separately. In this example, you have to type the name, and

```

10 ? "}:":DIM NM$(20)
20 ? "NAME AND NUMBER, PLEASE ";;INPUT N
M$,N
30 ?
40 ? "THE NAME IS ";NM$
50 ? "THE NUMBER IS ";N

```

Fig. 3.10. Putting in two variables in one INPUT step. When the variables are of different types, as here, they must be put in separately.

### 30 *Get More From The Atari*

then press RETURN. A second question mark then appears to remind you that your work is not yet finished. You can then type the number and press RETURN again. The string can consist of anything, including commas. If, for example, you typed JONES, 2241716 as the name, then this is what would be stored as NM\$. You could then enter another number in response to the request for N. Other computers operate differently, so you will have to be careful if you are trying to type into your Atari a program that was designed for another type of computer.

```
10 ? "}"  
20 ? "TYPE FOUR NUMBERS, PLEASE"  
30 INPUT A, B, C, D  
40 ? "THE SUM OF THESE IS "; A+B+C+D
```

*Fig. 3.11.* An INPUT step which calls for four numbers. These can be entered in one operation.

Figure 3.11 shows a set of number entries. When only number variables are used in an INPUT, you have a choice of methods that you can use when you enter the numbers. You can enter the numbers separately, as we did in Fig. 3.10. The other method consists of entering all the numbers in one go, using commas to separate them, and pressing RETURN only when all of the numbers have been typed. You could, for example, type:

1,2,3,4

and then press RETURN to enter all four of these numbers.

### **Reading the data**

There's yet another way of getting data into a program while it is running. This one involves reading items from a list, and it uses two instruction words READ and DATA. The word READ causes the program to select an item from the list. The list is marked by starting each line of the list with the word DATA. The items of the list can be separated by commas. Each time an item is read from such a list, a 'pointer' is altered so that the next time an item is needed, it will be the next item on the list.

We'll look at this in more detail in Chapter 5, but for the moment we can introduce ourselves to the READ ... DATA instruction. Figure 3.12 uses the instruction in a very simple way. Line 20 reads the first item on the list and assigns it to the variable NM\$. This is

```

10 ? "3":DIM NM$(20)
20 READ NM$
30 ? NM$;
40 ? " IS VALUED AT ";
50 READ N
60 ? N;
70 ? " POUNDS"
100 DATA GOLD WATCH,700

```

Fig. 3.12. Using the READ and DATA words to place information into a program.

printed in line 30, with the semicolon keeping printing in the same line so that the phrase in line 40 follows it. The semicolon at the end of line 40 once more keeps the printing in the same line, and line 50 reads the number which is the second item in the list. This is assigned to the variable name N (we could just as easily have used NM\$) and printed in line 60. Once again, a semicolon prevents a fresh line from being taken, so that the final word of line 70 is printed following the number.

The READ...DATA instructions really come into their own when you have a long list of items that are read by repeating a READ step. We're not quite ready for that yet, so having introduced the idea, we'll leave it for now. As before, though, we have to match the data items with the variable names that we use for them. We can read a number item and assign it to a string variable name, but we can't read a string item and assign it to a number variable name. Note that a string item in a DATA line does not need to have quotemarks around it.

## Number antics

The amount of computing that we have done so far should have persuaded you that computers aren't just about numbers. For some applications, though, the ability to handle numbers is very important. If you want to use your computer to solve scientific or engineering problems, for example, then its ability to handle numbers will be very much more important than if you bought it for games, for accounts or for word processing. It's time, then, to take a very brief look at the number abilities of the Atari. It is a brief look because we simply don't have space to explain what all the mathematical operations do. In general, if you understand what a mathematical term like *sin* or *tan* or *exp* means, then you will have no problems about using these mathematical functions in your

programs. If you don't know what these terms mean, then you can simply ignore the parts of this section that mention them.

The simplest and most fundamental number action is counting. Counting involves the ideas of incrementing if you are counting up and decrementing if you are counting down. Incrementing a number means adding 1 to it, decrementing means subtracting 1 from it. These actions are programmed in a rather confusing looking way in BASIC, as Fig. 3.13 shows. Line 20 sets the value of variable X as 5.

```

10 ? "}"
20 X=5
30 ? "VALUE OF X IS ";X
40 X=X+1: ?
50 ? "NOW THAT WE'VE USED X=X+1": ?
60 ? "VALUE IS NOW ";X

```

Fig. 3.13. Incrementing, using the equals sign to mean 'becomes'.

This is printed in line 30, but then line 40 'increments X'. This is done using the odd-looking instruction:  $X = X + 1$ , meaning that the new value that is assigned to X is 1 more than its previous value. The rest of the program proves that this action of incrementing the value of X has been carried out.

The use of the = sign to mean 'becomes' is something that you have to get accustomed to. When the same variable name is used on each side of the equality sign, this is the use that we are making of it. We could equally well have a line:

$$X = X - 1$$

and this would have the effect of making the new value of X one less than the old value. X has been decremented this time. We could also use  $X = 2 * X$  to produce a new value of X equal to double the old value, or  $X = X / 3$  to produce a new value of X equal to the old value divided by three. Figure 3.14 shows another assignment of this type, in which both a multiplication and an addition are used to change the value of X.

```

10 ? "}"
20 X=5: ? "X IS ";X
30 ?
40 X=2*X+4
50 ? "NOW IT'S ";X

```

Fig. 3.14. A more elaborate reassignment, using an 'expression'.

```

10 ? "}:X=2.5:NP=2.3025851
20 ? "X SQUARED IS ";X^2
30 ?
40 ? "ITS SQUARE ROOT IS ";SQR(X)
50 ?
60 ? "ITS NATURAL LOG. IS ";LOG(X)
70 ? "ITS ORDINARY LOG. IS ";LOG(X)/NP

```

Fig. 3.15. Some number functions. In line 70, CLOG(X) can be used in place of LOG(X)/NP, and NP is not then needed.

## Number functions

Figure 3.15 illustrates some number functions. A number function in this sense is an instruction which operates on a number to produce another number. Line 10 picks the value of 2.5 for X. Line 20 then prints the value of X squared, meaning X multiplied by X. This is programmed by typing the character which the Atari gets by pressing SHIFT along with the \* key. It looks, as the printout shows, like an inverted 'V'. In line 40, to get the square root of the number that has been assigned to X, we use the instruction word SQR. An alternative is  $X^{.5}$ , but SQR(X) is easier to type and remember. For other roots, like the cube root, you can use expressions like  $X^{(1/3)}$  and so on. LOG(X) produces the natural logarithm of X, and line 70 shows how you can get the value of the ordinary (base 10) logarithm.

Now when you run this one, you will see that the square of 2.5 is printed as 6.2499999, and any calculator will illustrate that the value should be 6.25. One of the problems of small computers is precision of numbers. You probably know that the fraction  $1/3$  cannot be expressed exactly as a decimal. How near we can get to its true value depends on the number of decimal places we are prepared to print, so that 0.33 is closer than 0.3, and 0.333 is closer still. The computer converts most of the numbers it works with into the form of a fraction and a multiplier. The fraction is not a decimal fraction but a special form called a binary fraction, and this conversion is seldom

---

To round up a number, add 0.5, then take the integer part. For example, the number 23.614 becomes 24.14 when 0.5 is added, and 24 when the INT is taken. The number 23.414 becomes 23.914 when 0.5 is added, and 23 when the INT is taken. When the number is in the form of a variable X, then use:  $X = \text{INT}(X+.5)$

---

Fig. 3.16. How to round up a number which is in variable form.

exact. The conversion is particularly awkward for numbers like 1, 10, 100 and also .1, .01, .001; all the powers of ten, in fact. You will find, then, that some number results are slightly out, and you will need to round them up. Figure 3.16 shows how this rounding up operation can be carried out in a program, so that you don't find yourself printing awkward quantities like 1.9999999 instead of 2.

---

ABS(X)	Converts negative sign to positive.
ATN(X)	Gives the angle whose tangent has value X.
CLOG(X)	Gives the common logarithm of X.
COS(X)	Gives the cosine of angle X.
EXP(X)	Gives the value of <i>e</i> to the power X.
INT(X)	Gives the whole number part of X.
LOG(X)	Gives the natural logarithm of X.
RND(X)	Gives a random number between 0 and 1. X is not used, but a number or variable must be present.
SGN(X)	Gives the sign of X. The result is +1 if X is positive, -1 if X is negative, 0 if X is zero.
SIN(X)	Gives the sine of angle X.
SQR(X)	Gives the square root of X.

---

*Fig. 3.17.* Atari number functions, with brief notes. Don't worry if you don't know what some of these do. If you don't know, you probably don't need them!

Figure 3.17 illustrates the various number functions that can be used, with a brief explanation of what each one does. Some of these actions will be of interest only if you are interested in programming for scientific, technical or statistical purposes. Others, however, are useful in unexpected places, such as in graphics programs. In addition, you should know that the Atari can accept numbers in two forms. One is the familiar form in which we usually write numbers, but with no commas. You can, for example, type numbers like 13456734 or .28462547. The other form is 'scientific form', in which a number is typed as a value that lies between 1 and 10, and a power of ten. A power of ten is a number whose value is that number of tens multiplied together. Ten to the power 2, for example, is  $10*10=100$ , and ten to the power of four is  $10*10*10*10=10000$ . The number 123456 can be written as  $1.23456*10^5$ , and this is written for the computer in the form 1.23456E5. Your Atari will convert very large or very small numbers automatically into this form. If you enter a number in scientific form that can be put into normal form on the screen, the Atari will carry out this change also. The short program



in Fig. 3.18 shows these actions being carried out. In line 20, type the number:

```
12345678987654321
```

When you see it listed or printed, it will appear as it has been printed in the illustration. The computer automatically makes the conversion when the number of figures exceeds nine. In line 30, the amount that was actually typed was 1E2, but the Atari has converted this to its more normal form of 100.

```
10 ? "}"
20 ? 1.23456789E+16
30 ? 100
```

*Fig. 3.18.* Numbers in 'standard' or 'scientific' form. You can use either way of typing a number. This is the only exception to the rule that a letter cannot be entered as a number variable.

## Chapter Four

# Repetitions And Decisions

### Loops

One of the activities for which any computer is particularly well suited is repeating a set of instructions. The Atari is no exception, and we'll start with the simplest of these 'repeater' actions, GOTO. GOTO means exactly what you would expect it to mean - go to another line number. Normally a program is carried out by executing the instructions in ascending order of line number. In plain language that means starting at the lowest numbered line, working through the lines in order and ending at the highest numbered line. Using GOTO can break this arrangement, so that a line or a set of lines will be carried out in the 'wrong' order, or carried out over and over again.

```
10 ? "}"  
20 ? "ATARI ATARI ATARI ATARI ATARI"  
30 GOTO 20  
40 REM PRESS BREAK TO STOP
```

*Fig. 4.1.* A very simple loop. You can stop this by pressing the BREAK key.

Figure 4.1 shows an example of a very simple repetition or 'loop', as we call it. Line 10 clears the screen, and line 20 contains a simple PRINT instruction. When line 20 has been carried out, the program moves on to line 30, which instructs it to go back to line 20 again. This is a never-ending loop, and it will cause the screen to fill with the word ATARI until you press the BREAK key to 'break the loop'. Any loop that appears to be running forever can normally be stopped by pressing the BREAK key, though if this does not work, you will have to press the SYSTEM RESET key.

Now try a loop in which there is slightly more noticeable activity. Figure 4.2 shows a loop in which a different number is printed out each time the computer goes through the actions of the loop. We call

```

10 ? "}" : N=10
20 ? N
30 N=N-1
40 GOTO 20
50 REM PRESS BREAK TO STOP

```

Fig. 4.2. A loop which carries out a count-down action very rapidly. You will also have to use BREAK to stop this one.

this 'each pass through the loop'. Line 10 sets the starting value of the variable N at 10. This is printed in line 20, and then line 30 decrements the value of N. Line 40 forms the loop, so that the program will cause a very rapid count-down to appear on the screen. Once again, you'll have to use the BREAK key to stop it.

Now an uncontrolled loop like this is not exactly good to have, and GOTO is a method of creating loops that we prefer not to use! We don't always have an alternative, but there is one – the FOR ... NEXT loop. As the name suggests, this makes use of two new instruction words, FOR and NEXT. The instructions that are repeated are the instructions that are placed between FOR and NEXT. Figure 4.3 illustrates a very simple example of the

```

10 ? "}"
20 FOR N=1 TO 10
30 ? "ATARI GENIUS AT WORK!"
40 NEXT N

```

Fig. 4.3. Using the FOR...NEXT loop for a counted number of repetitions.

FOR...NEXT loop in action. The line which contains FOR must also include a number variable which is used for counting, and numbers which control the start of the count and its end. In the example, N is the counter variable, and its limit numbers are 1 and 10. The NEXT N is in line 40, and so anything between lines 20 and 40 will be repeated.

As it happens, what lies between these lines is simply the PRINT instruction, and the effect of the program will be to print ATARI GENIUS AT WORK ten times. At the first pass through the loop, the value of N is set to 1, and the phrase is printed. When the NEXT N instruction is encountered, the computer increments the value of N, from 1 to 2 in this case. It then checks to see if this value exceeds

```

10 ? "}"
20 FOR N=10 TO 1 STEP -1
30 ? N;" SECONDS AND COUNTING"
40 FOR J=1 TO 500:NEXT J
50 ? "}" : NEXT N
60 ? "BLASTOFF"

```

Fig. 4.4. A program that uses nested loops, with one loop inside another.

the limit of 10 that has been set. If it doesn't, then line 30 is repeated, and this will continue until the value of N exceeds 10 – we'll look at that point later. The effect in this example is to cause ten repetitions.

You don't have to confine this action to single loops either. Figure 4.4 shows an example of what we call 'nested loops', meaning that one loop is contained completely inside another one. When loops are nested in this way, we can describe the loops as inner and outer. The outer loop starts in line 20, using variable N which goes from 10 to 1 in value. Line 30 is part of this outer loop, printing the value that the counter variable N has reached. Line 40, however, is another loop. This must make use of a different variable name, and it must start and finish again before the end of the outer loop. We have used variable J, and we have put nothing between the FOR part and the NEXT part to be carried out. All that this loop does, then, is to waste time, making sure that there is some measurable time between the actions in the main loop. The last action of the main loop is clearing the screen in line 50. The overall effect, then, is to show a count-down on the screen, slowly enough for you to see the changes, and wiping the screen clear each time. In this example we have used NEXT J in line 40 and NEXT N in line 50. This is essential, and if you omit the letter after the NEXT, or type the wrong letter, the Atari will refuse to accept the instruction. When you use NEXT J and NEXT N like this, you must be absolutely sure that you have put the correct variable names following each NEXT. If you don't, the computer will remind you!

There's another novelty in this program, though. The loop in Fig. 4.3 counted upwards, adding 1 to the value of the counter each time. We don't always want this, and we can add the instruction word STEP to the end of the FOR line to alter this change of variable value. We could, for example, use a line like:

```
FOR N=1 TO 9 STEP 2
```

which would cause the values of N to change in the sequence 1,3,5,7,9. When we don't type STEP, the loop will always use increments of 1. Figure 4.4 uses an outer loop which has a step of -1, so that the count is downwards. N starts with a value of 10, and is decremented on each pass through the loop.

Every now and again, when we are using loops, we find that we need to use the value of N after the loop has finished. It's important to know what this will be, however, and Fig. 4.5 brings it home. This contains two loops, one counting up, the other counting down. At the end of each loop, the value of the counter variable is printed. This

```

10 ? "}"
20 FOR N=1 TO 5
30 ? N
40 NEXT N
50 ? "N IS NOW ";N
60 FOR N=5 TO 0 STEP -1
70 ? N
80 NEXT N
90 ? "N IS NOW ";N

```

Fig. 4.5. Finding the value of the loop variable after a loop action is completed.

reveals that the value of N is 6 in line 50, after completing the FOR N = 1 TO 5 loop, and is -1 in line 90 after completing the FOR N= 5 TO 0 STEP -1 loop. If you later want to make use of the value of N, or whatever variable name you have selected to use, you will have to remember that it will have changed by one more step at the end of the loop.

One of the most valuable features of the FOR...NEXT loop, however, is the way in which it can be used with number variables instead of just numbers. Figure 4.6 illustrates this in a simple way.

```

10 ? "}"
20 A=2:B=5:C=10
30 FOR N=A TO B STEP B/C
40 ? N
50 NEXT N

```

Fig. 4.6. A loop instruction that is formed with number variables.

The letters A, B and C are assigned as numbers in the usual way in line 20, but they are then used in a FOR...NEXT loop in line 30. The limits are set by A and B, and the step is obtained from an expression, B/C. The rule is that if you have anything that represents a number or can be worked out to give a number, then you can use it in a loop like this.

## Loops and decisions

It's time to see loops being used rather than just demonstrated. A simple application is in totalling numbers. The action that we want is that we enter numbers and the computer keeps a running total, adding each number to the total of the numbers so far. From what we have done up to now, it's easy to see how this could be done if we wanted to use numbers in fixed quantities, like ten numbers in a set. The program of Fig. 4.7 does just this.

```

10 ? "}:":TOTAL=0
20 ? "PROGRAM FOR TALLING NUMBERS"
30 ? "ENTER EACH NUMBER AS REQUESTED."
40 ? "THE PROGRAM WILL GIVE THE TOTAL"
50 FOR N=1 TO 10
60 ? "NUMBER ";N;" PLEASE - ";
70 INPUT J:TOTAL=TOTAL+J
80 NEXT N
90 ? :? "TOTAL IS ";TOTAL

```

Fig. 4.7 A number-totalling program for ten numbers.

The program starts by setting a number variable called TOTAL to zero. This is the number variable that will be used to hold the total, and it has to start at zero. As it happens, the Atari arranges this automatically at the start of a program, but it's a good habit to ensure that everything that has to start with some value actually does so.

Lines 20 to 40 issue instructions, and the action starts in line 50. This is the start of a FOR...NEXT loop which will repeat the actions of lines 60 and 70 ten times. Line 60 reminds you of how many numbers you have entered by printing the value of N each time, and line 70 allows you to INPUT a number which is then assigned to variable name J. This is then added to the total in the second half of line 70, and the loop then repeats. At the end of the program, the variable TOTAL contains the value of the total, the sum of all the numbers that have been entered.

It's all good stuff, but how many times would you want to have just ten numbers? It would be a lot more convenient if we could just stop the action by signalling to the computer in some way, perhaps by entering a value like 0 or 999. A value like this is called a *terminator*, something that is obviously not one of the normal entries that we would use, but just a signal. For a number-totalling program, a terminator of 0 is very convenient, because if it gets added to the total it won't make any difference.

```

10 ? "}:":? "ANOTHER TOTAL PROGRAM!"
20 ? :? "THE PROGRAM WILL TOTAL NUMBERS
FOR YOU"
30 ? "UNTIL YOU ENTER A ZERO TO STOP IT.
"
40 TOTAL=0
50 ? "NUMBER, PLEASE- ";
60 INPUT N:TOTAL=TOTAL+N
70 ? "TOTAL SO FAR IS ";TOTAL
80 IF N<>0 THEN 60

```

Fig. 4.8. A number-totalling program which can't use FOR...NEXT.

Figure 4.8, therefore, shows an example of this type of program in action. We can't use a FOR ... NEXT loop, because we don't know in advance how many times we might want to go through the loop, so we have to go back to using GOTO. This time, however, we'll keep GOTO under closer control – the word won't even appear in the program! This time the instructions appear first, but we still have to make the total variable TOTAL equal to zero in line 40. Each time you type a number, then, in response to the request in line 50, the number that you type is added to the total in line 60, and line 70 prints the value of the total so far. Line 80 is the loop controller, and the key to the control is the instruction word IF. IF is used to make a test, and the test in line 80 is to see if the value of N is not equal to zero. The odd-looking sign that is made by combining the 'less-than' and the 'greater than' signs, <>, is used to mean 'not equal', so the line reads: 'if N is not equal to zero, then (go to) line 60'. We can put the GOTO in, or leave it out. Since it's just a few more letters to type, I've left it out.

The effect, then, is that if the number which you have typed in line 60 was not a zero, line 80 will send the program back to repeat line 60. This will continue until you do enter a zero. When this happens, the test in line 80 fails (N is zero), and the program stops. This kind of action is called a 'repeat...until' loop.

Now this allows you much more freedom than a FOR...NEXT loop, because you are not confined to a fixed number of repetitions. The key to it is the use of IF to make a decision – and that's what we need to look at more closely now.

---

Sign	Meaning
=	Exact equality.
>	Left-hand quantity greater than right-hand quantity.
<	Left-hand quantity less than right-hand quantity.
The signs can be combined as follows:	
<>	Quantities not equal.
>=	LHS greater than or equal to RHS.
<=	LHS less than or equal to RHS.

---

Fig. 4.9. The mathematical signs that are used for comparing numbers and number variables.

**Decisions, decisions**

We can make a number of types of comparisons between number variables or numbers, and these are listed in Fig. 4.9. The mathematical signs are used for convenience, and you have to remember which way round the 'greater than' and 'less than' signs have to be. It's important to note that the equals sign means 'identical to' when it is used in a test like this. If A is 3.9999999 and B is 4.0000000 then a test such as IF A = B will fail - A is not identical to B, even though it is close enough to be equal in our eyes.

```

10 ? "}:":DIM A$(3)
20 ? "PRESS Y OR N KEY"
30 ? "-THEN PRESS RETURN"
40 INPUT A$
50 IF A$="Y" THEN ? "THAT'S YES"
60 IF A$="N" THEN ? "THAT'S NO"

```

*Fig. 4.10.* Testing string variables, in this example to find whether a reply is Y or N.

Figure 4.10 shows another test - this time on string variables. The instructions are in lines 20 to 30; you are asked to type the Y or N key. Line 40 gets your answer; you have to type Y or N and then press RETURN. The key that you have pressed has its value assigned to A\$, so that A\$ should be Y or N. Lines 50 and 60 then analyse this result. If the key that you pressed was neither Y nor N, nothing is printed by the line 50 or 60.

The test in this example is for identity. Only if A\$ is absolutely identical to Y will the phrase 'THAT'S YES' be printed. If you typed a space ahead of Y, or a space following, or typed y in place of Y, then A\$ will not be identical, and the test fails. Failing means that A\$ is not identical to Y and everything that follows THEN in that line

```

10 ? "}:":DIM A$(1)
20 ? "TYPE Y OR N"
30 INPUT A$
40 IF A$="Y" THEN 100
50 IF A$="N" THEN 200
60 ? "YOUR ANSWER ";A$;" IS NOT Y OR N":
? "PLEASE TRY AGAIN":GOTO 30
70 END
100 ? "THAT WAS YES!"
110 END
200 ? "THAT WAS NO!"
210 END

```

*Fig. 4.11.* Testing string variables, with a mugtrap incorporated.



will be ignored. It's up to you to form these tests so that they behave in the way that you want!

We often find it better to test so that we can detect an incorrect reply as well. This is illustrated in Fig. 4.11. If A\$ is Y, then the first test in line 40 succeeds, and the program moves to line 100. This prints a message, and the program ends. If A\$ is N, then the first test in line 40 fails, but the next test in line 50 is carried out and, if A\$ is N, the program jumps to line 200, prints a different message, and ends. If both tests fail, though, the program will move from line 50 to line 60. Your answer was not exactly Y or N, so that you are asked to try again, and the GOTO30 at the end of line 60 causes the program to repeat from line 30. This line constitutes a *mugtrap*, a way of trapping mistakes. Very often when you have a choice of answers, you want to be sure that only certain replies are permitted. A mugtrap is a section of program that is intended to deal with an incorrect entry. A good mugtrap should show the user the error of his or her ways, and indicate what answer or answers might be more acceptable. This is very often important, because an incorrect entry in some types of program could cause the program to stop with an error message showing. For the skilled programmer (you, later), this is just a minor annoyance, but for the inexperienced user it can cause a minor panic. A good program doesn't allow any entries that would cause the program to stop. Mugtraps are our method of ensuring this.

Just to emphasise the sort of power that these simple instructions give you, Fig. 4.12 illustrates a very simple number-guessing game.

```

10 ? "":X=1+INT(RND(0)*10)
20 ? "GUESS THE NUMBER!"
30 ? :? "IF YOU GET NEAR, I'LL TELL YOU"
40 INPUT N
50 IF N=X THEN ? "SPOT ON!":END
60 IF ABS(N-X)<3 THEN ? "CLOSE- IT WAS "
;X:END
70 ? "TRY AGAIN -":GOTO 40

```

Fig. 4.12. A simple number-guessing game which uses number comparisons.

Line 10 clears the screen, and the  $X = 1 + \text{INT}(\text{RND}(0) * 10)$  step causes variable X to take a value that lies between 1 and 10. We can't predict what this value will be, because RND means 'select at random' – a fractional number is picked, somewhere in the range of just above zero to just less than 1. Multiplying this fraction by ten will give a number, picked at random by the computer, which must be somewhere between almost zero and 9.999999. By taking INT,

which rounds the number down, we get a number which must lie between 0 and 9. Adding 1 to this then ensures that the number is a whole number which lies between 1 and 10. RND picks numbers randomly enough for games purposes, but not quite randomly enough for serious statistical users. In lines 20 and 30, the instructions ask you to guess the size of the number, with the difference that you don't have to find it exactly. You enter your number at line 40, and the tests are made in lines 50 and 60. If the number that you picked is identical to the random number, then you get the SPOT ON message in line 50, and the program ends. The less obvious test is in line 60. The expression  $N-X$  is the difference between your guess,  $N$ , and the number  $X$ . If your guess is larger than the number, then  $N-X$  is a positive number. If your guess is less than  $X$ , then  $N-X$  is a negative number. The effect of ABS, however, is to make any number positive, so that if  $X$  were 5 and you guessed 6 or 4, then  $ABS(N-X)$  would come to 1. If you get a difference of 1 or 2 (less than 3), the message in line 60 is printed. If you don't get anywhere near, the program repeats because of its GOTO 40 in line 70. It's very simple, but quite effective. How about devising a scoring system?

### Single key reply

So far, we have been putting in  $Y$  or  $N$  replies with the use of INPUT, which means pressing the key and then pressing RETURN. This has the advantage of giving you time for second thoughts, because you can delete what you have typed and type a new letter before you press RETURN. For snappier replies, however, there is an alternative in the form of GET. GET is an instruction that is used to find a code number, and we can select where the computer is required to look for this number. Before GET can be used, however, we have to specify to the computer where it is to look for the code. This is done in line 10 by the OPEN instruction (see Fig. 4.13). At this stage, you will have to take it on trust that the #1,4,0, part is needed, but we can explain the "K:" - this means 'keyboard'. Because we have used #1 after OPEN, we can use this also with GET to ensure that what we get comes from the keyboard. By using GET#1,X, the computer carries out a check of the keyboard to find if a key is pressed. This checking action is very fast, but it will be repeated until a key is pressed, so that the computer waits for you. Line 40 then prints the value of  $X$ . This is a number, not a letter, and

its value is the number-code that the computer uses to represent the letter of the key you have pressed. By using the program of Fig. 4.13 you can see the effect of pressing different keys. If you add a line 60:

```
60 GOTO 20
```

you can make this program repeat until you press the BREAK key. In this way, you can find which keys will have an effect. Some keys will produce no visible character on the screen in line 40, but will nevertheless allow the program to jump out of its loop in line 30. Note, by the way, that one key certainly won't work in this way - the BREAK key!

These number codes need some explanation. The code system that is used is called ASCII, meaning American Standard Code for Information Interchange. Figure 4.14 shows the list of these codes as they apply to this program. The Atari makes use of other code numbers as well, and we'll look at these when we come to Graphics in Chapter 7.

```
10 ? "}:OPEN #1,4,0,"K:"
20 ? "PRESS ANY KEY"
30 GET #1,X
40 ? X
50 ? "THIS MEANS THE ";CHR*(X);" KEY."
```

Fig. 4.13. Using GET to find the code number for a key.

## Menus and subroutines

A choice of two items, such as in Fig. 4.11 isn't exactly a consumer's dream, not in the West anyway. We can extend the choice by a program routine that is called a *menu*. A menu is a list of choices, usually of program actions. By picking one of these choices, we can cause a section of the program to be run. One way of making the choice is by numbering the menu items, and typing the number of the one that you want to use. We could use a set of lines such as:

```
IF K = 1 THEN 10000
IF K = 2 THEN 20000
```






and so on. There is a much simpler method, however, which uses a new instruction ON N GOTO, where N is a number variable. You can use any number variable, of course, not just N.

Figure 4.15 shows a typical menu that uses this instruction. Lines

46 *Get More From The Atari*

CODE	CHARACTER	CODE	CHARACTER	CODE	CHARACTER
32	Space	48	0	64	@
33	!	49	1	65	A
34	"	50	2	66	B
35	#	51	3	67	C
36	\$	52	4	68	D
37	%	53	5	69	E
38	&	54	6	70	F
39	'	55	7	71	G
40	(	56	8	72	H
41	)	57	9	73	I
42	*	58	:	74	J
43	+	59	;	75	K
44	,	60	<	76	L
45	-	61	=	77	M
46	.	62	>	78	N
47	/	63	?	79	O

Fig. 4.14. The ASCII codes for characters.

CODE	CHARACTER	CODE	CHARACTER	CODE	CHARACTER
80	P	96		112	p
81	Q	97	a	113	q
82	R	98	b	114	r
83	S	99	c	115	s
84	T	100	d	116	t
85	U	101	e	117	u
86	V	102	f	118	v
87	W	103	g	119	w
88	X	104	h	120	x
89	Y	105	i	121	y
90	Z	106	j	122	z
91	[	107	k	123	
92	\	108	l	124	
93	]	109	m	125	
94	^	110	n	126	
95	_	111	o	127	

```

10 ? "}:OPEN #1,4,0,"K:"
20 POSITION 17,1:? "MENU"
30 POSITION 2,3
40 ? "1. ENTER NAMES."
50 ? "2. ENTER PHONE NUMBERS."
60 ? "3. LIST ALL NAMES."
70 ? "4. LIST LOCAL NUMBERS."
80 ? "5. END PROGRAM."
90 ?
100 ? "PLEASE SELECT BY NUMBER.":? "DO N
OT PRESS RETURN KEY."
110 GET #1,K
120 IF K<49 OR K>53 THEN ? "INCORRECT CH
OICE- 1 TO 5 ONLY":? "PLEASE TRY AGAIN."
:GOTO 110
130 ON K-48 GOTO 150,160,170,180,190
140 END
150 ? "NAMES HERE":END
160 ? "NUMBERS HERE":END
170 ? "NAMES LIST":END
180 ? "LOCAL NUMBERS":END
190 ? "ANNOUNCE END"

```

*Fig. 4.15.* A menu choice which uses the ON N GOTO instruction. The actual quantity that is used is not N but K-48.

10 to 90 are concerned with presenting the menu items on the screen, and line 100 then invites you to pick one item by typing its number. The GET action in line 110 keeps the program looking for a key until you make your choice, and then line 120 tests your choice with a mugtrap. The mugtrap is not as you might expect at first sight. We don't test for values less than 1 or greater than 5 because that's not what GET gets! What is got is an ASCII code number, remember, and the ASCII code for 1 is 49, with 53 used for '5'. These are the numbers that we have to test for in line 120.

The choice is then made in line 130, with the ON K-48 GOTO instruction. Now what happens here? If K equals 49, then K-48 is 1, and the first line number that follows GOTO is used. If K equals 50, then the second line number following GOTO is used, and so on. All that you have to do is to arrange the line numbers in the same order as your choices. You needn't have a list that looks neat. A line such as ONKGOTO250,216,484,714,1000 would be just as satisfactory so long as these numbers contained the start of routines that dealt with the menu choices. In this example, the line numbers simply lead to PRINT instructions so as to keep the example reasonably short. What you must remember, however, is that the ON N GOTO instruction is organised to use the numbers 1,2,3... and so on. You can't expect it to deal with the codes that are produced by GET

unless you operate on the codes. The operation of subtracting 48 does just what we want.

This type of menu selection is useful, but an even more useful method makes use of *subroutines*. A subroutine is a section of program which can be inserted anywhere that you like in a longer program. A subroutine is inserted by typing the instruction word GOSUB, followed by the line number in which the subroutine starts. When your program comes to this instruction, it will jump to the line number that follows GOSUB, just as if you had used GOTO. Unlike GOTO, however, GOSUB offers an automatic return. The word RETURN is used at the end of the subroutine lines, and it will cause the program to return to the point immediately following the GOSUB. Figure 4.16 illustrates this. When the program runs, line

```

10 ? "}"
20 ? "THE NORTH STAR IS CALLED";
30 GOSUB 1000
40 ?
50 ? "THE";:GOSUB 1000: ? "MISSILE USES T
HE SAME NAME."
60 END
1000 ? " POLARIS ";
1010 RETURN

```

Fig. 4.16. Using a subroutine—this is the key to more advanced programming.

20 prints a phrase, with the semicolon used to prevent a new line from being selected. The GOSUB 1000 in line 30 then causes the word POLARIS to be printed, but the RETURN in line 1010 will send the program back to line 40, the instruction that immediately follows the GOSUB 1000. This action will also occur even when the GOSUB is part of a multistatement line, as line 50 demonstrates. The GOSUB 1000 will cause the word POLARIS to be printed, but the return is to the PRINT instruction that follows GOSUB 1000 in line 50; it doesn't jump to line 60. This is, of course a rare example of a missile fired from a subroutine.

Now for something more serious. Figure 4.17 shows subroutines in use as part of an (imaginary) games program. Lines 10 to 80 offer a choice, and line 90 invites you to choose. The familiar GET and mugtrap actions follow, and then line 120 causes the choice to be carried out. This time, however, the program will return to whatever follows the choice. For example, if you pressed key 1, then the subroutine that starts at line 1000 is carried out, and the program returns to line 120 to check if you might also want subroutines 2000,

```

10 ? "}:OPEN #1,4,0,"K:"
20 POSITION 9,1:? "CHOOSE YOUR MONSTER"
30 ?
40 ? "1. FRANKENSTEIN'S CREATION."
50 ? "2. THE CURSED MUMMY."
60 ? "3. THE THING FROM THE LAGOON."
70 ? "4. KING KONG."
80 ? "5. THE GIANT SQUID."
90 ? :? "SELECT NUMBER, PLEASE. DO NOT U
SE RETURN."
100 GET #1,K
110 IF K<49 OR K>53 THEN ? "FAULTY SELEC
TION- 1 TO 5 ONLY":? "PLEASE TRY AGAIN":
GOTO 100
120 ON K-48 GOSUB 1000,2000,3000,4000,50
00
130 ? "THAT'S THE END"
140 END
1000 ? "ANOTHER FRIDAY JOB":RETURN
2000 ? "ASK YOUR DAD, THEN.":RETURN
3000 ? "IT'S PART OF THE LAGOON SHOW.":R
ETURN
4000 ? "SOUNDS LIKE AN ORANG UTANG.":RET
URN
5000 ? "CHANGE IT FOR TWO SFIFTIES":RETU
RN

```

*Fig. 4.17.* A menu choice that makes use of subroutines.

3000, 4000, or 5000. Since the value of K is unchanged, the program then goes to line 130 and ends. If line 1000 had altered the value of K, however, you could find that a second subroutine was selected following the first one. A subroutine is extremely useful in menu choices, but it's even more useful for pieces of program that will be used several times in a program. One such example is the GET#1,K type of routine, but we can have whole sections of a program which can be written as a subroutine. We sometimes find it an advantage to have subroutines used even when there is no repetition - but that's impinging on the material of Chapter 6!



## Chapter Five

# String Up Your Programs!

### String Functions

In Chapter 3, we took a fairly brief look at number functions. If numbers turn you on, that's fine, but *string functions* are in many ways more interesting. What makes them that way is that the really eye-catching and fascinating actions that the computer can carry out are so often done using string functions. What's a string function, then? As far as we are concerned, a string function is any action that we can carry out with strings. That definition doesn't exactly help you, I know, so let's look at an example. Figure 5.1 shows a program that prints ATARI as a title. What makes it more eye-catching is the fact that the word is printed with twelve hash-marks (#) on each side. The hash-marks are produced by a string function, one called 'string insertion' that few computers possess.

```
10 DIM A$(40),B$(12)
20 ? "}"
30 B$="#":B$(12)=B$:B$(2)=B$
40 A$=B$:A$(13)="ATARI":A$(18)=B$
50 POSITION 4,5: ? A$
```

Fig. 5.1. String insertion. This is an action which is almost unique to Atari.

Take a look at Fig. 5.1. It starts by dimensioning two strings. Of these, A\$ is dimensioned as long enough to fill one complete line on the screen – in fact it's longer than we need unless we are using the complete screen width of 40 characters. B\$ is dimensioned to twelve characters, and we will use it to hold the hash-marks. Line 20 clears the screen in the conventional way. Now the real novelty starts in line 30, because this shows how we can create a string of twelve identical characters without typing all twelve! The first part of line 30, B\$="#", causes a hash-mark to be put into the first space in B\$. The second part of line 30 copies this hash-mark into space twelve.

The interesting action is provided by the last part of the line, `B$(2)=B$`, which fills the rest of the string with hash-marks!

This is how it works. The command `B$(2)=B$` means that the string on the left should be identical to the string on the right. The string on the left has a starting position of place two in `B$`, which starts as an empty space. The process works one character at a time, and this is the reason for its effect. It will start by making the character at `B$(2)`, the second space, equal to the first character of `B$`, which is a hash-mark. By the time it has done this, though, `B$` is one hash-mark longer than it was. When it comes to make the third space of the left hand string equal to the second character of `B$`, it finds that this is now a hash-mark, not a blank space. The hash-mark then is placed in the third space. It then makes the fourth space of the left-hand string equal to the third space of the right-hand string – and that's now a hash-mark! The process continues until the string has reached its dimensioned length, and stops there. The result in this example is a string of hash-marks. Other types of computers use a command called `STRING$` to achieve this useful effect. Figure 5.2 shows another application of this useful technique, which is not described in the Atari manual. In this example, it creates a complete line of asterisks so that they can be used to underline a title.

```
10 ? " ":DIM A$(37)
20 A$="*":A$(37)=A$:A$(2)=A$
30 POSITION 16,3: ? "ATARI"
40 ? A$
```

*Fig. 5.2.* Using string insertion to create an underlining.

### How long is a piece of string?

String variables allow us to carry out a lot of operations that can't be done with number variables. One of these operations is finding out how many characters are contained in a string. Since a string can contain as many characters as we like, memory space permitting, a method of counting them is rather useful, and `LEN` is that method. `LEN` has to be followed by the name of the string variable, within brackets, and the result of using `LEN` is always a number so that we can print it or assign it to a number variable.

Figure 5.3 shows a simple example of `LEN` in use. Line 20 assigns a variable and line 30 tells you how many letters are in this variable.

```

10 ? "}" : DIM A$(9)
20 A$="ATARI"
30 ? "THERE ARE ";LEN(A$);" LETTERS IN "
;A$

```

Fig. 5.3. Introducing LEN, a member of the string function family.

This is hardly earth-shattering, but we can turn it to very good use, as Fig. 5.4 illustrates. This program uses LEN as part of a subroutine which will print a string called TITLE\$ centred on a line. This is an extremely useful subroutine to use in your own programs, because its use can save you a lot of tedious counting when you write your programs. The principle is to use LEN to find out how many characters are present in the string TITLE\$. This number is then divided by two, and subtracted from 20, using the formula that we saw first in Chapter 2. If the number of characters in the string is an odd number, then  $17 - \text{LEN}(\text{TITLE})/2$  will contain a .5, but this is completely ignored by POSITION in line 1010. Once the subroutine is in place, we can call it to centre anything that has the name TITLE\$. In line 20, TITLE\$ is assigned to the words ATARI GENIUS!, and this phrase is printed centred. In line 50, TITLE\$ is assigned to a string of thirteen asterisks, using the simple method of typing them. This also is printed centred by the subroutine.

```

10 ? "}" : DIM TITLE$(30)
20 TITLE$="ATARI GENIUS!"
30 Y=1:GOSUB 1000
40 ?
50 TITLE$="*****"
60 Y=2:GOSUB 1000
70 ? :?
80 END
1000 T=(17-LEN(TITLE$)/2)
1010 POSITION T,Y
1020 ? TITLE$
1030 RETURN

```

Fig. 5.4. Using LEN to print titles centred.

Notice, by the way, that if we want anything printed centred by this subroutine, we have to give it the variable name of TITLE\$. This action is called 'passing a variable' to the subroutine, and it's something that we have to keep a careful eye on when we use subroutines. You can't expect a subroutine that is written to print TITLE\$ centred to have any effect on a string called A\$. Another point that is worth noticing is that LEN gives the correct length for each string, despite the fact that TITLE\$ was dimensioned to 30

characters. LEN gives the number of characters that we place into the string, not the maximum number that we dimension.

### String cutter's delight

The next group of string operations that we're going to look at are called *slicing operations*. The result of slicing a string is another string, a piece copied from the longer string. String slicing is a way of finding what letters or other characters are present at different places in a string.

All of that might not sound terribly interesting, so take a look at Fig. 5.5. The string A\$ is assigned in line 20, and sliced in line 30.

```
10 ? " ":DIM A$(20)
20 A$="ATARAXIA"
30 ? A$(1,4);: ? "I"
```

Fig. 5.5. Using the Atari string slicing action.

What's printed in line 30 is the word ATARI. Now how did this happen? The instruction ?A\$(1,4) means, take a slice out of A\$, starting at position 1 and ending at position 4. We number the places in the string starting with 1 at the left-hand side. Counting consecutively towards the right, the first letter is A and the fourth is R, so A\$(1,4) is ATAR. The semicolon keeps the printing on the same line when we add the I in the next part of the line. That's how ATARI comes out of ATARAXIA!

For a more serious use of this instruction, take a look at Fig. 5.6. This has the effect of extracting your initials from your name, and it's done by using this type of slicing method along with the opposite process—placing letters into a string in the way that we introduced earlier. The program starts by getting two names, which are assigned to variables SN\$ and FN\$. In line 40, then, IN\$=FN\$(1,1) gets the first letter of FN\$, and assigns it to IN\$. You must use (1,1) for this

```
10 ? " ":DIM SN$(20),FN$(20),IN$(5)
20 POSITION 5,2: ? "YOUR SURNAME, PLEASE-
";: INPUT SN$
30 POSITION 5,4: ? "YOUR FIRST NAME, PLEA
SE- ";: INPUT FN$
40 IN$=FN$(1,1): IN$(2)=" ": IN$(3)=SN$(1,
1): IN$(4)="."
50 ? "ROUND HERE, YOU ARE "; IN$
```

Fig. 5.6. Extracting initials with string slicing.

if you use (1) only, you will make IN\$, starting at character 1, equal to the whole of FN\$. The next part of line 40 places a full-stop in the second place of IN\$. The third part then puts the first letter of SN\$ into the third place of IN\$, and then we put in another full-stop in the fourth place. The result is printed in line 50. If you have two players in a game, it's often useful to show the initials and score rather than printing the full name, but the full names can be held stored for use at various stages in the game.

String slicing isn't confined to copying a selected piece of the left-hand side of a string. We can take a copy of characters from any part of a string that we want to use. Figure 5.7 illustrates the use of the

```

10 ? "}:DIM A$(20),B$(6)
20 A$="ATARIMAGIC"
30 B$=A$(5,8)
40 ? A$(1,5); " ";A$(6); " IN ";B$; "ES"

```

Fig. 5.7. Building up words and phrases by slicing.

instructions to take the letters IMAG out of a word. Line 40 then shows how a phrase can be built up out of this and other pieces snipped from the word. Think, for example, how you could conceal a message by defining a string consisting of the alphabet, and selecting letters from it like this. Figure 5.8 illustrates another light-hearted use – making selections from a string that happens to be your name. There are more serious uses than this. You can, for example, extract the last four figures from a string of numbers like 010-242-7016. I said a string of numbers deliberately, because something like this has to be stored as a string variable rather than as a number. If you try to assign this to a number variable, you'll get a silly answer. Why? Because when you type  $N = 010-242-7016$  then the computer assumes that you want to subtract 242 from 10 and 7016 from that result. The value for N is -7248, which is not exactly what you have in mind! If you use  $N\$ = "010-242-7016"$ , then all is well.

The string slicing method that the Atari uses is a very powerful and simple one, but it is not used by many other computers – though

```

10 ? "}:DIM A$(20)
20 ? "TYPE YOUR SURNAME, PLEASE"
30 INPUT A$:L=LEN(A$)
40 FOR N=1 TO L
50 ? A$(1,N); "           ";A$(L-N+1)
60 NEXT N

```

Fig. 5.8. Forming patterns from words by slicing actions.

---

LEFT\$(A\$,X) means slice the first X characters from A\$.  
The Atari equivalent is A\$(1,X).

*Example:* LEFT\$(Q\$,5) becomes Q\$(1,5).

MID\$(A\$,X,Y) means slice Y characters from A\$, starting with character number X.

The Atari equivalent is A\$(X,X+Y-1).

*Example:* MID\$(P\$,2,5) becomes P\$(2,6).

RIGHT\$(A\$,X) means slice the last X characters from A\$.

The Atari equivalent is A\$(LEN(A\$)-X).

*Example:* RIGHT\$(R\$,3) becomes R\$(LEN(R\$)-3).

When the second number in an Atari slicing instruction is omitted, all the characters to the right of the starting character are included.

---

*Fig. 5.9.* The Atari equivalents of the slicing commands that are used on many other computers. This can be useful if you want to convert a program to run on your Atari.

the Sinclair Research machines use a similar method. Most other computers use a set of three instructions called LEFT\$, MID\$ and RIGHT\$ to achieve what the Atari does with the numbers following the string name. In case you want to convert a program that was written for another type of computer, Fig. 5.9 shows each of these instructions in action, and the Atari equivalent.

## Getting more value

It's time now to look at some other types of string functions. We've met VAL previously - it's used to convert a number that is in string form back into number form so that we can carry out arithmetic. There's an instruction that does the opposite conversion, STR\$. When we follow STR\$ by a number, number variable, or expression within brackets, we carry out a conversion to a string variable. We can then print this as a string, or assign it to a string variable name, or use string functions like slicing, LEN, and all the others. Figure 5.10 illustrates these processes. Lines 10 to 30 show that we can do arithmetic on N\$ if we use VAL with it. Line 50 converts the number variable V into string form. Now V has been assigned to the number 2.5 in line 20, and line 60 reveals that there are three characters in V\$. This is what we would expect, since there are two digits and a decimal point. Line 80 shows the strings being used in an addition,

```

10 ? "}:":DIM N$(5),V$(5)
20 N$="22.5":V=2.5
30 ? N$;" TIMES ";V;" IS ";V*VAL(N$)
40 ?
50 V$=STR$(V)
60 ? "THERE ARE ";LEN(V$);" CHARACTERS I
N ";V
70 ?
80 ? N$;" ADDED TO ";V$;" GIVES ";VAL(N$
)+VAL(V$)

```

*Fig. 5.10.* Using STRS and VAL for converting between string and number form.

using VAL to make a conversion back to number form before carrying out the addition operation.

If you hark back to Chapter 4 now, you'll remember that we introduced the idea of ASCII code. This is the number code that is used to represent each of the characters that we can print on the screen. We can find out the code for any letter by the function ASC, which is followed, within brackets, by a string character. The result of ASC is a number, the ASCII code number for that character. If you use ASC("ATARI"), then you'll get the code for the A only, because the action of ASC includes rejecting more than one character. Figure 5.11 shows this in action. String variable A\$ is assigned in line 20 and in line 30 a loop starts which will run through all the letters in A\$. The letters are picked out one by one, using A\$(N,N) to slice one letter from each place in the string, and the ASCII code for each letter is found with ASC. The space between quotes, along with the semicolons in line 40 makes sure that the codes are all printed on one line with a space between the numbers. It's a simple and neat way of finding what ASCII codes are needed to build up a word, something that we'll look at later.

ASC has an opposite function, CHR\$. What follows CHR\$, within brackets, has to be a code number, and the result is the character whose code number is given. The instruction PRINT CHR\$(65), for example, will cause the letter A to appear on the screen, because 65 is the ASCII code for the letter A. We can use this for concealing messages. Every now and again, it's useful to be able

```

10 ? "}:":DIM A$(5)
20 A$="ATARI"
30 FOR N=1 TO LEN(A$)
40 ? ASC(A$(N,N));" ";
50 NEXT N

```

*Fig. 5.11.* ASCII codes. Using ASC to find the ASCII code for letters.

```

10 ? "}:? :OPEN #1,4,0,"K:"
20 ? "WHAT'S THE WORD FOR COMPUTER?"
30 ?
40 ? "PRESS ANY KEY TO REVEAL"
50 GET #1,X
60 ?
70 FOR J=1 TO 5:READ N
80 ? CHR$(N);
90 NEXT J
100 END
110 DATA 65,84,65,82,73

```

*Fig. 5.12.* Using CHR\$ to obtain the character that corresponds to a code number.

to hide a message in a program so that it's not obvious to anyone who reads the listing. Using ASCII codes is not a particularly good way of hiding a message from a skilled programmer, but for non-skilled users it's good enough. Figure 5.12 illustrates this use. Line 50 contains a GET #1,X step to make the program wait for you. Remember that you have to prepare for this by having the OPEN step at some point in the program earlier than the GET. When you press a key, the loop that starts in line 70 prints 5 characters on the screen. Each of these is read as an ASCII code from a list, using a READ...DATA instruction in the loop. The PRINT CHR\$(N) in line 80 then converts the ASCII codes into characters and prints the characters, using a semicolon to keep the printing in a line. Try it! If you wanted to conceal the letters more thoroughly, you could use quantities like one quarter of each code number, or 5 times each code less 20, or anything else you like. These changed codes could be stored in the list, and the conversion back to ASCII codes made in the program. This will deter all but really persistent decoders!

### **Making comparisons**

We saw earlier in Fig. 4.12, how numbers can be compared. We can also compare strings, using the ASCII codes as the basis for comparison. Two letters are identical if they have identical ASCII codes, so it's not difficult to see what the identity sign, =, means when we apply it to strings. If two long strings are identical, then they must contain the same letters in the same order. It's not so easy to see how we use the > and < signs until we think of ASCII codes. The ASCII code for A is 65, and the code for B is 66. In this sense, A is 'less than' B, because it has a smaller ASCII code. If we want to



place letters into alphabetical order, then, we simply arrange them in order of ascending ASCII codes.

This process can be taken one stage further, though, to comparing complete words, character by character. Figure 5.13 illustrates this use of comparison using the = and > symbols. Line 20 assigns a nonsense word – it's just the first six letters on the top row of letter keys. Line 30 then asks you to type a word. The comparisons are then carried out in lines 40 and 50. If the word that you have typed, which is assigned to B\$ is identical to QWERTY, then the message in

```

10 ? "":DIM A$(20),B$(20),C$(20)
20 A$="QWERTY"
30 ? :? "TYPE A WORD ";;INPUT B$
40 IF B$=A$ THEN ? "SAME AS MINE":END
50 IF A$>B$ THEN C$=A$:A$=B$:B$=C$
60 ? "ORDER IS ";A$;" THEN ";B$
70 END

```

Fig. 5.13. Comparing words to decide on their alphabetical order.

line 40 is printed, and the program ends. If QWERTY would come later in an index than your word, then line 50 is carried out. If, for example, you typed POLYGON, then since P comes before Q in the alphabet and has an ASCII code that is lower than the code for Q, the word A\$, which is QWERTY, scores higher than B\$, which is POLYGON, and line 50 swaps them round. This is done by assigning a new string, C\$ to A\$ (so that C\$ = "QWERTY"), then assigning A\$ to B\$ (so A\$ = "POLYGON"), then B\$ to C\$ (so that B\$="QWERTY"). Line 60 will then print the words in the order A\$ and then B\$, which will be the correct alphabetical order. If the word that you typed comes later than QWERTY (for example, TAPE) then A\$ is not 'greater than' B\$, and the test in line 50 fails. No swap is made, and the order A\$, then B\$, is still the correct one. Note the important point, though, that words like QWERTZ and QWERTX will be put correctly into order – it's not just the first letter that counts.

## Lists and letters

The variable names that we have used so far are useful, but there's a limit to their usefulness. One important limit is the number of variable names that you can use. You might think that because you can use variable names of any length, then you could have any

number of variable names. This is not so, because the Atari reserves only a limited amount of its memory for variable names. This limit is 128 variable names. It's not likely, in fact, that you find this a limit, certainly not in the early days of computing, but it's something you have to keep tucked away in your memory for use later. Another point concerned with this is that once you have assigned a variable, the name of it is stored until you carry out the NEW instruction (or switch off). If you have a program that continually creates variable names, then even if these variables have no value (they have been deleted), the names are still stored, and will count to your limit of 128.

As it happens, there is a way in which we can use a single variable name to store as many values as we like. Figure 5.14 illustrates this.

```

10 ? " ":DIM A(10)
20 FOR N=1 TO 10
30 A(N)=10+INT(90*RND(0))
40 NEXT N
50 ?
60 ? "          MARKS LIST"
70 ? :FOR N=1 TO 10
80 ? "ITEM ";N;" RECEIVED ";A(N);" MARKS
"
90 NEXT N

```

Fig. 5.14. An array of subscripted number variables. It's simpler than the name suggests!

Lines 10 to 40 generate an (imaginary) set of examination marks. This is done simply to avoid the hard work of entering the real thing! The variable in line 30 is something new, though. It's called a *subscripted variable*, and the 'subscript' is the number that is represented by N. How often do you make a list with the items numbered 1,2,3... and so on? These numbers 1,2,3 are a form of subscript number, put there simply so that you can identify different items. Similarly, by using variable names A(1), A(2), A(3) and so on, we can identify different items that have the common variable name of A. A member of this group like A(2) has its name pronounced as 'A-of-two'. The whole group is referred to as the 'array A', and a member of the group, like the value of A(1), is called an 'element of the array A'.

The usefulness of this method is that it allows us to use one single variable name for the complete list, picking out items simply by their identity numbers. Since the number can be a number variable or an expression, this allows us to work with any item of the list. Figure 5.14 shows the list being constructed from the FOR...NEXT loop in

lines 30 to 60. Each item is obtained by finding a random number between 1 and 100, and is then assigned to A(N). Ten of these 'marks' are assigned in this way, and then lines 60 to 90 print the list. It makes for much neater programming than you would have to use if you needed a separate variable name for each number.

You can't just leap into this array business without some preparation, though. The preparation is in the second part of line 10, and it consists of the DIM instruction that we have also used with strings. This is not accidental, because the computer treats a string as an array of ASCII code numbers. Following the DIM, we have the name of the array and, in brackets, the number of elements. In this example, we want to use ten marks, so we dimension A as ten items, A(10). We can, incidentally, dimension more than one array at a time. If we type, for example, DIM A(10), B(12) then this has the effect of dimensioning the variable B as well as A, but to 12 elements rather than the ten of A.

Number arrays are very useful when you want to store a whole set of numbers, particularly because an array name counts as just one single variable name, no matter how many elements it contains. Numbers may play little part in the kind of computing you want to do, though, and it would be useful if you could create and use arrays of strings. Some computers allow string arrays to be created in exactly the same way as number arrays, but the Atari does not have these instructions. That doesn't mean that we can't use string arrays, only that we have to create them in a different way.

```

10 ? " ":DIM A$(200),NM$(20),X(10)
20 ? "PLEASE ENTER NAMES":J=1:A$=""
30 FOR N=1 TO 10
40 ? "NAME ";;INPUT NM$
50 A$(J)=NM$:X(N)=LEN(A$):J=1+X(N)
60 NEXT N
70 ? " ":X(0)=0:FOR N=0 TO 9
80 ? A$(X(N)+1,X(N+1)):NEXT N

```

Fig. 5.15. Using strings in a form of array. The names are packed into one long string, with a number array used to locate the end of each name.

Figure 5.15 shows how a string array can be created. The techniques are already familiar to you, but the results are not. We start by dimensioning three items. Of these, A\$ is a long string which is going to be our string array. NM\$ is a temporary string that we will use for entering an item into this array, and X is a number array which will be used to keep track of the items in the string A\$. What we are going to do is to type names, assign each one to NM\$, and

then tack it on to the end of the previous name that is stored in A\$. In this way, A\$ will eventually consist of all the names that we have typed, joined together. Now since we use string arrays to store strings that we will eventually want to separate again, we need some way of keeping track. This is the reason for the number array X. Each time we add a name to A\$, we assign an element of X equal to the length of A\$. In this way, the array X contains a set of figures which are the lengths of the string A\$ at each stage. We can use these X numbers to find the start and end position of any one of the strings.

Now let's look at the program in detail. The entry of names starts in line 30, but some preparation has to be made in line 20. This line prints brief instructions, and 'initialises the variables'. This impressive phrase simply means that we set the values of J and A\$ to what we want them to start with. J starts at 1, because the string A\$ will start at its first character space, and A\$ starts as a blank, which is programmed as A\$="", with no space between the quotes. The entry process starts with the loop in line 30. At line 40, the word NAME is printed, and the INPUT step allows you to type a name. This will be limited automatically to twenty letters, because of the dimensioning of NM\$ in line 10. Line 50 then makes A\$(J) equal to NM\$. This means, remember, that NM\$ is copied into A\$, starting at character position J, which means position 1 since J is 1 on the first pass through the loop. In the second part of line 50, the length of A\$ is found, and is allocated to X(1), because N=1. The third part of line 50 then alters the value of J to J+X(1), which is the number of the next blank space in A\$.

Think of an example. If you typed SINCLAIR, then this will now take up the first eight spaces in A\$. X(1) will have the value 8, and J will be set to 9, the next vacant space in A\$. In line 60, the NEXT N sends the program back to pick up another name. The process is the same as before, but because J is now 9, in the example, the new name will be copied in at position 9. If you had typed ATARI this time, A\$ would now be SINCLAIRATARI, X(2) would be 13, and J would be 14, ready for the next name. This continues until all ten names have been entered and added to the string A\$.

Now that the string array has been created, we need to be able to separate out the names again. This is simple, and line 80 does the trick. Line 70 clears the screen, assigns X(0) as zero, and starts a loop. Once again, the easiest way to see what is happening is to think of the example. In line 80, when the loop starts, N is 0. Since we have made X(N)=0 in line 70, then X(N)+1 is just 1. This is *not* the same

```

10 ? " ":DIM A$(200),NM$(20),X(10)
20 ? "PLEASE ENTER NAMES":J=1:A$=""
30 FOR N=1 TO 10
40 ? "NAME ";:INPUT NM$
50 A$(J)=NM$:X(N)=LEN(A$):J=J+X(N)
60 NEXT N
70 ? " ":DIM Q$(1),B$(20)
80 ? "NOW WE'LL FIND A NAME"
90 ? "PLEASE TYPE THE INITIAL LETTER":?
"OF THE NAME YOU WANT"
100 INPUT Q$
110 X(0)=0:FOR N=0 TO 9
120 B$=A$(X(N)+1,X(N+1))
130 IF Q$=B$(1,1) THEN ? B$
140 NEXT N
150 ? "NOW ANOTHER...":GOTO 100

```

Fig. 5.16. Creating a string array and then making use of it to find any name.

as  $X(N+1)$ , because  $N+1=1$ , and  $N(1)$  in our example is 8. The PRINT (or ?) instruction in line 80 will therefore give  $A$(1,8)$  which is the first eight characters of  $A\$$ , the word SINCLAIR in my example. When the NEXT N in line 80 is reached, the value of N changes to 1. Now  $X(N)$  is  $X(1)$ , and its value is 8,  $X(N)+1$  is therefore 9.  $X(N+1)$  is  $X(2)$ , and its value is 13. What is printed, then, is  $A$(9,13)$ , which is ATARI. The loop continues in this same way, printing out all the names that you typed in.

Figure 5.16 shows a useful variation on this principle. You only need to alter the lines from 70 onwards, so if you still have Fig. 5.15 stored in the computer, or available on tape, you can save yourself some typing. This time, the array is created, and you can select names that start with a specified letter. As usual, we have to carry out some dimensioning, and this is done in line 70. It could just as easily have been put into line 10, but I wanted to leave the first part of the program exactly as it was. You type an initial letter in line 100, and the dimensioning of  $Q\$$  ensures that only one letter is accepted. Lines 110 to 140 then extract each name from  $A\$$  and assign it, in turn, to  $B\$$ . The first letter of  $B\$$ , found by using  $B$(1,1)$  is compared with  $Q\$$  and, if they match,  $B\$$  is printed. When the loop is complete, each name which starts with the letter assigned to  $B\$$  will have been printed. If no names start with that letter, nothing is printed. The GOTO in line 150 allows you to make another choice from the same array. It's simple and neat. Suppose, though, that you wanted to print a message such as "NO SUCH NAME" if the program did not find a name starting with your selected letter? How would you do

**64** *Get More From The Atari*

that? I'll leave that to you, with some broad hints. Suppose you had a third part to line 130, which then read:

```
IF Q$=B$(1,1) THEN ? B$:F=1
```

and we had previously had a line:

```
15 F=0
```

You could then have a line:

```
145 IF F=0 THEN PRINT"NO SUCH NAME"
```

and this would take care of the problem. You would then have to be sure that F (called a 'flag' because it signals an event) was reassigned to 0 in line 150 before you selected another letter.

## Chapter Six

# Filing And Planning

Up to now, we have used variables in a program, but we have made no attempt to store these variables on tape. As it happens, when you save a program that has been run, using the CSAVE command, you will save all the values of variables along with the program. This means that when you replay a program, using CLOAD, you can type GOTO (then a line number) instead of RUN, and the program will start with the variable values that it had when it was recorded. When you type RUN, you will generally clear all the variable values. For some purposes, though, it's useful to be able to record variable values separate from a program. This way, you can use the same variable values in more than one program. If you have created a 'string array', for example, of all your friends' names and addresses, you could use this information in a lot of programs, from printing party invitations to keeping a track on birthdays. It's time, then, for us to take a look at how we save and load variable values.

```
10 ? " ": DIM A$(5), B$(50)
20 A$="ATARI": ? "PLEASE TYPE YOUR NAME"
30 INPUT B$: C=342.16
40 ? "PLEASE HAVE A DATA CASSETTE READY"
: ? "PRESS RECORD/PLAY KEYS ON RECORDER":
? "THEN PRESS RETURN"
50 OPEN #1, B, 0, "C: "
60 PRINT #1; A$; CHR$(155); B$; CHR$(155); C;
CHR$(155)
70 CLOSE #1
```

*Fig. 6.1.* The steps that are needed to save variable values on tape.

Figure 6.1 demonstrates saving variables. As always, we need dimensions, and line 10 takes care of this. Lines 20 and 30 then produce some values for us to save. I'll assume that you are using the cassette program recorder, rather than the disk drive. When you use cassettes, you have to be sure that there is a cassette in the program recorder, and that it is wound on to a blank position. You don't, for

example, want to record all over a copy of a program. This is the purpose of the message in line 40.

The real novelty starts in line 50. We have met OPEN before, when we made use of GET, and the principle is the same. OPEN means that you are instructing the computer to prepare for transferring data. In the case of a GET, the transfer was from the keyboard into the memory. In this example, the transfer is from the memory to the cassette recorder, so the bits that follow OPEN are rather different. The #1 is called the 'channel number' – the hash-mark # is used in the USA in the way that we use No. to mean 'number'. You can use numbers 1 to 5 freely for this 'channel number', but it's wise to avoid 0, 6, and 7 until you know a lot more about the Atari. These numbers are reserved for other uses, as we'll see in the next chapter. If you are using only one OPEN, it makes sense to use #1. If you want to use two, as when you are using GET as well as data recording, you can use #1 for the GET, and #2 for the recording. The important point is that you need to use different numbers.

Having 'opened a channel', then, the rest of the OPEN instruction is concerned with what you are going to do. By specifying 8 following #1, you are instructing the computer to send data out – using 4 would specify reading data in. The 0 which follows the 8 just reserves a space – some types of instruction need a figure here. The final part is another important one – "C:". This specifies that the data is being sent out to the cassette recorder, not to the printer, the screen, the disk drive or anywhere else. When the OPEN#1,8,0,"C:" instruction is encountered in your program, the built-in loudspeaker of the Atari will hoot twice, the usual warning that you need to press the PLAY and RECORD keys of the program recorder, and then press the RETURN key of the Atari.

The OPEN instruction is a preparation, though, and we still have to specify what we want to record. This is done in line 60, using PRINT#1. The PRINT#1 means that you are putting data out 'on channel 1', and the instruction has to be followed by a list of what you want to send out. You have to be rather careful here. The items must be separated by semicolons, and there must be a CHR\$(155) following each item. The CHR\$(155) is called the 'end of file' character and, if you forget it, you won't be able to read back your variables correctly. You don't need the CHR\$(155) if you take a new line and a new PRINT#1; for each item. Finally, in line 70, you need CLOSE#1. This doesn't just set the channels back to normal, it actually completes the recording process. If you omit this



instruction, the recording will not end correctly.

The result of these lines 50 to 70 is that you hear the warning hoots, you press PLAY and RECORD, and then press RETURN. The recorder operates, recording your data on tape, and then stops. It is at this point that recording is complete. In this example, you know when the recording is complete because the word READY appears on the screen. When this recording step is part of a longer program, you would need to have a PRINT step following the CLOSE#1 to remind you that you could stop the recorder and remove or rewind the cassette.

```

10 ? "":DIM X$(5),Y$(50)
20 ? "PLEASE PREPARE CASSETTE FOR REPLAY
.":? "PRESS PLAY, THEN RETURN."
30 ? :OPEN #1,4,0,"C:"
40 INPUT #1,X$,Y$,Z
50 ? "X$ IS ";X$:?
60 ? "Y$ IS ";Y$:?
70 ? "Z IS ";Z

```

Fig. 6.2. Replaying variable values from tape.

All of this recording would not be of much use unless we could also replay the data. Figure 6.2 shows a program which will do just this for the data that we recorded in Fig. 6.1. Once again, we start by dimensioning, but the variables have been given different names! It's the values that we record, not the names, so this is quite permissible – and very useful. We have the usual printed message in line 20, and the line 30 contains the important OPEN instruction. We use Channel #1 again, but the figure that follows this is 4 now, because this is the code number for replaying. The rest of the instruction is as before. This prepares for the replay, and the action is decided by line 40. This time we use INPUT#1, and it lists once again the names of the variables. These must be separated by commas. This is important, because the PRINT#1 used semicolons, and it's easy to forget that the two are different. The variable names must match with the type of data that you are replaying – don't expect to be able to read a string value into a number variable name! Finally, lines 50 to 70 print the values so that you can see that they have been replayed correctly.

## Array antics

One of the commonest uses for data recording is the recording and

```

10 ? " ":DIM A(40)
20 FOR N=1 TO 40
30 A(N)=10+INT(80*RND(0))
40 NEXT N:GOSUB 500:OPEN #1,8,0,"C:"
50 FOR J=1 TO 40
60 PRINT #1,A(J)
70 NEXT J
80 CLOSE #1
100 END
500 ? "PLEASE PREPARE CASSETTE.":? "PRES
S PLAY/RECORD KEYS, THEN RETURN."
510 RETURN

```

*Fig. 6.3.* Recording array values on tape.

replaying of array values. This means number arrays, because as we have seen, a string array is simulated by a long string variable. The recording of an array is pretty similar to the recording of simple variables, except that a loop is used. Figure 6.3 illustrates the technique. Lines 20 to 40 generate some numbers and fill an array with them. The subroutine then prints the warning about having a cassette ready, and this is followed by the usual OPEN step. The recording is done by lines 50 to 70, using PRINT#1,A(J) in a loop. The channel is closed in line 80, and that's it. Elementary, my dear Sir or Madam.

There's slightly more to replaying, as Fig. 6.4 shows. Lines 10 to 30 cover familiar ground, but there's a slight oddity in line 40. You cannot have an INPUT#1 to an array name, only to a simple variable name. Because of this, we have to have INPUT#1,Z, and then follow this with another assignment, X(Q)=Z, so as to get the value into the array. Lines 50 to 70 then print the values. All in all, there isn't much more to it.

```

10 ? " ":DIM X(40)
20 GOSUB 500
30 OPEN #1,4,0,"C:":FOR Q=1 TO 40
40 INPUT #1,Z:X(Q)=Z:NEXT Q:CLOSE #1
50 ? " ":FOR J=1 TO 39 STEP 2
60 ? X(J);" ";X(J+1)
70 NEXT J
100 END
500 ? "PLEASE PREPARE CASSETTE.":? "PRES
S THE PLAY KEY , THEN RETURN."
510 RETURN

```

*Fig. 6.4.* Replaying array values – note that you cannot INPUT #1 directly to an array variable.

## Your own creation

You can get a lot of enjoyment from your Atari when you use it with

cartridges or cassettes that you have bought. You can obtain even more enjoyment from typing in programs that you have seen printed in magazines. Even more rewarding is modifying one of these programs so that it behaves in a rather different way, making it do what suits you. Another step to mastery of your Atari is reached when you can modify a program that was written for another machine. The pinnacle of satisfaction, as far as computing is concerned, however, is achieved when you design your own programs. These don't have to be masterpieces. Just to have decided what you want, written it as a program, entered it and made it work is enough. It's 100% your own work, and you'll enjoy it all the more for that.

Now I can't tell in advance what your interests in programs might be. Some readers might want to design programs that will keep tabs on a stamp collection, a record collection, a set of notes on food preparation or the technical details of LNER steam locomotives. Programs of this type are called *database* programs, because they need a lot of data items to be typed in and recorded. On the other hand, you might be interested in games, colour patterns, drawings or sound. Programs of that type need instructions that we have not looked at yet, and they are dealt with in the next three chapters. What we are going to look at in this section is the database type of program, because it's designed in a way that can be used for all types of programs.

Two points are important here. One is that experience counts in this design business. If you make your first efforts at design as simple as possible, you'll learn much more from them. That's because you're more likely to succeed with a simple program first time round. You'll learn more from designing a simple program that works than from an elaborate program that never seems to do what it should. The second point is that program design has to start with the computer switched off, preferably in another room! The reason is that program design needs planning, and you can't plan properly when you have temptation in the shape of a keyboard in front of you. Get away from it!

### **Put it on paper**

We need to start with a pad of paper. I use a 'student's pad' of A4 which is punched so that I can put sheets into a file. This way, I can keep the sheets tidy, and add to them as I need. I can also throw away

any sheets I don't need, which is just as important. Even a very simple program is probably going to need more than one sheet of paper for its design. If you then go in for more elaborate programs, you may easily find yourself with a couple of dozen sheets of planning and of listing before you get to the keyboard. Just to make the exercise more interesting, I'll take an example of a program, and design it as we go. This will be a fairly simple program, but it will illustrate all the skills that you need.

You start by writing down what you expect the program to do. You might think that you don't need to do this, because you know what you want, but you'd be surprised. If you don't write down what you expect a program to do, it's odds on the program will never do it! The reason is that you get so involved in details when you start writing the lines of BASIC that it's astonishingly easy to forget what it's all for. If you write it down, you'll have a goal to aim for, and that's as important in program design as it is in life. Don't just dash down a few words. Take some time about it, and consider what you want the program to be able to do. If you don't know, you can't program it!

---

Aims:

1. Present the name of a country on the screen.
  2. Ask what its capital city is called.
  3. The reply must be correctly spelled.
  4. User must not be able to read the answer from a listing.
  5. Give one point for each correct answer.
  6. Allow two chances at each question.
  7. Keep a track of the number of attempts.
  8. Present the score as a number of successes out of number of attempts.
  9. Pick country names at random.
- 

*Fig. 6.5.* A program outline plan. This is your starter!

As an example, take a look at Fig. 6.5. This shows a program outline plan for a simple game. The aim of the game is to become familiar with the capital cities of some countries around the world. The program plan shows what I expect of this game. It must present the name of a country on the screen, and then ask what the name of the capital city is. A little bit more thought produces some additional points. The name of the city will have to be correctly spelled. A little bit of trickery will be needed to prevent the user (son, daughter, brother, sister) from finding the answers by typing LIST and looking for the DATA lines. Every game must have some sort of scoring system, so

we allow one point for each correct answer. Since spelling is important, perhaps we should allow more than one try at each question. Finally, we should keep track of the number of attempts and the number of correct answers, and present this as the score at the end of each game. Now this is about as much detail as we need, unless we want to make the game more elaborate. For a first effort, this is quite enough. How do we start the design from this point on?

The answer is to design the program the way an architect designs a house. That means designing the outlines first and the details later. The outlines of this program are the steps that make up the sequence of actions. We shall, for example, want to have a title displayed. Give the user time to read this, and then show some instructions. There's little doubt that we shall want to do things like assign variable names, dimension arrays, and other such preparation. We then need to play the game. The next thing is to find the score, and then ask the user if another game is wanted. Yes, you have to put it all down on paper! Figure 6.6 shows what this might look like at this stage.

- 
1. Display title, then instructions.
  2. Display name of country on the screen.
  3. Ask for the name of the capital.
  4. Use INPUT for reply.
  5. Compare reply with correct answer.
  6. If correct, add 1 to score and ask if another one is wanted.
  7. If incorrect, allow another try.
  8. If second attempt is also incorrect, select another question.
  9. Ends when user types N in response to 'Do you want another one?'
- 

*Fig. 6.6.* The next stage in expanding the outline.

## Putting down foundations

Now, at last, we can start writing a chunk of program. This will just be a foundation, though. What you must avoid at all costs is filling pages with BASIC lines at this stage. As any builder will tell you, the foundation counts for a lot. Get it right, and you have decided how good the rest of the structure will be. The main thing you have to avoid now is building a wall before the foundation is complete!

Figure 6.7 shows what you should aim for at this stage. There are only fifteen lines of program here, and that's as much as you want. This is a foundation, remember, not Nelson's Column! It's also a

program that is being developed, so we've hung some 'danger - men at work' signs around. These take the form of the lines that start with REM. REM means REMinder, and any line of a program that starts with REM will be ignored by the computer. This means that you can type whatever you like following REM, and the point of it all is to allow you to put notes in with the program. These notes will not be printed on the screen when you are using the program, and you will see them only when you LIST. In Fig. 6.7, I have put the REM notes on lines which are numbered just 1 more than the main lines. This way, I can remove all the REM lines later. How much later? When the program is complete, tested, and working perfectly. REMs are useful, but they make a program take up more space in memory, and run slightly more slowly. I always like to keep one copy of a program with the REMs in place, and another 'working' copy which has no REMs. That way I have a fast and efficient program for everyday use, and a fully-detailed version that I can use if I want to make changes.

```

10 ? "}:GOSUB 1000
11 REM TITLE
15 OPEN #1,4,0,"K:"
20 GOSUB 1200
21 REM INSTRUCTIONS
30 GOSUB 1400
31 REM SETUP
40 GOSUB 2000
41 REM PLAY
50 GOSUB 3000
51 REM SCORE
60 GOSUB 4000
61 REM ANOTHER?
80 IF K=89 THEN 40
100 END

```

Fig. 6.7. A 'core' or 'foundation' program for the example.

Let's get back to the program itself. As you can see, it consists of a set of GOSUB instructions, with references to lines that we haven't written yet. That's intentional. What we want at this point, remember, is foundations. The program follows the plan of Fig. 6.6 exactly, and the only part that is not committed to a GOSUB is the IF in line 80. What we shall do is to write a subroutine which will use GET to look for a 'Y' or 'N' being pressed, and line 80 deals with the answer. What's the question? Why, it's the 'Do you want another game' step that we planned for earlier.

Take a good long look at this piece of program, because it's important. The use of all the subroutines means that we can check

this program easily – there isn't much to go wrong with it. We can now decide in what order we are going to write the subroutines. The wrong order, in practically every example, is the order in which they appear. Always write the title and instructions last, because they are the least important to you at this stage. In any case, if you write them too early, it's odds on that you will have some bright ideas about improving the game soon enough, and you will have to write the instructions all over again. A good idea at this stage is to write a line such as:

```
9 GO TO 30
```

which will cause the program to skip over the title and instructions. This saves a lot of time when you are testing the program, because you don't have the delay of printing the title and instructions each time you run it.

The next step is to get to the keyboard (at last!) and enter this core program. If you use the GOTO step to skip round the title and instructions temporarily, you can then put in simple PRINT lines at each subroutine line number. We did this, you remember, in the program of Fig. 4.15, so you know how to go about it. This allows you to test your core program and be sure that it will work before you go any further.

### Next steps

The procedure now is to keep adding to the core. If you have the core recorded, then you can load this into your Atari, add one of the subroutines, and then test. When you are satisfied that it works, you can record the whole lot on another cassette. Next time you want to add a subroutine, you start with this version, and so on. This way, you keep tapes of a steadily growing program, with each stage tested and known to work. In addition, if anything goes wrong, you have all your previous stages of the program on tape.

```
4000 ? "WOULD YOU LIKE ANOTHER ONE?":? "  
PLEASE ANSWER Y OR N."  
4010 GET #1,K:RETURN
```

Fig. 6.8. The subroutine for line 4000.

### Subroutine routine

The next thing we have to do is to design the subroutines. Now some of these may not need much designing. Take, for example, the

subroutine that is to be placed in line 4000. This is just our familiar GET routine, along with a bit of PRINT, so we can deal with it right away. Figure 6.8 shows the form it might take. The subroutine is straightforward, and that's why we can deal with it right away! Type it in, and now test the core program with this subroutine in place.

### The hard part

Now we come to what you might think is the hardest part of the job – the subroutine which carries out the Play action. In fact, you don't have to learn anything new to do this. The Play subroutine is designed in exactly the same way as we designed the core program. That means we have to write down what we expect it to do, and then arrange the steps that will carry out the action. If there's anything that seems to need more thought, we can relegate it to a subroutine to be dealt with later.

- 
1. Keep the answers as an array of ASCII codes in a long string. Use a number array to keep positions.
  2. Keep list of countries in another string, with positions located by another number array.
  3. The number which selects the country will also select the answer.
  4. Use variable TRY to record tries. SCORE to keep score.
  5. Use variable GO to record the number of attempts at one question.
- 

*Fig. 6.9.* Planning the 'Play' subroutine.

As an example, take a look at Fig. 6.9. This is a plan for the Play subroutine, which also includes information that we shall need for the setting-up steps. The first item is the result of a bit of thought. We wanted, you remember, to be sure that some smart user would not cheat by looking up the answers in the DATA lines. The simplest deterrent is to make the answers in the form of ASCII codes. It won't deter the more skilled, but it will do for starters. The codes can be held as one long string, and we shall also keep the names of the countries in a long string. We shall treat these strings as arrays, with separate number arrays used to indicate where each name starts and finishes. This has several advantages. One of them is that it's beautifully easy to select one at random if we do this. The other is that it also makes it easy to match the answers to the questions. If, for example, we select a number V for an item, we can use an array item, ARRAY(V) to find the name that corresponds to that item. The method should already be familiar from Chapter 5. We can then



use V to find another number Q, so that ARRAY(Q) gives the answer! If that sounds confusing, the detailed explanations of the subroutines will make it much clearer.

The next thing that the plan settles is the names that we shall use for variables. It always helps if we can use names that remind us of what the variables are supposed to represent. In this case, using 'SCORE' for the score and 'TRY' for the number of tries looks self-explanatory. The third one, 'GO' is one that we shall use to count how many times one question is attempted. Finally, we decide on a name for the string that will hold the country names -Q\$.

```

2000 GO=0:V=1+INT(10*RND(0))
2001 REM PICK AT RANDOM
2010 ? ")":POSITION 6,5: ? "THE COUNTRY I
S -";
2020 ? Q$(ARRAY(V),ARRAY(V+1)-1)
2030 POSITION 8,8: ? "THE CAPITAL IS- ";
2040 INPUT CAP$:TRY=TRY+1
2050 GOSUB 5000
2051 REM FIND CORRECT ANSWER
2060 RETURN

```

Fig. 6.10. The program lines for the 'Play' subroutine.

## Play for today

Figure 6.10 shows what I've ended up with as a result of the plan in Fig. 6.9. The steps are to pick a random number V, use it to print a country name, and then find the answer. That's all, because the checking of the answer and the scoring is dealt with by another subroutine. Always try to split up the program as much as possible, so that you don't have to write huge chunks at a time. As it is, I've had to put another subroutine into this one to keep things short.

We start in line 2000 by picking a number, at random, lying between 1 and 10. As before, we use line 2001 to hold a REM that reminds us of what's going on. Line 2010 is straightforward; the line 2020 picks the country name out of Q\$. This is done by using the values stored in the number array, and the method is exactly the one that was illustrated in Fig. 5.15. We print the name of the country that corresponds to the random number, and ask for an answer, the capital of that country. The last section of line 2040 counts the number of attempts. This is the logical place to put this step, because we want to increase the count each time there is an answer. Now it's chicken-out time! I don't want to get involved in the reading of

1. For correct answer, increment SCORE.
2. For first incorrect answer, with GO=0, allow another try and make GO=1.
3. For second incorrect answer, when GO=1, pass the next question and make GO=0 again.

*Fig. 6.11.* Planning the 'Score' subroutine.

ASCII codes right now, so I'll leave it to a subroutine, starting in line 5000, which I'll write later. The REM in line 2051 reminds me what this new subroutine will have to do, and the Play subroutine ends with the usual RETURN.

### Working at the details

With the Play subroutine safely on tape, we can think now about the details. The first one to look at should be one that precedes or follows the Play step, and I've chosen the Score routine. As usual, it has to be planned, and Fig. 6.11 shows the plan. Each time that there is a correct answer, the number variable 'SCORE' will be incremented, and we can go back to the main program. More is needed if the answer does not match exactly. We need to print a message, and allow another go. If the result of this next go is not correct, that's an end to the attempts.

Figure 6.12 shows the program subroutine developed from this plan. Line 3000 deals with a correct answer, by comparing your answer, CAP\$, with the correct answer, ANS\$. The GOTO 3030 ensures that if the answer was correct, the rest of the subroutine is skipped, and the subroutine returns. If the answer is not correct, though, line 3010 swings into action. This prints a message, and then calls the subroutine at line 2010 again so that the user can make another answer entry. The GOTO 3000 at the end of line 3010 then tests this answer again.

```

3000 IF ANS$=CAP$ THEN SCORE=SCORE+1: ? :
? M1$; " "; SCORE: ? " IN "; TRY; " ATTEMPTS.
": GOSUB 7000: GOTO 3030
3010 IF GO=0 THEN ? M2$: GOSUB 7000: GO=1:
GOSUB 2010: GOTO 3000
3020 GO=0: ? "NO LUCK- TRY THE NEXT ONE."
3030 RETURN

```

*Fig. 6.12.* The 'Score' subroutine written.

Now there's a piece of cunning here. The number variable 'GO' must start with a value of 0 (make a note of it!). When there is an incorrect answer, however, and 'GO' is still 0, line 3010 is carried out. One of the actions of line 3010, however, is to set 'GO' to 1. When you answer again, with GO=1, line 3000 will be used, and if your second answer is wrong, line 3010 cannot be used, because 'GO' is not zero. The next line that is tried, then is 3020. This puts 'GO' back to zero for the next round, prints a sympathetic message, pauses, and then lets the subroutine return in line 3030.

```

1400 TRY=0: SCORE=0: GO=0
1405 DIM M2$(64), M1$(32), Q$(80), NUM$(160
), NUN$(70), ARRAY(45)
1406 DIM CAP$(20), ANS$(20), REP$(40)
1410 M2$="NOT CORRECT- BUT IT MIGHT BE Y
OUR      SPELLING. TRY AGAIN - FREE!"
1412 M1$="CORRECT-YOUR SCORE IS NOW "
1415 Q$="ALBANIAHOLLANDALGERIANDORWAYCOLO
MBIATURKEYLIBERIAINDONESIAPAKISTANCHILE"
1420 NUM$="84738265786565778384698268657
7657671736982837983767966797179846565787
56582657779788279867365"
1425 NUN$="74658275658284657383766577656
665688365788473657179"
1430 NUM$(LEN(NUM$)+1)=NUN$
1435 FOR N=1 TO 22: READ A: ARRAY(N)=A: NEX
T N
1450 RETURN

```

Fig. 6.13. The dimensioning and array subroutine.

Now that we've got the bit between our teeth, we can polish off the rest of the subroutines. Figure 6.13 shows the subroutine that deals with dimensioning and arrays. Line 1400 sets all the variables for the scoring system to zero. Lines 1405 and 1406 dimension all the strings and arrays that will be used for the names in the program. Lines 1410 and 1412 create strings that will be used for messages. The point of this is that if we printed these messages out in full, we would either have lines that were too long, or we would have to use GOTO instructions all over the place. Lines 1415 to 1425 then create the strings that hold the names of the countries and the ASCII codes for their capitals, using the same order. The codes have to be put into two strings and then combined, because of the limit to the number of characters that can be typed following one line number. The combining operation is done in line 1430. Finally, the array of numbers that holds the positions of question and answer words is read in line 1435, and the subroutine returns in line 1450.

```

5000 Q=V+11:REP$=NUM$(ARRAY(Q),ARRAY(Q+1
)-1)
5010 ANS$="":FOR N=1 TO LEN(REP$) STEP 2
5015 J=(N+1)/2
5020 ANS$(J,J)=CHR$(VAL(REP$(N,N+1)))
5030 NEXT N
5040 RETURN

```

*Fig. 6.14.* Checking the answer.

```

1200 ? "}":? "          INSTRUCTIONS"
1210 ? :? "YOU WILL BE GIVEN THE NAME OF
  A ":? "COUNTRY. YOU ARE ASKED TO TYPE T
HE"
1220 ? "NAME OF ITS CAPITAL CITY, AND TH
EN":? "PRESS ENTER. YOU ARE ALLOWED TWO"
1230 ? "ATTEMPTS AT EACH COUNTRY.THE COM
PUTER":? "WILL KEEP THE SCORE."
1240 ? "PRESS ANY KEY TO START."
1250 GET #1,X:RETURN

```

*Fig. 6.15.* The instructions - always leave these until you have almost finished.

Next comes the business of finding the answer. We have planned this, so it shouldn't need too much hassle. Figure 6.14 shows the program lines. The variable V is the one that we have selected at random, and we add 11 to it so as to find the corresponding answer number. Since there are ten questions, the eleventh item in the array is the answer to the first question, hence the 11. We extract the string of numbers from NUM\$, and assign it to REP\$, which is the reply for this question. We now have to convert each number in REP\$ into a letter and add it to the string ANS\$, which starts off blank in line 5010. Lines 5010 to 5030 build up the answer string. There are two digits in each ASCII code, so we need the STEP of 2 in the loop. The limit of the loop is the number of characters in REP\$, and we obtain this by using LEN. Line 5015 is a formula for J which picks out values of 1,2,3 and so on as N goes in steps of 1,3,5, etc. Line 5020 then places each letter into the answer string, using J as the position number. The letter has to be found by extracting the number from REP\$ - using REP\$(N,N+1) - then VAL to get the number form, then CHR\$ to convert to a letter. That's the hard work over!

Figure 6.15 is the subroutine for the instructions, and Fig. 6.16 is the title subroutine. Each of them includes a pause. Finally Fig. 6.17 shows the DATA lines along with another pause subroutine. Now we can put it all together, and try it out. Because it's been designed in

```

1000 ? "}"
1010 POSITION 15,5: ? "CAPITALS"
1030 FOR N=1 TO 3000:NEXT N
1040 RETURN

```

Fig. 6.16. The title subroutine.

```

6000 DATA 1,8,15,22,28,36,42,49,58,66,71
,1,13,31,45,53,65,77,93,109,127,143
7000 FOR N=1 TO 200:NEXT N:RETURN

```

Fig. 6.17. The DATA lines that are needed, along with a time delay subroutine.

sections like this, it's easy for you to modify it. You can use different names, for example. You can use a lot more names – but remember to change the dimensioning. You can make it a question-and-answer game on something entirely different, just by changing the data and the instructions. Take this as a sort of BASIC 'Meccano set' to reconstruct in any way you like. It will give you some idea of the sense of achievement that you can get from mastering your Atari!

## Chapter Seven

# The Coarser Characters

The word 'resolution' keeps turning up when we discuss computer graphics. Graphics, to start with, means drawing *pictures* on the screen rather than letters or numbers. The resolution of graphics is a measure of how much detail the pictures can show. Think of it this way. Suppose you were presented with a picture frame that was 12" × 9" in size. Imagine that this area was divided into 1000 square holes, and that you were told you could make a picture by placing a coloured square into each hole. There's no way that you could make a picture like this which could display any fine detail, and it would look reasonably good only if you viewed it at a distance. That's what we mean by *low resolution*. The squares are the picture elements (or *pixels*), and a comparatively small number of pixels per picture means low resolution. On the other hand, if we had divided the frame into 60,000 squares, it would be possible to

Mode No.	Type	Columns	Row (Split screen)	Row (Full screen)	Colours	Bytes of memory used
∅	Text	40	-	24	2	993
1	Text	20	20	24	5	513
2	Text	20	10	12	5	261
3	Graphics	40	20	24	4	273
4	Graphics	80	40	48	2	537
5	Graphics	80	40	48	4	1017
6	Graphics	160	80	96	2	2025
7	Graphics	160	80	96	4	3945
8	Graphics	320	160	192	1 or 2	7900
9	Graphics	- see text				

and higher for examples.

Fig. 7.1. The Graphics Modes of the Atari. These are programmed by using the GRAPHICS instruction.

create pictures which showed much more detail. These would be *high resolution* pictures. We can measure the resolution of pictures by the number of pixels that will fit into the frame. A good TV monitor can operate with about 250,000 pixels, but computers generally don't give resolution as high as this. In any case, an ordinary TV receiver can't cope with the sort of resolution that you can get from a monitor.

Your Atari computer allows you a very wide choice of both text and graphics arrangements. These arrangements are called *modes*, and Fig. 7.1 shows what is available. The text modes allow text (letters and numbers), as well as some graphics shapes, to appear on the screen. The simplest of these text modes is mode  $\emptyset$ , which is the mode that is automatically selected when the Atari is switched on. If you have been using the Atari in any other mode, you can select this normal one by typing GRAPHICS  $\emptyset$  (or GR. $\emptyset$ ) and then pressing RETURN. These different graphics modes require different amounts of memory, allow different combinations of colours, and have different figures of resolution.

### Mode zero manipulations

Mode zero allows you to use a row of 4 $\emptyset$  characters, though the preset margins limit this to 37 characters unless you change them. The number of rows is 24. Two colours can be displayed, and the standard arrangement is blue for the main screen and black for the border that surrounds it. Text on the screen appears as light blue on a darker blue background. We can, however, change these border and screen colours. When we change the background colour for the screen, the characters of text will always appear as a lighter shade of the same colour.

The instruction that has to be used to decide on the colour choice is SETCOLOR (or SE). SETCOLOR has to be followed by three numbers, thus:

SETCOLOR A,B,C

A is the register number

B is the colour number

C is the luminance number

The first number is a 'register' number, a register being a store for information, and some registers in the Atari are used to store information on items like colour and sound. The registers are

Colour No.	Colour	Colour No.	Colour
0	Grey (Black at zero luminance)	8	Dark blue
1	Gold	9	Green-blue
2	Orange	10	Blue
3	Red	11	Dark blue
4	Pink	12	Green
4	Violet	13	Dark green
6	Purple	14	Olive green
7	Light blue	15	Orange

*Fig. 7.2.* The colours that can be used with the SETCOLOR instruction. The colour numbers are listed as they appear at luminance 8. The actual colour that you see depends on the luminance value, and also to some extent on the tuning of the TV receiver.

numbered, and the one we can use to control the screen colour in Mode 0 is register 2. The second number in the SETCOLOUR instruction then specifies which colour we can use on the main part of the screen. The numbers that we can use range from 0 to 15, and are listed in Fig. 7.2. The third number is called the 'luminance' number, and it decides how bright the colour will be. The range is of the even numbers from 0 to 14. If you use an odd number, the effect will be the same as that of the next lower even number. Using a luminance value of zero makes the text on the screen appear very dark; a value of 14 makes it look almost white. The normal luminance value for text is 8.

Figure 7.3 is a program which illustrates this use of SETCOLOR. The loop in lines 10 to 50 places the complete range of colour numbers into register 2, one at a time, so that you can see the range of screen colours. The next loop, in lines 60 to 100, alters the

```

10 ? "3":FOR N=0 TO 15:GRAPHICS 0
20 SETCOLOR 2,N,8
30 ? "COLOUR ";N
40 FOR J=1 TO 1000:NEXT J
50 NEXT N:GRAPHICS 0
60 FOR N=0 TO 14 STEP 2
70 SETCOLOR 1,8,N
80 ? "LUMINANCE ";N
90 FOR J=1 TO 500:NEXT J
100 NEXT N

```

*Fig. 7.3.* A program which illustrates the use of SETCOLOR in Mode 0.



luminance of letters on the screen. These are controlled by the number in register 1, so that SETCOLOR 1,8,N keeps the colour constant (colour No. 8), but alters the luminance. When the luminance of the letters is the same as the luminance of the screen,

```

10 GRAPHICS 0:FOR N=0 TO 15
20 SETCOLOR 4,N,7:? "BORDER ";N
30 FOR J=1 TO 1000:NEXT J
40 NEXT N

```

Fig. 7.4. Altering the border colours.

the letters are invisible – a very useful way of making letters disappear and reappear instantly.

The other form of control that you can exercise with SETCOLOR in mode 0 is on the border colour. The border is controlled by register 4, and is normally set to a sombre black. We can alter this, however, by use of SETCOLOR, and Fig. 7.4 shows this effect. The loop that starts in line 10 runs through the complete range of border colours, so that the border can be set to any colour that you use for the rest of the screen, or to a contrasting colour as you please. It is the particularly wide range of colours and luminances that make the Atari so very effective for striking displays, and we'll slowly unravel some of its many secrets in this and the following chapters.

Now take a look at an extension to our notions of mode 0 graphics, in Fig. 7.5. As it stands, lines 10 to 50 cause the word ATARI! to flash on the screen. This is because the background is being printed alternately dark (luminance 2) and bright (luminance 14). The text is being printed in a constant value of luminance. If you lengthen the delay loop in line 100, you will be able to see these changes in slow motion. In line 50, GRAPHICS 0 (which you can

```

10 GRAPHICS 0:FOR J=1 TO 20
20 SETCOLOR 2,8,2:POSITION 17,12
30 GOSUB 100
40 SETCOLOR 2,8,14:POSITION 17,12
45 GOSUB 100
50 NEXT J:GRAPHICS 0
60 POKE 752,1:FOR J=1 TO 6:READ D
70 COLOR D:PLOT 16+J,14:NEXT J
80 GOTO 80
90 DATA 65,84,65,82,73,33
95 END
100 PRINT "ATARI!":FOR X=1 TO 20:NEXT X:
RETURN

```

Fig. 7.5. Using different background luminance values to flash a complete screen.

type as GR.Ø – the computer will always print the whole word) will clear the screen. Line 6Ø then introduces a useful instruction, POKE 752,1. This makes the cursor invisible, and it's a handy way of preventing your graphics from being spoiled by having the cursor appear to be tacked on to them.

This brings us to another instruction word. COLOR. Now this one has different uses in different modes. In Mode Ø, where we are at the moment, COLOR can be followed by a number. This will normally be an ASCII code number, and its effect will be to print the letter that corresponds to the code. The printing action is not carried out by COLOR, though, but by another instruction PLOT. PLOT, like POSITION, uses the column and row numbers following it to determine where on the screen the letter will appear. We could do the same by using POSITION and PRINT instructions, but it's useful to have the alternative. Lines 6Ø to 1ØØ illustrate this in action. Note that line 8Ø forms an endless loop. This is done to prevent the READY appearing at the end of the program, and it's something that you will see a lot of in graphics programs. Figure 7.6 shows what colours are available in each mode if the SETCOLOR numbers are left at their 'default' values, that is the values that are fed into them when you switch on the computer.

Register	Colour	Luminance	Appearance
Ø	2	8	Orange
1	12	1Ø	Sea blue/green
2	9	4	Blue
3	4	6	Pale red
4	Ø	Ø	Black

*Fig. 7.6.* The default register colours in each mode.

## Modes 1 and 2

Modes 1 and 2, which we can take together because they are so similar, are also text modes. They have, however, some features which don't exist in mode Ø. To start with, the screen is split in these modes. Screen splitting means that different parts of the screen can be used for different purposes. Some five-sixths of the screen height, starting at the top, is reserved for display. This means the display of

text or graphics, anything that uses COLOR and PLOT, along with some types of PRINT instructions. The remaining one-sixth of the screen at the bottom is used for listing the program, putting in commands, or for putting in the words that are specified by PRINT instructions. The cursor and the READY will appear only on this lower section of the screen. When you are in graphics modes 1 or 2 and you LIST, then the listing will be only in this small 'window' of the screen. This can be a nuisance if you want to see the whole of a short listing, and it's often useful to type GR.Ø and press RETURN before listing. In these modes, the PRINT instruction by itself will cause printing on the lower text window. If you want to use PRINT in the main part of the screen, you have to use the modified instruction PRINT#6. This is what Channel 6 is reserved for, incidentally. You can, however, still use COLOR and PLOT in the same way as on the Mode Ø screen.

Modes 1 and 2 allow you a lot more choice of colours, both for background and for characters. As before, these colours are decided by the SETCOLOR instructions, but if you don't use SETCOLOR, there are default values. These are black for border and background, and characters in orange, light green, dark blue and red. How do we decide what character colours are used? Simply by the choice of the code number for the characters. The strict ASCII code is abandoned for these modes. Each character can be printed by four different code numbers, and each different code number specifies a different colour for the character. For example, the letter A can be placed on the screen using code 65 (the normal ASCII code for A), which gives orange. Code 97 gives A in light green, 193 gives dark blue, and 225 gives red. The program in Fig. 7.7 runs through the choice. Try it

```

10 GRAPHICS 1:FOR N=1 TO 255 STEP 32
20 FOR J=0 TO 31
30 IF N+J=125 THEN 50
40 PRINT #6;N+J;" ";CHR*(N+J);" ";
50 NEXT J
60 A=PEEK(764):IF A=255 THEN 60
70 POKE 764,255
80 PRINT #6;CHR*(125)
90 NEXT N

```

Fig. 7.7. The use of non-ASCII codes in Mode 1 (and 2) to print characters in colour.

out, and then replace the GRAPHICS 1 in line 10 by GRAPHICS 2 to see the difference. The table in Fig. 7.8 shows what to expect. The colours that are produced are the result of the code numbers that

Value for Colour Register				Character		Value for Colour Register				Character	
0	1	2	3	Standard	Alternative	0	1	2	3	Standard	Alternative
32	0	160	128	□	♥	64	96	192	224	@	◆
33	1	161	129	!	†	65	97	193	225	A	a
34	2	162	130	"	‡	66	98	194	226	E	h
35	3	163	131	#	§	67	99	195	227	C	c
36	4	164	132	\$	¶	68	100	196	228	D	d
37	5	165	133	%	‡	69	101	197	229	E	e
38	6	166	134	&	§	70	102	198	230	F	f
39	7	167	135	'	¶	71	103	199	231	G	g
40	8	168	136	(	§	72	104	200	232	H	h
41	9	169	137	)	¶	73	105	201	233	I	i
42	10	170	138	*	§	74	106	202	234	J	j
43	11	171	139	+	¶	75	107	203	235	K	k
44	12	172	140	,	§	76	108	204	236	L	l
45	13	173	141	-	¶	77	109	205	237	M	m
46	14	174	142	.	§	78	110	206	238	N	n
47	15	175	143	/	¶	79	111	207	239	O	o
48	16	176	144	0	§	80	112	208	240	P	p
49	17	177	145	1	¶	81	113	209	241	Q	q
50	18	178	146	2	§	82	114	210	242	R	r
51	19	179	147	3	¶	83	115	211	243	S	s
52	20	180	148	4	§	84	116	212	244	T	t
53	21	181	149	5	¶	85	117	213	245	U	u
54	22	182	150	6	§	86	118	214	246	V	v
55	23	183	151	7	¶	87	119	215	247	W	w
56	24	184	152	8	§	88	120	216	248	X	x
57	25	185	153	9	¶	89	121	217	249	Y	y
58	26	186	154	:	§	90	122	218	250	Z	z
59	27	187	None	;	¶	91	123	219	251	[	↑
60	28	188	156	<	§	92	124	220	252	\	
61	29	189	157	=	¶	93	None	221	253	]	↵
62	30	190	158	>	§	94	126	222	254	^	↶
63	31	191	159	?	¶	95	127	223	255	_	↷

Fig. 7.8. Table of colours and characters for Modes 1 and 2.

```

5 GRAPHICS 1
10 PRINT CHR$(125):FOR J=0 TO 3:FOR X=0
TO 11:SETCOLOR J,X+J,6:GOSUB 1000
20 FOR Y=1 TO 50:NEXT Y
30 NEXT X:NEXT J
100 END
1000 FOR N=1 TO 18:READ D
1010 COLOR D:PLOT 1+N,5:NEXT N
1020 RESTORE :RETURN
1030 DATA 65,116,193,242,73,0,109,197,22
5,78,115,0,195,239,76,111,213,242

```

Fig. 7.9. Obtaining differently coloured letters in a message.

exist in registers 0, 1, 2, and 3. These colour and luminance values can be altered by using the SETCOLOR instruction, as you might expect. Figure 7.9 demonstrates the use of SETCOLOR in this way and shows how effective Modes 1 and 2 are for coloured letters. Try this one also with GRAPHICS 2 replacing GRAPHICS 1 in line 5.

The characters that have been displayed on the screen by these last two programs are not the only ones that exist in Modes 1 and 2. There is an alternate character set, which is obtained by a POKE operation to address 756. We have made some use of these POKE and PEEK operations, and now it's time to explain them. Each unit of memory of a computer can store a number, and the range of numbers is 0 to 255. So that we can get at each unit (or byte) of memory, each one is numbered, and this number is, very reasonably, called an 'address number'. A PEEK allows us to find what is stored at an address, so that PEEK(756) will, for example, give the number stored at address 756. This can be printed or assigned to a number variable. POKE performs the opposite operation of altering what is

```

10 GRAPHICS 1
20 N=0:GOSUB 100
30 N=N+32:GOSUB 100
40 N=N+96:GOSUB 100
50 N=N+32:GOSUB 100
60 IF PEEK(756)=224 THEN POKE 756,226:GO
TO 20
65 END
70 DATA 71,82,65,80,72,73,67,83
100 FOR J=1 TO 8:READ D
110 COLOR N+D:PLOT 5+J,12
120 NEXT J
130 RESTORE
140 FOR X=1 TO 500:NEXT X
150 RETURN

```

Fig. 7.10. Using the alternate character set.

stored. POKE 756,226 will place the number 226 into memory address 756. The rules, if you don't know what's going on inside your computer, are simple. You can PEEK as much as you like, but don't POKE unless you know what you are doing! Having said that, try Fig. 7.10. This prints a message in the various colours, and then alters the content of address 756. The result of POKE 756,226 is to bring in the alternate character set, so that the characters change. You will have to carry out POKE 756,224, or use SYSTEM RESET to get back to normal again.

### Shapes on the screen

Using modes 0, 1, or 2, we can produce patterns on the screen by using the standard 'built-in' graphics blocks, or by the use of what are called 'user-defined' characters. The built-in graphics characters can be obtained from the keyboard by using the ESC key and the CTRL key. If, for example, you press ESC, then CTRL H, you will see a triangle shape appear. ESC followed by CTRL J will make the other triangle shape. These shapes are illustrated in the Atari manual. They can be printed in the same way as any other character. All you need to do is to type the ? mark followed by the usual quote mark. You then type the shapes that you want, using ESC followed by the CTRL key and whatever key will produce the character. At the end of a row of characters, type another quote mark, and the whole line can be printed. Unfortunately, I can't illustrate, because the special shapes do not appear on my printer.

The alternative, which leads to much more interesting graphics possibilities in these modes, is to create your own character shapes to

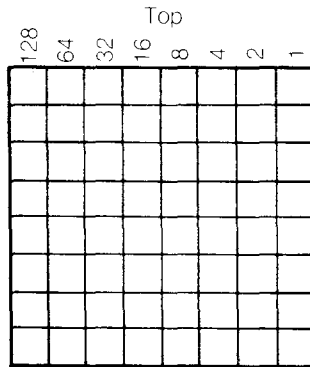


Fig. 7.11. A grid that you can use to design your own character shapes.

use in place of the ordinary text letters. This point is not covered at all in the Atari manual, so I shall deal with it in detail in this chapter. This facility, called *user-defined graphics* is available in most modern designs of computers. At the time when the Atari was designed, however, it was a most unusual feature, and there may be many Atari owners who do not even know that it is possible.

Each character that we can print on to the screen, either by using PRINT or by COLOR and PLOT, is stored in the memory of the computer as a set of eight code numbers. These code numbers are, in turn, derived from a set of 64 squares set in an 8 × 8 pattern as in Fig. 7.11. To create a character, you shade in squares on this block, or on tracing paper placed on top of this block. You then write down a code number for each row of eight squares. The code for each row is found by adding the column numbers for the shaded squares, as the illustration in Fig. 7.12 shows. In this way, you end up with a set of eight code numbers for each character. You must always have eight numbers. Even if there is a completely blank set of squares, its code is  $\emptyset$ , and this  $\emptyset$  must not be omitted. It's a machine you're dealing with, remember!

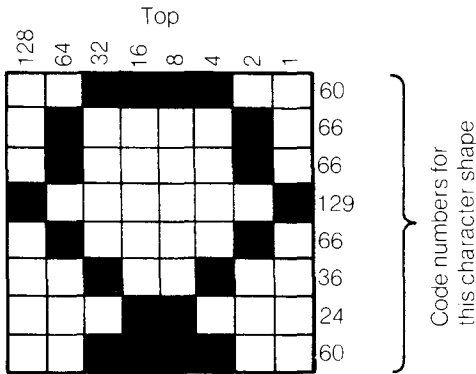


Fig. 7.12. An example of a 'user-defined' character.

The code numbers which are used for the standard characters are stored in memory which cannot be altered (called ROM - Read Only Memory). So that the computer can keep track of where these numbers are stored, however, the starting address for the collection of code numbers is stored at another place in the memory, and this one can be altered. We have already met this address - it's 756. If we carry out:

```
PRINT PEEK(756)
```

then we normally get the answer 224. This is not the actual address number for the characters, but we can find the true address number – it's  $256*224$ , which is 57344. The alternate character set is stored starting at address 57856, which is  $256*226$ . This is how we can obtain the two character sets that we use in Modes 1 and 2.

There's nothing to stop us from putting any other number into address 756, however. If the number that we place in address 756, when multiplied by 256, happens to be the address of unused read/write memory (RAM), then we can use that memory to place code numbers for any shapes that we want, and we can use these shapes for our own graphics patterns. The three stages in the process consist of designing the shape, finding some free memory, and then placing the code numbers into this memory.

### **Design-a-shape-time**

Designing your character shape is the easiest part of the exercise. Using the grid that was shown in Fig. 7.11, shade in the blocks that you want to see lit on the screen. There's no particular reason to work this way round – you can shade the blocks that you want to see dark if you like, but most users seem to prefer to see shapes appear as dark marks on white paper. We'll keep to one character block at the moment, and Fig. 7.12 shows an example. The next thing is to establish a set of eight code numbers for this character. We do so by adding up the codes that are shown on top of the block for each shaded piece of the grid. These numbers have been shown in the example.

The next step is to find some spare memory. Unless you have typed in an exceptionally long program, there will always be some spare memory left, particularly in the 48K machines. The computer keeps a track of this spare memory in address 742. The number that is stored here leads to the highest address that is free, and by subtracting from this number, we can reserve some memory for our own purposes. If we want to prepare a complete character set for Mode 0, we need to reserve 1024 units (bytes) of memory; for Modes 1 and 2 we need only 512 bytes. If we don't intend to change every character, we may need less than this, but we can't reserve less than 256 bytes.

The next step is a simple one. We POKE the eight data bytes that describe our character into the first eight bytes of memory starting at the address we find from PEEK(742), less the 1024 or 512 bytes



(depending on Mode). This can be done by using a FOR...NEXT loop that runs from 0 to 7, along with a variable name for the starting address. Suppose, for example, that we have found that we need to use the memory address 38917. We can assign this number to some variable name such as CH (for character). The first memory address that we shall use is CH+0, and the last is CH+7. If we now want to repeat the loop to store another character, then we can use CH=CH+8 to get to the next available memory location. We must use consecutive units of memory for our characters – or keep a very careful note of what addresses we do use.

As a result of all this, we will have a set of numbers stored in the memory starting at address CH, and we can use this as an alternate character set. We now have to ‘point’ the operating system of the computer at this new starting address, so that the computer takes its characters from this address rather than from 57344 or 57856. This is done by changing the number that is stored in address 756. The number that we need here is INT(CH/256). INT has been used because this must be a whole number.

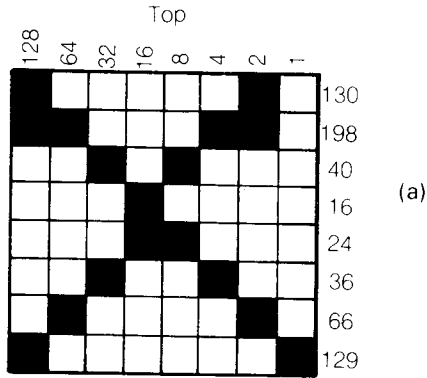
```

10 GRAPHICS 0:PRINT CHR$(125)
20 CH=(PEEK(742))*256-1024
30 FOR J=0 TO 7:READ X
40 POKE CH+J,X:NEXT J
50 POKE 756,INT(CH/256)
60 FOR Z=1 TO 2000:NEXT Z
70 POKE 756,224
100 DATA 60,66,66,129,66,36,24,60

```

*Fig. 7.13.* The program which places the numbers into memory so as to print the shape.

There's quite a lot to digest here, and it's best digested with the aid of an example, Fig. 7.13. We have already designed the character, so we can put its data numbers into a DATA line, and design the rest of the program to suit. We start as usual by selecting the graphics mode which also clears the screen. The next step, in line 20, consists of finding the address of the top of the memory (using CH=(PEEK(742))\*256). Since we are using Mode 0, we need a maximum of 1024 bytes of memory reserved, so we subtract this from the result. Having found a starting address 1024 bytes below the end of memory, and assigned this to CH, we can now POKE numbers into this address. This is done in lines 30 and 40, using the loop that runs from 0 to 7, as we outlined earlier. When the numbers are stored in the memory, we then have to replace the normal character set of the computer by our own set, and this is done in line



```

10 GRAPHICS 0:PRINT CHR$(125)
15 SETCOLOR 1,2,14:SETCOLOR 2,2,2
20 CH=(PEEK(742))*256-1024
30 FOR J=0 TO 7
40 POKE CH+J,0:NEXT J
44 FOR J=8 TO 15:READ D
46 POKE CH+J,D:NEXT J
50 POKE 756,INT(CH/256)
55 PRINT "!"
60 FOR Z=1 TO 2000:NEXT Z
70 POKE 756,224
100 DATA 130,198,40,16,24,36,66,129

```

Fig. 7.14. (a) The cross shape. (b) A better choice of memory will avoid filling the screen with characters!

50. After a delay programmed by line 60, the normal character set is restored by line 70.

We don't have to do anything special to make these characters appear on the screen! The normal character set of Mode 0 uses code numbers 0 to 127, a total of 128 characters. Since each character needs eight numbers in turn to describe it the whole set needs  $128 \times 8 = 1024$  bytes of memory, which is the reason for using this figure. The characters are stored in order, so that the first eight bytes of memory, starting from address CH, correspond to the character whose code number is 0. These code numbers are not ASCII codes, they are the 'internal' codes of the Atari, but the manual shows you how they correspond to the ASCII codes. The internal code 0 is a space, so that a blank screen is a screenful of code 0 characters. When we redefine this character, then, it appears all over the screen!

If we want to print any other characters, however, we will have to return to the normal character set. When this is done, in line 70, the screen will promptly change back to its normal appearance. If you

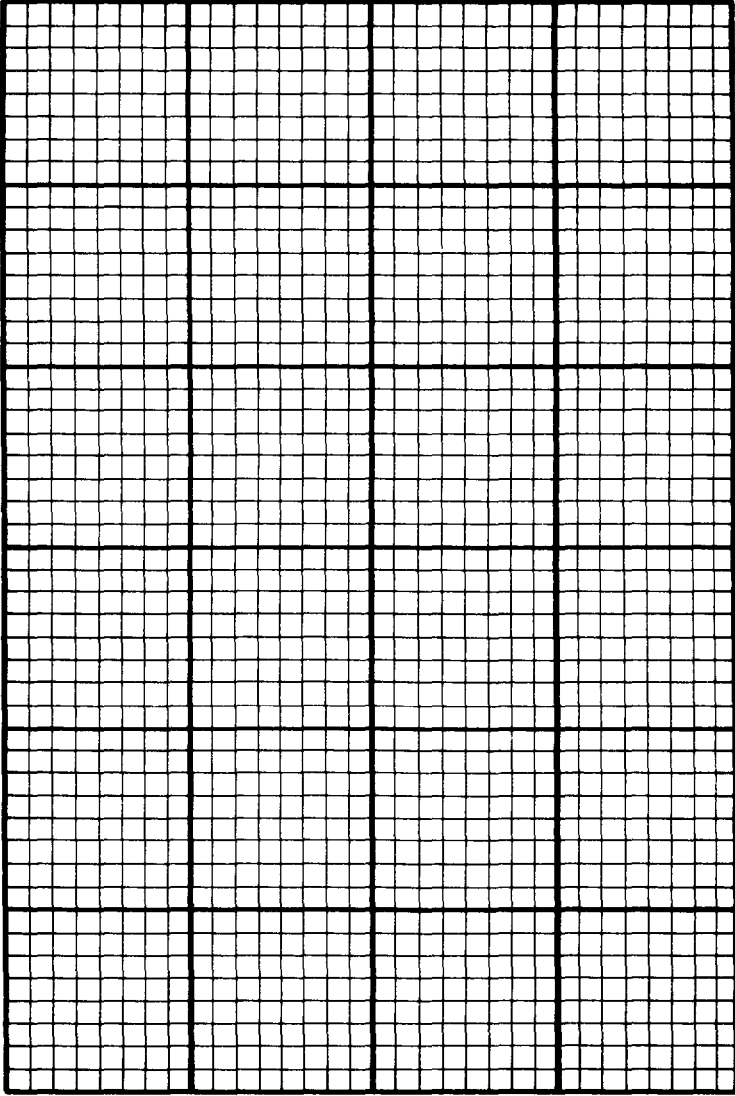
want to keep using your own characters, you need to put them all in at the memory addresses which start at address CH. For most purposes, we don't want to fill the screen with new characters, so we will want to replace a character other than the space character. Figure 7.14 illustrates this. The character is a cross shape which is created by the DATA line 100. We go through the usual procedure of reserving memory, but this time we fill the first eight bytes with zero. This ensures that the space character is still a space, so that the screen will not fill with shapes now. Our own data for the cross shape is put into the next eight bytes of memory by lines 44 and 46.

Having put the bytes in place, the next step is to switch to this new character set, if you can call two characters a set! This is done in line 50 and then we're ready to display the character. Now if you look again at the internal character set, you'll see that the character whose internal code is 1 is the exclamation mark. This is the key we must use to produce our own character on the screen, and it's done in line 55. The delay loop in line 60 gives you time to look at your handiwork, and then we return to the normal character set in line 70. Note that if you press BREAK before line 79 has been executed, you will be left with only one character that can be printed! Your commands, like GR.0 will, however, still be effective, though you won't see them appear on the screen.

### Cast of thousands?

An object which is only of the size of a single character doesn't make much impression. You'll find, incidentally, that you may need higher than normal luminance to make the character appear just as you want it. For really impressive shapes, you need to use several characters printed together, and this requires a lot more planning. You have to start with a multiple-character grid, as in Fig. 7.15. This allows for 24 character blocks, arranged as six across and four down. It's a size that should be ample for most purposes. To plan your shape, you place tracing paper over this grid and draw, keeping to the path of the small squares inside the 8x8 blocks. If you find that you need a larger grid, or if you want to work on a larger surface, draw out your own master grid. The ideal material is the old-fashioned eighth-inch graph paper, because the inch squares of such graph paper look ready-made for use as character blocks.

Suppose you have traced a pattern, like the one in Fig. 7.16. How do you go about plotting it on the screen? The answer is in much the



*Fig. 7.15. A multiple-character grid for drawing shapes that are of more than one character in size.*

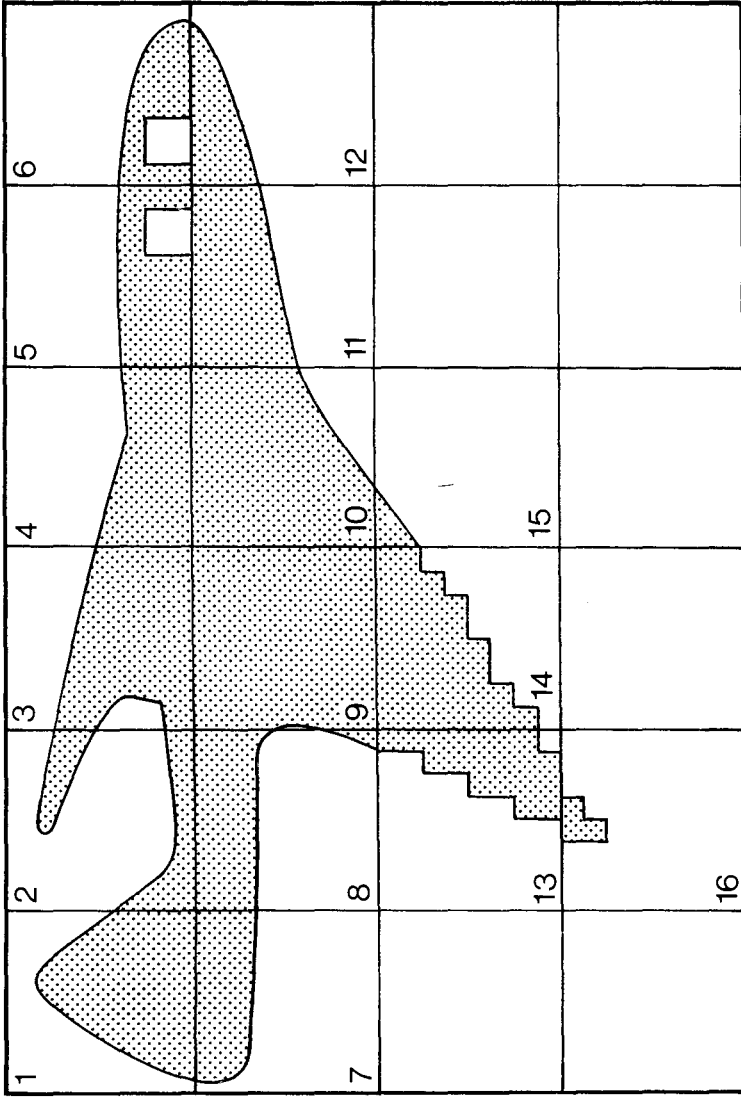


Fig. 7.16. A pattern traced on the grid. Trace the outlines only until you are satisfied, then fill in the correct shape, using blocks that correspond as closely as possible to your outline.

same way as before, but we have to make sure that we use characters that we don't need for other purposes. Looking again at the internal character set, Character 2 is the quotemark. Each time we have a PRINT instruction, we need to use the quote mark to separate what is printed from what is defined as a new character. We know that we also must avoid redefining the space character. It's safer, then, to avoid having any new shapes in the first three internal codes, 0, 1 and 3.

The going starts to get rough now. I want to illustrate the creation and use of these characters in graphics Mode 1 rather than in Mode 0 (fig. 7.17). If, however, you switch to Mode 1 at the start of the program, you will find that things don't exactly work out right. You have to carry out the preparation work in Mode 0, and then switch to Mode 1 only when you are ready to roll. Lines 10 to 80 are the

```

10 GRAPHICS 0
15 SETCOLOR 4,0,0
20 CH=(PEEK(742))*256-512:ST=CH
30 FOR J=0 TO 23:POKE CH+J,0:NEXT J
40 CH=CH+16:FOR X=1 TO 16
50 CH=CH+8
60 FOR J=0 TO 7:READ D
70 POKE CH+J,D:NEXT J
80 NEXT X
85 GRAPHICS 1
90 POKE 756,INT(ST/256)
100 PRINT #6;"#%&' ("
110 PRINT #6;")*+,-."
120 PRINT #6;" /01"
130 PRINT #6;" 2"
140 END
500 DATA 0,8,28,28,62,63,127,127
510 DATA 0,30,15,1,0,0,0,3
520 DATA 0,0,254,255,127,31,127,255
530 DATA 0,0,0,240,255,255,255,255
540 DATA 0,0,0,0,255,255,249,249
550 DATA 0,0,0,0,192,252,158,156
560 DATA 255,255,255,15,0,0,0,0
570 DATA 255,255,255,1,0,0,0,0
580 DATA 255,255,255,255,127,255,255,255
590 DATA 255,255,255,255,254,252,240,224
600 DATA 255,255,255,248,0,0,0,0
610 DATA 248,240,192,0,0,0,0,0
620 DATA 1,1,3,3,7,7,15,15
630 DATA 255,255,252,248,224,192,128,0
640 DATA 128,0,0,0,0,0,0,0
650 DATA 24,16,0,0,0,0,0,0

```

Fig. 7.17. Plotting the shape in Mode 1 is not so easy as in Mode 0.

preparation lines for this example. The first thing to notice is that less memory has to be used, because Mode 1 graphics need less. The loop in line 30 then places zero into the first three character positions (24 bytes) so that we don't have any shapes corresponding to the space, the exclamation mark, or the quote mark. The loop that starts in line 40 then changes the start-of-memory address, and pokes our data into sixteen character positions. These will have internal codes starting at 3, the # sign, and going up to 18, which is the figure 2. Only when all this has been done can we change to Mode 1 in line 85, poke the address of the new character set into 756, and then print the shapes on the screen. Since we are using Mode 1, this needs the PRINT#6 instruction. The characters that are placed between the quotes are the keyboard characters which normally use the internal codes 3 to 18.

That's it! It takes a lot of planning and a lot of programming, but the effect can be very gratifying. Now you may very reasonably complain that all the shapes we have produced have been static, not moving. It's possible to move a shape, like any of the ones we have considered, by printing it at one position, waiting, printing a blank shape of the same size, waiting and then repeating the process at another position. On the comparatively low resolution screens that we use with Modes 0, 1 and 2, though, the movement looks rather jerky. Any kind of animation looks better on a higher-resolution screen, and we'll look at that in the next chapter. In addition, you will find that the Atari makes special provision for creating movable user-defined graphics. These are often called 'sprites', but the Atari users prefer to call them 'player-missile graphics'. Their main advantage is that movement is much more easily programmed.

## Chapter Eight

# More Resolution

The graphics modes that we can obtain by using numbers 3 to 8 inclusive permit a wider choice of colours and resolution, and also the use of some new instructions. The action of the COLOR instructions is also changed when these modes are in use. There are also some modes that the Atari manual doesn't mention, and we'll take a look at these too.

The simplest way of approaching Modes 3 to 8 is by forming them into groups. Modes 3, 5 and 7 permit four colours to be displayed by simple instructions. Of these four, one is the border and background colour, whose code number is stored in register 4. This colour will be black by default - if you don't change it. Registers 0, 1 and 2 are used for foreground colours whose default values are orange, light green and dark blue respectively. As usual, the actual numbers which are stored in these registers can be altered by using the SETCOLOR instruction. The COLOR instruction then selects which of the registers will be used to provide the colour for any particular part of the screen. The instruction COLOR 0 always selects the background colour (register 4 in this case), and COLOR followed by numbers 1 to 3 will select registers whose number is one less than the COLOR number. In other words, if we use COLOR 2, we will select the colour of register 1; if we select COLOR 3, we take the colour code from register 2, and so on.

Though these three Modes all permit the same choices of colours, they differ considerably in the amount of resolution they can offer. GRAPHICS 3 allows 40 columns and 20 rows when we use the split screen, and 24 rows if we use the full height of the screen. The extended height is obtained by adding 16 to the GRAPHICS number, so that GRAPHICS19 will give the effect of GRAPHICS 3 but with all of the screen used for patterns. This is something that you have to use with some caution, because it leaves no space for messages. If there is an error message, or if the program ends with a



'READY', then the display automatically goes back to GRAPHICS 0 in order to display the printing. If you are using all of the screen, then, you should disable the cursor and prevent the READY prompt message by using an endless loop in your program.

GRAPHICS 5 uses 80 columns and 40 rows on its split screen, with 48 rows available on the full screen. This offers four times as much resolution as Modes 1 or 3. GRAPHICS 7 uses 160 columns and 80 rows (split screen) or 96 rows (full screen), which gives sixteen times as much resolution as Mode 1 or 3. These increased amounts of resolution, along with the choice of four colours, are paid for in use of memory. Mode 3 is economical, requiring less memory than Mode 0 (432 bytes for a full screen). Mode 7, at the other extreme, requires 4200 bytes of memory for a full-size screen display. The Atari automatically commandeers as much memory as it might need from the RAM. If you select GRAPHICS 7, for example, 4200 bytes of memory will be reserved. You cannot use this memory for program storage, even if nothing is placed on the screen. You can find out how much memory is available by typing:

?FRE(0)

and pressing RETURN. The quantity which appears is the amount of memory that is still available, not reserved or used in any way.

### The drawing instructions

The two main drawing instructions for the high resolution graphics are PLOT and DRAWTO. Each of these instructions has to be followed by two numbers that locate the pixel on the screen. These numbers are called *co-ordinates*, and they are our familiar column and row numbers. The column number is referred to as the X co-ordinate number, and the row number is referred to as the Y co-ordinate number. The range of values that you can use for these X and Y co-ordinates varies according to the graphics mode that you are using. In Mode 3, assuming that the screen is split, you have the use of 40 columns and 20 rows. This allows the use of numbers 0 to 39 as X co-ordinates, and 0 to 19 as Y co-ordinate numbers. As usual, X=0 means the left-hand side of the screen, and Y=0 means the top of the screen. For a full screen in Mode 3 (after using GRAPHICS 19), the range of the Y co-ordinate numbers are 0 to 23. In Mode 5 the X range is 0 to 79, and the Y range is 0 to 39 for the split screen or 0 to 47 for the full screen. In Mode 7, the X co-

ordinate range is 0 to 159 and the Y range is 0 to 79 for a split screen, or 0 to 95 for the full screen. Mode 7 corresponds to the high resolution mode of many low-cost computers which allow only one text mode and one graphics mode.

## PLOT graphics

Unlike POSITION, PLOT causes a pixel to be lit, assuming that a foreground colour is used for the pixel. The main use of PLOT by itself is to create graphs and other shapes that can be produced by the use of equations. Figure 8.1 shows an example of PLOT being

```

10 GRAPHICS 7:POSITION 0,40
20 COLOR 1:DEG
30 FOR X=5 TO 155
40 PLOT X,40+(COS(3*X)^3)*30
50 NEXT X

```

Fig. 8.1. The PLOT instruction being used to draw a graph.

used in this way, with Mode 7 graphics so as to obtain reasonably high resolution. Line 10 sets the mode and also defines the starting position. Line 20 then uses COLOR 1 to ensure that the foreground colour for the pixels will be the colour that is stored in register 0, which is orange. This is the default colour, and we can change it by use of SETCOLOR if we wish. For the moment, the default colour will do very nicely. The second part of line 20 uses DEG as an instruction which prepares the computer to work with angles. When DEG is used, the computer will take any figures for the size of angles as being in degrees. If you do not use DEG, all angles are taken as being in units of *radians*. A radian is about 57 degrees. Lines 30 to 50 then draw the graph. The quantity X is taken as the size of an angle, and what is plotted as Y is the cube of the cosine of three times X, multiplied by 30, and to which 40 has been added. The reason for adding 40 is so that the graph is centred on the screen. The cosine of an angle can take values which range from -1 to +1, and the cubes will also be within this range. Multiplying by 30 gives a range of -30 to +30, and we add 40 to make sure that the Y number is never negative. The permitted range of Y, remember, is 0 to 79, and the program will stop with an error message if you try to use any number outside this range. The reason for using 3\*X, incidentally, is so that more than one complete cycle of the wave is drawn. The range of angle for a complete cycle is 360 degrees, and our X range is only 5 to 155 in this example.

```

10 GRAPHICS 7+16:POSITION 0,40
20 DEG
30 FOR X=5 TO 155
40 COLOR 1:PLOT X,48+COS(3*X)*45
50 COLOR 2:PLOT X,48+COS(3*X)^2*45
60 COLOR 3:PLOT X,48+COS(3*X)^3*45
70 NEXT X
80 GOTO 80

```

Fig. 8.2. Drawing multiple graphs in three colours.

Figure 8.2 shows the use of PLOT for more than one graph shape. The preparatory steps in lines 10 to 30 are as before, but the PLOT steps in lines 40 to 60 will draw three graphs in different colours. This is because each PLOT has been preceded by a COLOR instruction which selects a different register. Once again, we are making use of the default values in these registers, but we could alter them by using SETCOLOR. The graphs are held on the screen at the end of the plotting by using an endless loop in line 80. This is necessary because we are using the full height of the screen, having typed GRAPHICS7+16 in line 10. Notice that we don't even have to do the addition for ourselves!

## DRAWTO in action

DRAWTO has to be followed by a pair of co-ordinate numbers in the usual X-then-Y order. The action of DRAWTO is to draw a straight line from the last point that was plotted, or drawn to, up to the new point specified by the co-ordinate numbers. For example:

```
PLOT0,0:DRAWTO 39,23
```

will draw a diagonal line from the top left-hand corner (0,0) to the bottom right-hand corner (39,23) in Mode 3, full screen. If DRAWTO 39,23 were then followed by DRAWTO 0,23, then a line would be drawn along the bottom of the screen from 39,23 to the 0,23 position.

All of that sounds quite straightforward, but there are a few pitfalls for the newcomer. One is that you can't assume that the drawing will be visible! Unless you specify a colour (using COLOR) for your drawing, you may find that it simply doesn't appear, because the default is COLOR 0, and that's background colour. The other thing you have to watch is that the starting position of a DRAWTO is not altered by the POSITION instruction, only by the use of PLOT or another DRAWTO.

```

10 GRAPHICS 7+16
20 COLOR 1
30 PLOT 30,10
40 FOR N=1 TO 20:READ X,Y
50 DRAWTO X,Y:NEXT N
60 PLOT 70,30:DRAWTO 70,3:DRAWTO 65,6:DR
AWTO 70,9
70 PLOT 40,40:DRAWTO 40,50:DRAWTO 45,50:
DRAWTO 45,40:DRAWTO 40,40
80 PLOT 95,40:DRAWTO 95,50:DRAWTO 100,50
:DRAWTO 100,40:DRAWTO 95,40
90 PLOT 65,70:DRAWTO 50,95
100 PLOT 75,70:DRAWTO 90,95
110 GOTO 110
500 DATA 30,70,65,70,65,60,75,60,75,70,1
10,70
510 DATA 110,10,105,10,105,15,100,15,100
,10,95,10,95,30
520 DATA 45,30,45,10,40,10,40,15,35,15,3
5,10,30,10

```

*Fig. 8.3.* A shape formed with the use of PLOT and DRAWTO.

To work, then, on an example, Figure 8.3 shows a shape which has been created with the use of PLOT and DRAWTO. The full Mode 7 screen is used, and the colour is specified in line 20. The main outline of the shape is created by using a loop in lines 30 and 40. This reads X and Y numbers from DATA lines, and uses them in the DRAWTO instruction. The fine details are then provided by lines 60 to 100, and the pattern is maintained on the screen by the closed loop in line 110. Type it in and try it out. If you haven't done anything like this before, the main astonishment is how quickly the pattern is drawn!

Now before we launch into greater things, you should know how a pattern like this is planned. Figure 8.4 shows what is involved. I used graph paper which was scaled in 2cm and 2mm squares. I numbered these in tens along the top and one side, placing the numbers on the lines instead of on the spaces, and ignoring the 0 starting positions. This way, it's easier to make use of the graph paper. The pattern is drawn out, once again following the lines rather than trying to shade in the spaces, and the X,Y numbers at each change of direction are noted. I chose a starting position – it could have been any point, but the top left-hand corner of 30,10 was convenient. From that, I got the PLOT in line 30, and the DATA numbers were derived from the other points in the main shape. It's usually a good idea to put a small cross at each point that you intend to use as DATA, and that's what I did. From then on, it's plain sailing ... or castlebuilding!

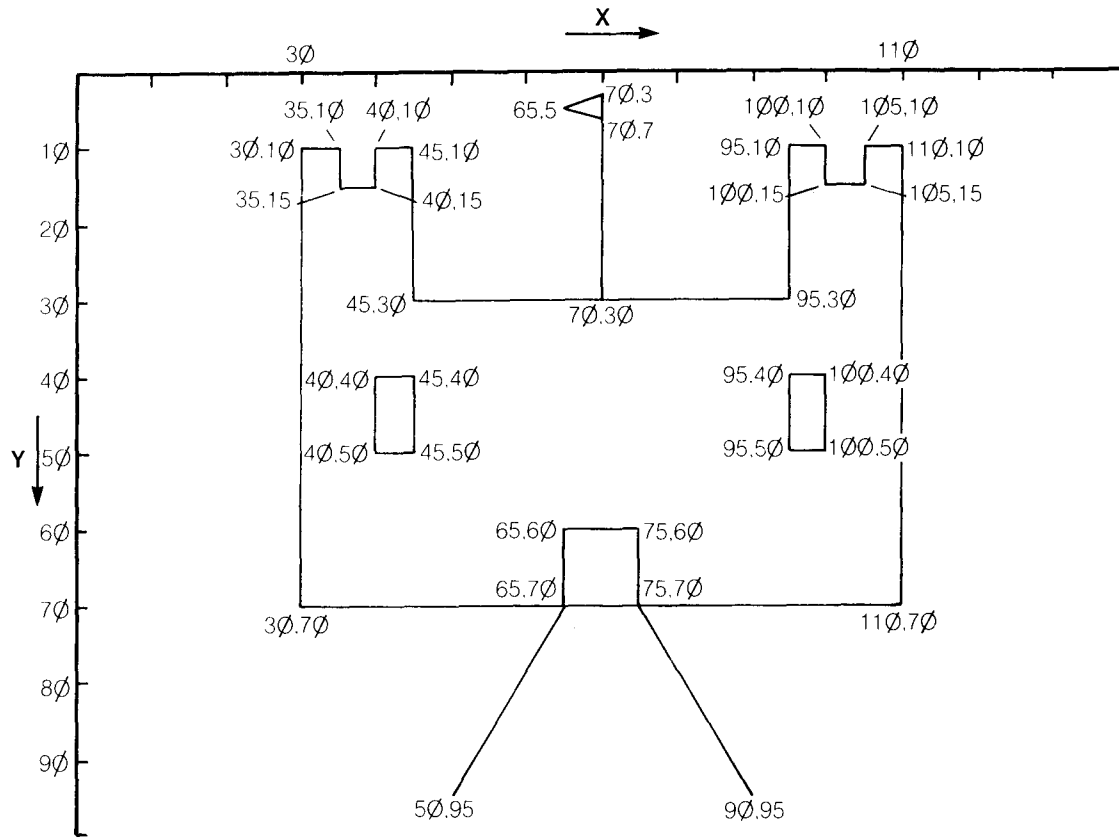


Fig. 8 4. Planning the pattern for the program.

**Circling around the point**

One noticeable lack in the Atari graphics instructions is a CIRCLE instruction. This is a pity, because circles are a useful part of your drawing kit. Using BASIC, we can create circles, but the drawing is slow. Figure 8.5 shows two methods which differ in appearance and speed. The first method makes an approximation to the shape of a circle by using straight lines. This is done by DRAWTO. The drawing can be considerably speeded up by using a larger STEP size in the loop of line 40, but the shape is then less circular. The second method uses plotting, but does not produce such a good appearance unless you are looking for dotted circles. Take your pick!

```

10 GRAPHICS 7
20 X=80:Y=40:R=15
30 PLOT X,Y+R:DEG
40 FOR N=0 TO 360
50 DRAWTO X+R*SIN(N),Y+R*COS(N)
60 NEXT N
70 PRINT "FIRST METHOD"
80 FOR J=1 TO 5000:NEXT J
85 GRAPHICS 7
90 PLOT X-R,Y
100 FOR J=(X-R) TO (X+R)
110 P=SQR(R^2-(J-X)^2)
120 PLOT J,Y+P:PLOT J,Y-P
130 NEXT J
140 PRINT "SECOND METHOD"

```

*Fig. 8.5.* Two circle-drawing routines.

```

10 GRAPHICS 7+16
20 COLOR 1
30 GOSUB 1000
40 COLOR 0
50 GOSUB 1000
60 GOTO 20
1000 PLOT 80,10:DRAWTO 70,40:DRAWTO 80,7
0:DRAWTO 90,40:DRAWTO 80,10
1010 DRAWTO 75,35:DRAWTO 80,70:DRAWTO 85
,35:DRAWTO 80,10
1020 PLOT 70,40:DRAWTO 75,35:DRAWTO 85,3
5:DRAWTO 90,40
1030 FOR J=1 TO 100:NEXT J:RETURN

```

*Fig. 8.6.* Drawing alternately in foreground and in background colour to give a twinkling appearance.

Now for something more ambitious, a touch of twinkle, twinkle little star. Figure 8.6 is a program which illustrates the use of COLOR in drawings. The drawing, of a double diamond pattern, is carried out by a subroutine which starts at line 1000. The closed loop between lines 20 and 70 alternately draws this pattern in COLOR 1, which makes the pattern visible, and in COLOR 0, which makes the pattern invisible. There is no delay loop, so the speed at which you see the pattern appear and disappear shows the speed of the DRAWTO instructions. A curious point about this is that it seems to present a 3-dimensional appearance. On my screen at least, viewing the pattern from different angles made it look as if it was quite solid.

### Painting the picture

One of the glories of the Atari graphics instructions is a very odd-looking one, XIO. Most of its uses are for purposes that most of us might never think of, but it has one application that is of very great interest. That is to fill in a shape with colour! The rules for this action are rather precise, and you will get odd results if you deviate from them. Nevertheless, the action is very useful and surprisingly fast.

The rules are as follows:

1. PLOT a point at the lower right-hand edge of the space.
2. Use DRAWTO to make a line to the top right-hand corner.
3. Use DRAWTO to make a line across to the top left-hand corner.
4. Use POSITION to place the cursor at the bottom left-hand corner.
5. POKE765,n:XIO 18,#6,0,0,"S: "

In this last command line, the 'n' is the same colour number that has been used for drawing the lines in the first steps.

The XIO sequence will work perfectly only if you have gone through these steps correctly. It is designed to be used with four-sided shapes but, if you make two of the points very close together, the shape is as near as makes no difference a triangle. With a lot of effort, it is possible to fill practically any shape with colour. Figure 8.7 demonstrates the stunning effect of watching XIO in action. Mode 7 is used, and the SETCOLOR in line 20 changes the colour of the zero register to a high luminance light blue. The rectangle that is defined by the instructions in lines 30 and 40 is then filled by line 50, using a 1 poked to 765. Remember that a 1 here is like a COLOR

```

10 GRAPHICS 7
20 SETCOLOR 0,9,12
30 PLOT 159,20:DRAWTO 159,0
40 DRAWTO 0,0:POSITION 0,20
50 POKE 765,1:XIO 18,#6,0,0,"S:"
60 PLOT 159,79:DRAWTO 159,20
70 DRAWTO 0,20:POSITION 0,79
80 POKE 765,2:XIO 18,#6,0,0,"S:"
90 A=0.5:X1=60:X2=70:COLOR 3
100 FOR Y=20 TO 79
110 PLOT INT(X1-A),Y:DRAWTO INT(X2+A),Y
120 A=A+0.5
130 NEXT Y

```

*Fig. 8.7.* Using the XIO instruction to fill a shape with colour.

1; it uses the colour of register zone. The next area is then defined by lines 60 and 70, and filled by line 80 with COLOR 2, a light green. The awkward part is then performed by a more conventional action. Lines 90 to 130 draw a set of coloured lines in dark blue across the screen to fill in a shape which couldn't so easily be dealt with by XIO. It's impressive - watch it!

## Two-colour modes

Modes 4 and 6 operate with two colours at a time, using register 4 for the background and the border, and register 0 for the foreground. As usual, the actual colour numbers in the registers can be changed by using SETCOLOR, and the colours that are selected are put in place by using COLOR 0 for background (remember that COLOR 0 is always background) and COLOR 1 for foreground. Mode 4 allows you the use of 80 columns and 40 rows (split) or 48 rows (full screen). Mode 6 allows 160 columns by 80 (split) or 96 (full) rows. These modes are used when the same resolution as modes 5 or 7 respectively is needed, but where only one foreground and one background colour need be used. The advantage is that less memory is needed as compared to Modes 5 or 7. Mode 4 needs 696 bytes for full-screen display, as compared to the 1176 bytes that Mode 5, with the same resolution but four colours, needs. Mode 6 needs 2184 bytes for the full screen.

To see these modes in action, take a look at the program in Fig. 8.8. This indicates the difference in the resolution of these two modes by drawing a cross in each mode. The drawing action is carried out by the subroutine which starts at line 10000. This incorporates a



```

10 GRAPHICS 4:COLOR 1
20 GOSUB 1000
30 PRINT "MODE 4"
40 FOR J=1 TO 3000:NEXT J
50 GRAPHICS 6
60 GOSUB 1000
70 PRINT "MODE 6"
80 END
1000 PLOT 0,0:X=79:Y=39
1010 IF PEEK(87)=6 THEN X=159:Y=79
1020 DRAWTO X,Y:PLOT 0,Y
1030 DRAWTO X,0:RETURN

```

*Fig. 8.8.* Comparing the resolutions of Mode 4 and Mode 6. A diagonal line in a lower resolution mode looks 'stepped'.

rather useful feature – how to find which mode you are using. Line 1000 places the cursor at the top left-hand corner, and then sets values for X and Y which are the maximum possible values. Now the values in line 1000 are the values for Mode 4, and we need a different set if we want to cover the whole screen in Mode 6. Line 1010 therefore tests for Mode 6. This is done using PEEK(87). The result of this PEEK is a number which is the mode number. If the subroutine has been called while Mode 6 is being used, then larger values of X and Y are set before the DRAWTO and PLOT steps of line 1020 and 1030.

## Mastering Mode 8

Mode 8 is the mode which is used for the highest resolution graphics. Only one colour can be used in this mode, with two levels of luminance, and control is exerted through registers 1,2 and 4. Register 1 is used for the foreground, register 2 for background, and register 4 for the border. COLOR 1 will select the foreground colour for drawing, and COLOR 0 will, as usual, select background colour. The resolution is 320 columns by 160 rows (split), or 192 rows (full screen), and the memory that is needed is 8112 bytes for the split screen or 8138 for the full screen display.

Mode 8 differs from the other graphics modes in having no way of selecting foreground colour separate from background colour. Though register 1 controls foreground, the colour of the foreground is always the same as that of the background, and only the luminance has any effect. The colour that appears for lines drawn in this mode therefore depends on what colour you choose for the

```

10 GRAPHICS 8
20 Z=319:W=159:GOSUB 1000
30 FOR Q=0 TO 15
40 SETCOLOR 2,Q,2
50 FOR J=1 TO 500:NEXT J
60 NEXT Q
70 FOR Q=0 TO 14 STEP 2
80 SETCOLOR 2,2,Q
90 NEXT Q:GOTO 70
1000 FOR Y=0 TO W STEP 3
1010 PLOT 0,0:DRAWTO Z,Y
1020 NEXT Y
1030 RETURN

```

*Fig. 8.9.* Fan patterns drawn in Mode 8.

background. The high resolution of this mode is nicely demonstrated by drawing fan patterns, and Fig. 8.9 illustrates the appearance of fine patterns drawn in Mode 8. The pattern is drawn by the subroutine in lines 1000 to 1030, and the various combinations of background colour and foreground luminance are tried. Lines 30 to 60 alter the colour of register 2, so controlling the background colour. Lines 70 to 90 then show the effect of altering the luminance of the background. Once you have tried this, alter the register number in line 80 to 1, and explore the effect of different foreground luminance values.

### The other graphics modes

Though the Atari manual mentions only Modes 0 to 8, you will find

```

10 GRAPHICS 9:Z=79:W=159
15 COLOR 7
20 GOSUB 1000
30 FOR Q=0 TO 15
40 SETCOLOR 4,Q,Q
50 FOR J=1 TO 500:NEXT J
60 NEXT Q
70 SETCOLOR 4,2,0
80 FOR C=0 TO 16:COLOR C
90 GOSUB 1000
100 FOR J=1 TO 100:NEXT J
110 COLOR 0:GOSUB 1000
120 NEXT C
130 END
1000 FOR Y=0 TO W STEP 3
1010 PLOT 0,0:DRAWTO Z,Y
1020 NEXT Y
1030 RETURN

```

*Fig. 8.10.* Using Mode 9, which is not listed in the Manual.

that typing higher numbers will get you into graphics modes that are quite different from the lower-numbered ones. As an example, try GRAPHICS 9, illustrated in Fig. 8.10. This is a one-colour mode, which allows register 4 to control the colour and the COLOR instruction to affect the brightness. The resolution is 80 columns  $\times$  160 rows, unlike any other modes. In the example, the subroutine in line 1000 plots the pattern which will be familiar by now, and using the brightness that is set by COLOR 7. The loop in lines 30 to 60 then runs through the range of colours that can be obtained. Lines 70 to 100 then use a fixed setting of SETCOLOR, and explore the range of COLOR which for this mode can be 0 to 16! The use of COLOR 0 in line 110 followed by the GOSUB 1000 then blanks out the pattern. It's different!

There's more, though. Figure 8.11 demonstrates Mode 10, which is a multi-colour mode. This again has resolution of 80  $\times$  160, and it uses register 3 to control the appearance of the first set of patterns, and COLOR to control the second set. The useful range for COLOR is 4 to 7 inclusive – other numbers will give the same colour range. Figure 8.12 demonstrates Mode 11 in use. This is controlled by register 4 and by the COLOR instruction once again, and offers a wide range of colours, but with only two tones on the screen at one time.

```

10 GRAPHICS 10:Z=79:W=159
15 COLOR 7
20 GOSUB 1000
30 FOR Q=0 TO 15
40 SETCOLOR 0,Q,0
50 FOR J=1 TO 500:NEXT J
60 NEXT Q
70 SETCOLOR 3,2,7
80 FOR C=4 TO 7:COLOR C
90 GOSUB 1000
100 FOR J=1 TO 100:NEXT J
110 COLOR 0:GOSUB 1000
120 NEXT C
130 END
1000 FOR Y=0 TO W STEP 3
1010 PLOT 0,0:DRAWTO Z,Y
1020 NEXT Y
1030 RETURN

```

Fig. 8.11. Mode 10 in action.

```

10 GRAPHICS 11:Z=79:W=159
15 COLOR 7
20 GOSUB 1000
30 FOR Q=0 TO 15
40 SETCOLOR 4,Q,0
50 FOR J=1 TO 500:NEXT J
60 NEXT Q
70 SETCOLOR 4,2,7
80 FOR C=0 TO 16:COLOR C
90 GOSUB 1000
100 FOR J=1 TO 100:NEXT J
110 COLOR 0:GOSUB 1000
120 NEXT C
130 END
1000 FOR Y=0 TO W STEP 3
1010 PLOT 0,0:DRAWTO Z,Y
1020 NEXT Y
1030 RETURN

```

*Fig. 8.12.* Mode 11 also exists and is demonstrated here.

### Locating the plot

A lot of games programs need to be able to detect when an object strikes another one, and one of the ways in which this can be done is the LOCATE instruction. LOCATE has to be followed by three variables. Of these, the first two are the familiar X and Y co-ordinate numbers, but the third is a variable whose value will be assigned by the LOCATE instruction. Suppose, for example, that you had LOCATE 20,10,Q. The value that variable Q has, after this instruction has been carried out, will depend on what was at position 20,10. Exactly what it is depends on which graphics mode you are using. If you are using Mode 0, then Q will take the value of the

```

10 GRAPHICS 7:X=80:Y=40:COLOR 1
20 PLOT 5,0:DRAWTO 5,79
30 PLOT 155,0:DRAWTO 155,79
40 POSITION X,Y:K=1
50 GOSUB 1000
60 LOCATE X+K,Y,Q
70 IF Q=1 THEN K=-K
80 GOTO 50
1000 COLOR 2:PLOT X,Y
1010 FOR J=1 TO 50:NEXT J
1020 COLOR 0:PLOT X,Y
1030 X=X+K:RETURN

```

*Fig. 8.13.* Animating a dot, using alternate background and foreground colours along with LOCATE.

ASCII code for whatever character is at position  $2\emptyset, 1\emptyset$ . In Modes 1 and 2, the number will give the colour and character, using the codes shown in the manual. In the high resolution graphics modes, the value of Q will show which colour register is in use at the chosen position.

Figure 8.13 shows a simple program which draws a pair of vertical walls using COLOR 1. The subroutine plots a point in colour 2, then 'unplots' it by using COLOR  $\emptyset$ , and shifts the X value. The LOCATE test in line  $6\emptyset$  then discovers what register is in use at this new position, and line  $7\emptyset$  tests this value. If a wall has been reached, then the X values are reversed, and X becomes smaller each time K is added. This causes the dot to carry out its Harvey Wallbanger action, bouncing from one side to the other.

### Sprites, or players and missiles

Several modern computers feature what are termed 'sprite graphics'. A *sprite* is a user-defined character which can be moved around the screen by the use of comparatively simple instructions. These instructions obviate the need to draw the shape, undraw it, shift, draw again and so on. Though sprite graphics are not mentioned in the Atari manual, the Atari nevertheless possesses a very effective sprite graphics system. The system is by no means elementary to use, but the introduction to user-defined graphics in Chapter 7 should have broken you in to the ideas that are needed for the creation of characters at least.

There are two types of sprites available on the Atari, called 'players' and 'missiles' respectively. The width of a player can be up to eight pixels, but its depth can be up to 256. These dimensions are constant, no matter which graphics mode we are using. This allows a player to take up the whole height of the screen if we want it to, but its width is limited to the eight pixels, one column of a 40-column screen. Missiles are only two pixels wide, and would normally use a comparatively small depth, perhaps 8 pixels or so. The full screen depth can be used, however, if you wish.

The particular advantage of these players and missiles is that they can be moved around the screen so easily. They do not require the character set to be changed, unlike user-defined characters and, most remarkably, they use their own colour registers. Each sprite object has its own colour register allocated to it, up to a maximum of four players and four missiles. The use of these extra colour registers

can permit up to four extra colours to appear on the screen. In this way, by using Mode 1, which is normally a 5-colour mode, it would be possible to display nine colours on the screen at one time by using all four players (or missiles).

The problem about player-missile graphics, however, is that there are no BASIC instructions for creating or using these objects. We have to carry out all of these graphics operations by using POKE instructions to place code numbers direct into memory, and this is never quite so simple or straightforward. One minor consolation is that the Atari system is no more complicated than any others, and easier than some!

### Creating sprites

As usual, we have to start by drawing out the shape or shapes that we want. We have to use a planning chart which is eight blocks wide and can be up to 256 blocks deep. In practice, we very seldom need so much depth except when we are creating fancy borders on the screen, so an  $8 \times 80$  block as shown in Fig. 8.14, is usually enough.

The rules for creating a player pattern on this grid now resemble very closely the rules for creating a user-defined character. The squares which we shade in represent the pixels that will be lit, and each line of eight pixels across can be represented by a code number. The number is obtained, as before, by adding up the column numbers for the shaded portions only. One important difference, however, is that the position of the pattern vertically on an  $8 \times 256$  grid will decide the vertical position of the player. If you make a

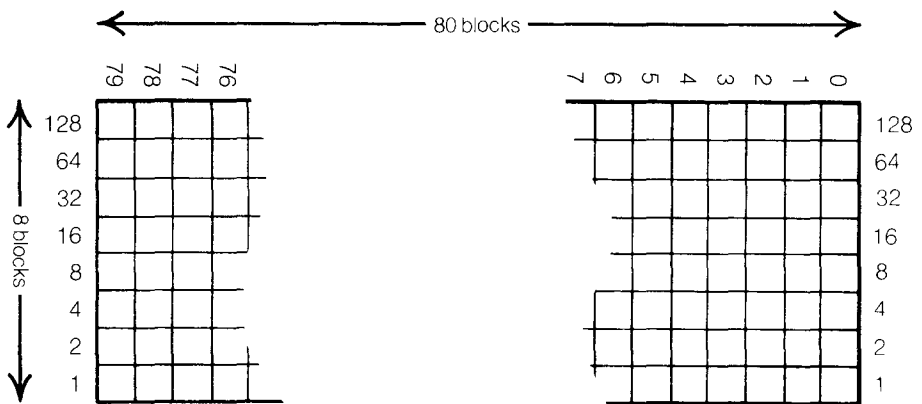


Fig. 8.14. An 8 by 80 block for creating sprite patterns.

player which is 6 pixels deep, and these are the first six pixels in a 256 block, then the player will appear at the top of the screen. If you use the last six pixels in the 256 deep block, the player will appear at the bottom of the screen. For missile creation, you need to use only two pixels' width, so that your artistic creativity is rather more restricted.

The next problem is that of storing the numbers. Strict rules have to be applied here, because the computer must be able to make use of the codes that you enter, whether the player is of full height or not. The first thing that you have to decide on is whether you want to use fine or coarse resolution for your sprite. Fine resolution sprites require 256 bytes of memory to be set aside for each player, whether the player is full height or not. Coarse resolution sprites need only 128 bytes each. The difference may not be important, depending on the appearance that you want, but the amount of memory that has to be set aside may be important. If we set our player codes starting at intervals of 128 bytes, then the computer will display them as coarse resolution. If we set them at 256 byte intervals, then the computer will use fine resolution.

## Placing the codes

We can place the codes for the sprites in the memory by using POKE instructions, but there are restrictions on the starting address. If we are using coarse resolution sprites, we have to use a starting address that divides exactly by 1024. If we are using fine resolution sprites, the starting address must be one that divides exactly by 2048. In addition, the amount of memory that is needed will be either 1024 bytes (for coarse) or 2048 bytes (for fine). This is true whether you use all of this memory or not. Even if each player uses only eight bytes, the full allocation of memory must be available so that the player can move vertically, and so that the set of numbers can start at the correct address.

The allocation of memory for coarse sprites is illustrated in Fig. 8.15. If we number the first address in this part of memory as  $\emptyset$ , then the bytes from  $\emptyset$  to 384 must not be used. This is because the machine has to make use of them for other purposes. The addresses between 384 and 512 are reserved for missiles, with up to four missiles stored. A missile consists of only two pixels of width, so we can pack four missiles into a block of eight pixels across, and this is how missile information is stored. In this way, the data on four missiles can take up the same memory as one player (Fig. 8.16).

Coarse resolution	Player No.	Fine resolution
X+512	0	X+1024
X+640	1	X+1280
X+768	2	X+1536
X+896	3	X+1792
X+1024	End of space	X+2048

*Fig. 8.15.* Allocation of memory for sprites. The number X is the address for the start of the storage space.

Coarse resolution		Fine resolution
X+384	Start of space	X+768
X+511	End of space	X+1023

*Fig. 8.16.* Allocation of memory for missiles. The number X is the address for the start of the storage space.

The first memory space that can be used for a player starts at 512, and subsequent players start at 640, 768 and 896. For fine resolution, these numbers are 768 for missiles, and 1024, 1280, 1536 and 1792 for players. These numbers, remember, are added to the memory address of the start of the reserved part of the memory.

We can use any part of the available memory for storing these numbers, but it makes sense to use a part of memory which can't be allocated to anything else. The snag is that the machine allocates its memory differently when you change graphics modes, and this can result in your numbers being wiped from the memory. The end of memory is convenient for many purposes when no fancy effects are being used, so we'll stick to this simple method.

The highest memory address that you can use is given by the instruction `PEEK(196)*256`. If you want to use coarse resolution sprites, you must subtract 1024 from this number to obtain the starting address for your sprite graphics numbers. For fine resolution sprites, the number to subtract is 2048. The result of this subtraction is the start of the block of memory at which we can store our sprite data. You can protect this address by preventing the



computer from using these addresses. This is done by dividing the starting address by 256 and poking the result back into location 106. The computer cannot use any address higher than this value, so preserving your graphics code numbers. For example, if `PEEK(106)*256` gives 40960, then the start address for coarse resolution sprites will be  $40960 - 1024$ , which is 39936. This, divided by 256, is 156, so that `POKE 106, 156` will ensure that the data is safe from the normal action of the computer.

You can't, however, assume that the piece of memory that you have roped off in this way will be clear, with each byte storing a zero. The computer may have stored some bytes in this part of memory during earlier parts of the program, and these bytes will affect the appearance of your graphics if you allow them to stay in place. The next step must therefore be to clean up this piece of memory for each player or missile that you are going to use. Since each player or group of missiles uses 128 bytes (coarse resolution), all 128 must be cleared. This can be done by using a `FOR...NEXT` loop to poke 0 into each memory address.

```

10 GRAPHICS 5
20 REM
30 X=PEEK(106)-4:POKE 106,X/256
40 ST=X*256+512
50 FOR J=ST TO ST+127
60 POKE J,0:NEXT J

```

Fig. 8.17. A fragment of program showing how memory is allocated and cleared.

Figure 8.17 shows where we have got to so far – it's not a working program. By using `X=PEEK(106)-4` in line 30, we set X to a number which when multiplied by 256 will give an address that is 1024 bytes below the end of usable memory. This is because  $4*256=1024$ . We never need to clear the unused piece of this memory, from the start to 384 for coarse players, to 768 for fine. If we are not using missiles, then we don't have to clear the missile memory region. We can start clearing at address ST, which is  $X*256+512$ , as in line 40. This is 512 bytes from the start of the reserved piece of memory. We clear the memory for players by using a loop in lines 50 and 60, which pokes a 0 into each address that will be used by one low resolution player. If we are using only one player, we don't need to clear any more of the memory. One thing to watch here is for repetition. Each time you repeat the steps in lines 10 to 60 of Fig. 8.17, you will rope off more memory, until there is none left to use! If you are likely to be running this piece of program several

times, then you should either poke back the original value into address 106 at the end of the program, or use the SYSTEM RESET key to put all these values back to normal.

The range of memory that has been cleared in this way represents the range of vertical positions for the player. If you want your player to appear first at the top of the screen, then you will poke the player data bytes at the start of the cleared section of memory. If you want the player to appear at the bottom of the screen, then you will poke the player data bytes near the end of the cleared section. By shifting all the data bytes together in the memory, you can make the player image move vertically – but more of that later!

To make the image of the player appear, we have to place the data bytes into the memory, and then carry out some poking into registers that control this image. Placing the data bytes into memory is simple enough, using READ and POKE, and we won't dwell on it. The only point to note is that you might want to start poking the bytes at an address such as ST+60 rather than at ST, if you want the image to appear about halfway down the screen.

This is where the tricky bits start. If the computer is to control the antics of the player that you have created, it has to keep track of where the data is stored. The first part of this is to store the address X, which is the start of the whole block of reserved memory, into address 54279. This allows the computer to find your data. You then have to poke two 'enabling' addresses. The first of these enabling addresses is 53277. This normally contains 0, and a zero value does not permit any sprites to appear. Poking 2 into this address enables players, poking 1 enables missiles, and poking 3 enables both. The second enabling address is 559. This normally contains 0, disabling sprites. To enable players, poke 8, to enable missiles, poke 4, and to enable both, poke 12 into this address. If you want to see fine resolution sprites, you have to add 16 to each figure.

POKE No.	Effect
0 or 2	Normal width
1	Twice normal width
3	Four times normal width

*Fig. 8.18.* Poking different missile widths. POKE address 53260 controls the width of all four missiles – you cannot select the width of any one missile independently.

These pokes allow the computer to locate a player or missile set of data, and allow the result to be displayed. We now have to look at how the size of sprites, their location and their colour can be controlled. This, as you might expect, is also done by poking values into register addresses. The width of both player and missile images on the screen can be controlled. The normal width of a player is eight pixels, and a missile is two pixels. We can, however, double or quadruple the size of both players and missiles on the screen. The width of players is governed by addresses 53256 to 53259, with 53256 controlling the first player. If you don't intend to change the width of a player, these registers can be left alone – the normal value is  $\emptyset$ . A value of 1 will set double width, a value of 3 will set quadruple width, but a value of 2 resets to normal again. The width of all missiles is dealt with by the one address 5326 $\emptyset$ , and Fig. 8.18 shows how this address has to be poked to affect the different missiles. If you use one missile only, then the numbers are the same as for a player,  $\emptyset$ , 1 or 3.

---

Start No.	Colour
$\emptyset$	Black (white at max. luminance)
16	Gold
32	Orange
48	Red
64	Pink
8 $\emptyset$	Violet
96	Purple
112	Blue
128	Blue
144	Pale blue
16 $\emptyset$	Blue-green
176	Blue-green
192	Green
2 $\emptyset$ 8	Dark yellow
224	Khaki
24 $\emptyset$	Pale orange

---

These are described as 'start numbers' because a luminance number ranging from  $\emptyset$  to 14 can be added. Even luminance numbers should be used.  $\emptyset$  gives no luminance; 14 gives maximum luminance.

Fig. 8.19. Colour and luminance values for sprites.

- 
1. Decide on coarse or fine resolution.
  2. Find starting address for POKEing values.
  3. Write down starting addresses for each player and missile.
  4. Write program lines to reserve memory and clear memory.
  5. POKE number data for player/missile into memory.
  6. POKE start address into 54279.
  7. POKE 53277 to enable players, missiles or both.
  8. POKE 559 to enable players, missiles or both. Remember to add 16 to the number if fine-resolution is being used.
  9. POKE start positions for each player and missile.
  10. POKE colours for each player and missile.

After this, POKE vertical and horizontal position as needed, along with width and colour.

---

*Fig. 8.20.* A summary of the steps that are needed for creating sprite graphics.

The position of a player or missile in the vertical direction is controlled by the position of the data in the 128 bytes (for coarse resolution) that are allocated. Horizontal movement is controlled by another set of addresses. For players, these are 53248 (first player) to 53251 (fourth player); for missiles, they are 53252 (first) to 53255 (fourth). A value of 0 poked into one of these addresses will place the player or missile at the extreme left-hand side of the screen, and 227 will place the object at the extreme right-hand side. Depending on the width of the object, you may have to use numbers greater than 0 for the left-hand side, and less than 227 for the right-hand side if the whole of the object is to be visible.

All that is left now is to decide the colour of players and missiles. A player and its missile of the same number (1,2,3 or 4) will have the same colour. The addresses which are used range from 704 (first) to 707 (fourth). Both the colour and the luminance can be set by a number which is poked into the appropriate address. The colour/luminance numbers are shown in Fig. 8.19. These are the zero-luminance numbers, and you can add a luminance number of 0 to 14 (even numbers only) to any of these colour numbers. In this way, 192 will set the colour to green, and adding 8 will set the brightness to about half of maximum.

That's it! Figure 8.20 summarises the steps that are needed for the full specification of a single player. There's quite a lot of work here but, as you will see, it can be very rewarding. The next step is to look at an example of a player in use.

```

10 GRAPHICS 4
20 SETCOLOR 0,12,4:SETCOLOR 4,8,4
30 X=PEEK(106)-4:POKE 106,X
40 ST=256*X+512
50 FOR J=ST TO ST+127
60 POKE J,0:NEXT J
70 FOR J=ST+60 TO ST+71
80 READ D:POKE J,D:NEXT J
90 POKE 54279,X:REM START OF TABLE
100 POKE 53277,2:REM PLAYER ONLY
110 POKE 559,8:REM PLAYER 1
120 POKE 53248,0:REM START POSITION
130 POKE 705,56:REM RED, LUM 8
140 GOSUB 1000
150 FOR J=0 TO 115
155 FOR Z=1 TO 20:NEXT Z
160 POKE 53248,J:NEXT J:REM MOVE HORIZON
TALLY
170 GOSUB 1000
180 POKE 53256,1:REM DOUBLE WIDTH
190 GOSUB 1000
200 POKE 53256,3:REM QUAD
210 GOSUB 1000
220 POKE 53256,0:REM NORMAL
230 GOSUB 1000
240 FOR P=ST+60 TO ST+30 STEP -1
250 FOR J=0 TO 10
260 POKE P+J-1,PEEK(P+J)
270 NEXT J:POKE P+J,0
280 NEXT P
290 POKE 704,132:REM DISAPPEAR
300 POKE 54279,255:POKE 53277,15
310 POKE 559,34:POKE 53248,0
320 POKE 704,0:POKE 106,160
330 GRAPHICS 0:END :REM RESTORE NORMALIT
Y.
500 DATA 60,60,145,74,145,74,145,74,145,
74,145,74
1000 FOR Z=1 TO 500:NEXT Z
1010 RETURN

```

Fig. 8.21. A program which illustrates the creation, movement, width and colour control of a player.

Figure 8.21 contains the program which illustrates the creation of a player, and of movement and expansion. Mode 4 is used, in which points that are plotted on the screen will use register 0, so that line 20 sets the contents of this register. The background is set to mid-blue by putting colour 8 into register 4. Since we do not have any COLOR or PLOT instructions in the program, there will be no green lines drawn or areas coloured.

The steps in lines 40 to 130 have already been described in detail.

Line 30 finds a starting address X for the player data, using 1024 bytes, and protects this address. Line 40 locates the starting address for the first player, the only one that is used in this example. Lines 50 and 60 then clear this section of the memory, and lines 70 and 80 put 2 bytes of data into the memory – a jellyfish shape. The data for the READ step is in line 500. Following this, a series of pokes sets up the controls. Line 90 places the start-of-data address into 54279, and lines 100 and 110 enable player action. The starting position at the left-hand side of the screen is determined in line 120, and the colour in line 130. From this point on, all that we do is to manipulate the position, width and colour of the player. So that you can see each action, one at a time, a delay subroutine has been programmed in line 1000, and is called between each pair of steps.

Lines 150 to 160 move the jellyfish horizontally. This movement is rather fast as jellyfish go, and it has to be slowed down by a delay routine in line 155. You can control the speed of the horizontal movement by altering the size of the count in the delay loop. Lines 180 to 220 then demonstrate the effect of altering width, returning to normal in line 220. The routine that starts at line 240 then moves the jellyfish vertically. This is done by shifting each byte of data one address number lower in its block of memory, and poking a zero into the highest address (the last one used) each time. This is not a particularly fast method of achieving vertical movement, but it's good enough for a jellyfish. For faster movement, you may have to resort to machine code rather than BASIC. Lines 300 to 330 are then used to restore normal operating conditions by putting the normal values back in the registers.

## Players and priorities

Finally, Fig. 8.22 shows a combination of ordinary graphics and players, along with some more advanced techniques. Lines 10 to 40 carry out a comparatively straightforward piece of drawing, with a coloured column running down the centre of the screen. Some Atari users refer to a piece of graphics like this as a 'playfield'. Line 50 chooses a starting address for the player data – this is considerably lower than we have used up to now. This lower starting address is necessary, because the techniques that we shall use simply don't work if higher addresses are used. Lines 70 to 80 clear the portion of memory for two players, and the player code numbers are poked in place in lines 90 to 130. There's nothing particularly impressive

```

10 GRAPHICS 7+16
20 COLOR 1:PLOT 90,95:DRAWTO 90,0
30 DRAWTO 70,0:POSITION 70,95
40 POKE 765,1:XIO 18,#6,0,0,"S:"
50 X=PEEK(106)-24
60 ST=256*X+512
70 FOR J=ST TO ST+512
80 POKE J,0:NEXT J
90 FOR J=ST+63 TO ST+68:READ D
100 POKE J,D
110 NEXT J
120 FOR J=ST+317 TO ST+324
130 READ D:POKE J,D:NEXT J
140 POKE 54279,X
150 POKE 53277,2
160 POKE 559,42:REM 34+8 FOR BOTH
170 POKE 53248,0:POKE 53250,159:REM POS
TION
180 POKE 704,50:POKE 706,202:REM COLOUR
190 POKE 623,1
200 GOSUB 1000:REM MOVE
210 POKE 623,2
220 GOSUB 2000:REM RETURN
230 POKE 623,4
240 GOSUB 1000
250 POKE 623,8
260 GOSUB 2000
270 FOR Z=1 TO 500:NEXT Z
280 POKE 54279,255:POKE 53277,15
290 POKE 559,34:POKE 53248,0
300 POKE 704,0:POKE 106,160
310 GRAPHICS 0:END
500 DATA 16,16,56,124,254,255
510 DATA 129,66,34,18,10,6,126,128
1000 FOR J=1 TO 227
1010 POKE 53248,J:POKE 53250,228-J
1020 FOR Z=1 TO 30:NEXT Z
1030 NEXT J:RETURN
2000 FOR J=227 TO 1 STEP -1
2010 POKE 53248,J:POKE 53250,228-J
2020 FOR Z=1 TO 30:NEXT Z
2030 NEXT J:RETURN

```

*Fig. 8.22.* Illustrating player priority selection. Priority can be allocated with reference to scenery or other players.

about the shapes – I simply want two shapes that are distinctly different. One of the players is a Player 1, the other is a Player 3. I have done this in preference to a 1 and a 2 so as to illustrate some of the remarkable effects that the Atari can achieve when these two (or a 2 and a 4) are used.

The pokes in lines 140 to 190 then plan out how the players will

appear. One important difference here is the number that is poked into 559. We would normally poke 8 into this address to enable coarse resolution players, and 24 to enable fine resolution players. When we want to display a background of normal graphics as well, however, different numbers have to be used. The rules here are not so clear, but I found that using the number 34 added to the normal player number gave the results I wanted. In this case, to enable players means adding 8, so that the number to be poked is  $34+8=42$ . The other pokes in these lines should be clear enough, remembering that we are using two players.

Now for the novelty. The Atari provides for what is called 'priority'. Priority comes into effect when a sprite moves across graphics, or when one sprite crosses the path of another. By assigning priority numbers, we can make sprites appear to pass in front of non-moving graphics shapes (playfield), or behind these shapes. We can also make sprites appear to move so that one will always appear to pass in front of another, or so that one colour will always appear to be in front of another. This takes us perilously close to really advanced programming, much more than should be in a beginners' book, but it's such an interesting and little-known aspect of the Atari that it's worth a mention.

The register address that decides priority is 623, and the effect of poking different numbers into this register is shown in Fig. 8.23. Just

---

POKE instructions into address 623

---

1. All players have priority over normal graphics.
  2. Players 0 and 1 have priority over other players and normal graphics.
  4. Normal graphics have priority over all players.
  8. Graphics using colour registers 0 and 1 have priority over all players and over graphics in colour registers 2 and 3.
- 

*Fig. 8.23.* The effects of poking different numbers into address 623.

to demonstrate this, look back to the program of Fig. 8.22 which starts a series of moves in line 190. Line 190 pokes 1 into register 623, which has the effect of giving priority to players over the playfield graphics. The subroutine at line 1000 then causes the two players to move in opposite directions, and you will see them both pass 'in



front' of the column at the centre of the screen. In line 210, the number that is poked into 623 is 2, which gives the first two players priority. Since we have a first player and a third player, this gives the first player priority over both the playfield and the third player (there's no second or fourth), and we will see the first player pass in front and the other one pass behind. This uses the subroutine at line 2000 to reverse the direction of both players. Next time, with 4 poked into 623, the background has priority over both players, so that the moving players seem to pass behind the column. Finally, in line 250, poking 8 into the register gives priority to any graphics that use colour registers 0 or 1, and the players again seem to pass behind the column. Finally, lines 280 to 310 restore normal conditions in the registers, so that you aren't left with a paralysed computer at the end of the show! If you forget these steps, you will have to press the SYSTEM RESET key when the program has finished.

## Chapter Nine

# Sounding Out The Atari

The ability to produce sound is an essential feature of all modern computers. The sound of the Atari comes from two places. One of these is the loudspeaker of the TV receiver that you use to see the display, so you have more control over the volume of this sound than is possible with a lot of other computers. In addition, the Atari has its built-in speaker which is normally used to deliver the warning honks that you hear when you are using the program recorder.

What we call sound is the result of rapid changes of the pressure of the air round our ears. We don't notice these pressure changes unless they are fairly fast, and we measure the rate in terms of cycles per second, or *hertz*. A cycle of a wave is a set of changes of pressure, first in one direction, then in the other and back to normal, which we can illustrate by the graph in Fig. 9.1. The reason that we talk about a sound *wave* is because the shape of this graph is a wave shape.

The *frequency* of sound is its number of hertz – the number of cycles of changing air pressure per second. If this amount is less than about 20 hertz, we simply can't hear it, though it can still have disturbing effects. We can hear the effect of pressure waves in the air at frequencies above 20 hertz, going up to about 15000 hertz. The frequency of the waves corresponds to what we sense as the 'pitch' of a note. A low frequency of 80 to 120 hertz corresponds to a low-pitch bass note. A frequency of 400 or above corresponds to a high pitch treble note.

The amount of pressure change determines what we call the loudness of a note. This is measured in terms of *amplitude*, which is the maximum change of pressure of the air from its normal value. For complete control over the generation of sound, we need to be able to specify the amplitude, frequency, shape of wave, and also the way that the amplitude of the note changes during the time when it sounds.

The Atari allows you very little control over the built-in

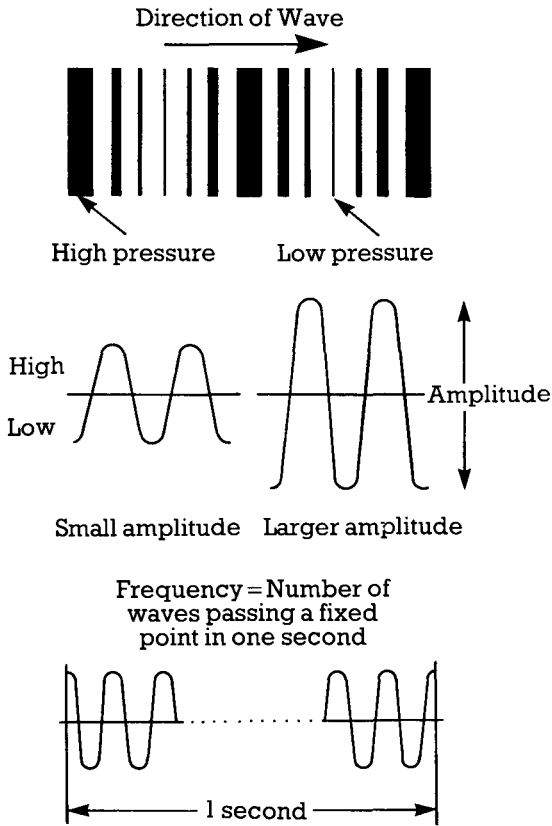


Fig. 9.1. A graph of a sound 'wave', illustrating amplitude and frequency.

loudspeaker with ordinary BASIC commands. Very much more can be done with machine code, but that is beyond the scope of this book. We can exert some influence by means of POKE instructions, as Fig. 9.2 shows. The effect of POKE 53279,0 is to push air forward from the speaker. POKE 53279,8 pulls the air back, and the two together cause a wave of sound. The Atari will perform the action of POKE 53279,8 automatically at a time which is about a fiftieth of a second after the POKE 53279,0, so that we can produce sounds by this one command. In Fig. 9.2, the loop in lines 20 to 50 performs this poke so that a buzz is produced. In lines 60 to 90, a delay loop is added to alter the pitch of the buzz. Try for yourself the effect of an extra line:

```
40 POKE 53279,8
```

on the sound that is produced by the first part of the program. You'll

```

10 GRAPHICS 0
20 FOR J=1 TO 200
30 POKE 53279,0:REM PULSE
50 NEXT J
60 FOR J=1 TO 10
70 POKE 53279,0
80 FOR Z=1 TO 50:NEXT Z
90 NEXT J
    
```

Fig. 9.2. Controlling the built-in loudspeaker directly. Only buzzing sounds can be produced with programs of this type.

find that this produces a more highly pitched buzz.

These buzzes are useful for some warning purposes but more is needed for most of the applications that we have for sound effects. This is provided by the SOUND instruction. The SOUND instruction has to be followed by four numbers. Of these, the first number is a *channel number*. The Atari can produce four notes of sound at the same time, and all four notes can be separately controlled. This is done by allocating each note to a separate 'channel'. These channel numbers can range from 0 to 3; any attempt to use higher numbers will cause an 'ERROR 3' message.

The second number that follows the SOUND instruction is the *pitch number*. This controls the pitch of the note that is produced by the Atari, and its range is 0 (highest pitch note) to 255 (lowest note). The musical equivalents are shown in Fig. 9.3, which shows the

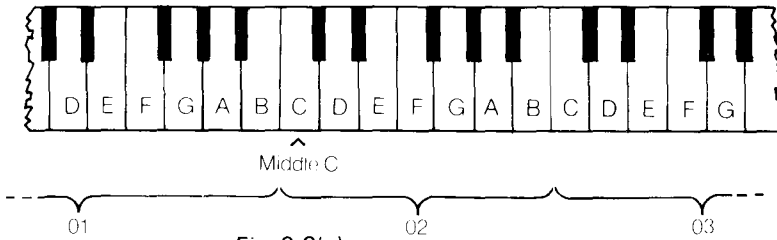


Fig. 9.3(a)

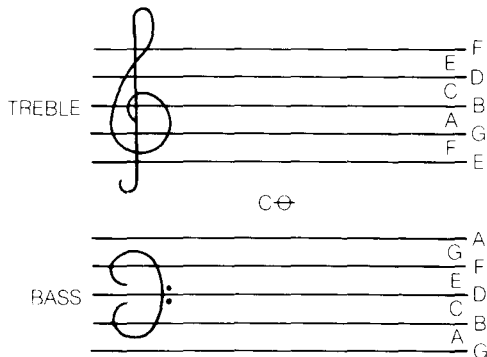
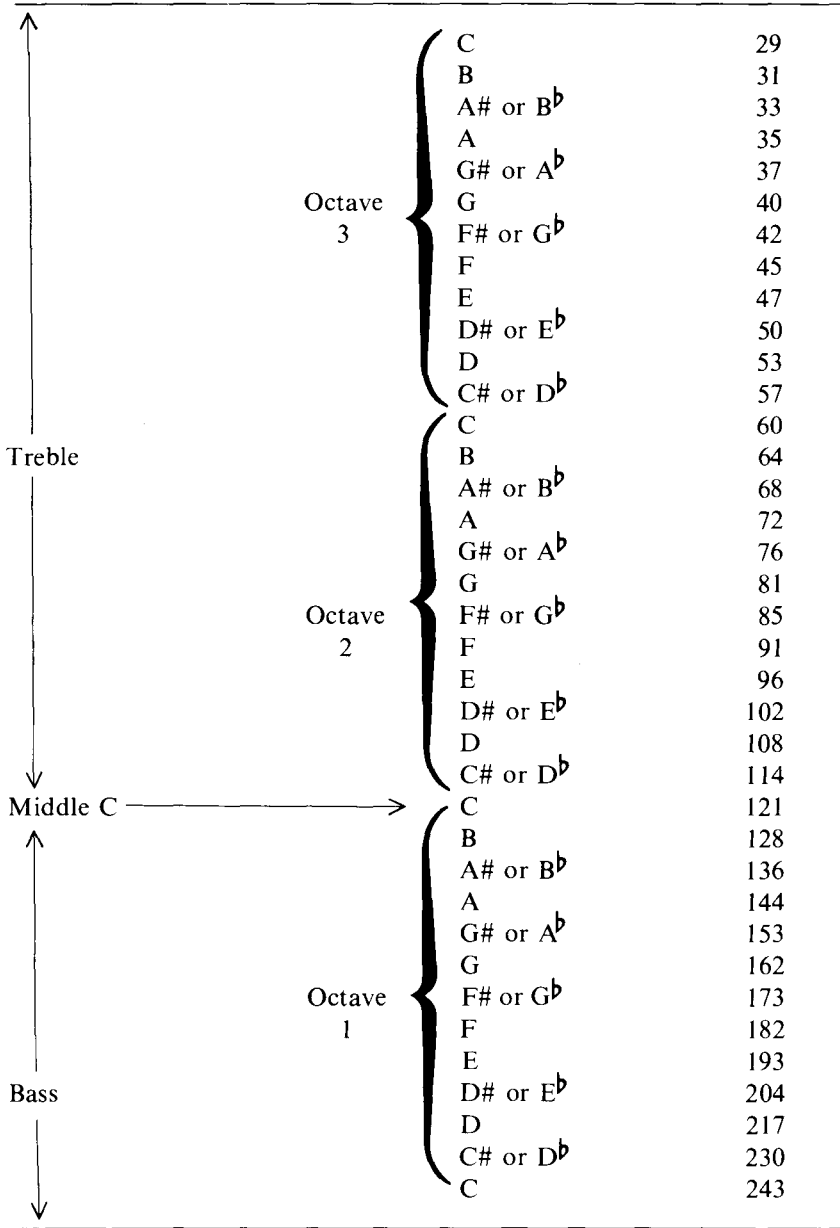


Fig. 9.3(b)



(c)

Fig. 9.3. (a) The piano keyboard, showing the notes and Middle C. (b) A table of notes with their corresponding pitch numbers. (c) Musical notation showing the notes set out on the treble and bass staves. Note the position of Middle C, between treble and bass.

piano scale and also the pitch numbers that correspond to written notes. The piano is the most familiar type of musical instrument, and its keyboard is set out so as to make it very easy to play one particular series of notes, called the 'scale of C Major'. The scale starts on a note that is called *Middle C*, and ends on a note that is also called C, but which is the eighth note above middle C. A group of eight notes like this is called an *octave*, so that the note you end with in this scale is the C which is one octave above Middle C.

The appearance of these keys on the piano keyboard is illustrated in Fig. 9.3(a). Middle C is, logically enough, at the centre of the keyboard, and we move right for higher notes, left for lower notes. One of the complications of music, however, is that the frequencies of the notes of a scale are not evenly spaced out. The 'normal' full spacing is called a 'tone' and the smaller spacing is called a 'semitone'. Each scale will contain twelve semitones.

The third number in the SOUND instruction is one that is much less simple to use. It is called a *distortion number*, and it controls the type of note that you get from the sound system. Its value has to be an even number between 0 and 14. Odd numbers cause only a single click in the loudspeaker. We shall look at the effect of different values of this number later in this chapter. For now, the value of 10 is a useful one to know, because this produces a pure tone, more suitable for music.

Finally, the fourth number sets the *relative volume* of the note. Relative volume means that if you use different values of this number with the same setting of the volume control of the TV receiver, you will hear different volumes of sound. You can set the absolute volume as you wish by using the volume control of the receiver as usual. The range of values for this number is 0 (silence) to 15 (full volume). Numbers greater than 15 will not cause an error message, but they won't cause any sound either!

Let's start our investigation of the SOUND instruction with a simple single note. This is illustrated in Fig. 9.4, which has the SOUND instruction in line 20. Channel 0 is used, and the pitch of the note is selected by the number 121, which gives Middle C. This is the note which is placed at the centre of the piano keyboard. A

```

10 GRAPHICS 0
20 SOUND 0,121,10,10
30 FOR Z=1 TO 500:NEXT Z
40 SOUND 0,0,0,0

```

Fig. 9.4. A simple single note program.

distortion number of 10 is used to give a pure sound, and a volume of 8 is used. When you are using one channel, as we are here, you have complete freedom with these volume numbers. When more than one channel is in use, however, the volume numbers have to be more carefully selected, as we shall see later.

Line 20 turns the sound on, and the action is that the sound will continue until something happens to turn it off. To control how long the sound lasts, we use a delay loop in line 30. The turn-off instruction is in line 40, in the form of SOUND 0,0,0,0. This is the method of turning off the sound that we shall use throughout this chapter. Pressing the SYSTEM RESET also turns off the sound, but that's a desperate measure!

The next step is to investigate the use of more than one channel of sound. When we do this, we have to be careful about the volume numbers. The sum of all the volume numbers of all the channels we use should not exceed 32. If it does, the sound will be distorted. This can cause some interesting effects, but for the moment we'll stick to the straightforward sound instructions. Figure 9.5 shows a chord

```

10 ? "}"
20 SOUND 0,255,10,8
30 SOUND 1,173,10,8
40 SOUND 2,144,10,8
50 SOUND 3,108,10,8
60 FOR Z=1 TO 500:NEXT Z
70 FOR C=0 TO 3:SOUND C,0,0,0:NEXT C
80 END

```

Fig. 9.5. A chord using all four channels.

being sounded with all four channels. A volume number of 8 has been used on each channel so as to keep to the limit of 32 for the total volume. As it happens, you will find when you use these instructions for music that it's often better to use lower values of volume for notes of higher pitch. This is because your ear is more sensitive to high notes than to low notes, but you will have to experiment with this for yourself. The sound is turned off in this example by using a loop in line 70 which sets all the numbers to zero in all four channels.

```

10 ? "}" : FOR J=255 TO 1 STEP -1
20 SOUND 0,J,10,8
30 FOR Z=1 TO 5:NEXT Z
40 NEXT J
50 SOUND 0,0,0,0
60 END

```

Fig 9.6. Programming for a rising pitch note.

```

10 ? "}:FOR J=1 TO 200
20 SOUND 0,121,10,8
25 FOR Z=1 TO 20:NEXT Z
30 SOUND 0,130,10,8
40 NEXT J
50 SOUND 0,0,0,0
60 END

```

*Fig. 9.7.* A warbling note program.

## Special effects department

The SOUND instruction can produce a large range of useful sound effects. Let's start with a rising pitch of note which makes a useful warning, or a 'something about to happen' note. This is illustrated in Fig. 9.6. The loop that starts in line 10 uses values of J that range from 255 to 1, the full range that the SOUND instruction permits. These are the numbers that we shall use as pitch numbers in the SOUND instruction in line 20. Line 30 is a short delay which stretches out the note.

Figure 9.7 shows a program that produces a warbling note. This is particularly useful for attracting attention, or for announcing an event in a game. For some reason, a warbling note attracts our attention more than a single note, which is why a warbling note was chosen for the later types of telephones. The warble in this program uses the loop that starts in line 10. This sounds 200 pairs of notes, which are short with a duration set by the short time delay loop in line 25. The two pitch numbers that have been chosen in this example are 121 and 130. Higher pitches are even more effective, and values like 20 and 30 give effective attention-getting warbles.

## Distortion unlimited

It's time now to investigate the effect of the distortion number on the

```

10 ? "}:FOR J=0 TO 15
20 POSITION 5,5:?"DISTORTION ";J
30 SOUND 0,121,J,8
40 FOR Z=1 TO 50:NEXT Z
50 NEXT J
60 SOUND 0,0,0,0
70 END

```

*Fig. 9.8.* Investigating the effect of different distortion numbers on one note. Try this with other note values also.



sound that you hear. The effect is anything but simple, and because it's difficult to describe in words what a noise sounds like, you simply have to try the programs and listen! We'll start in Fig. 9.8 with a simple example. This sounds Middle C, and allows the note to be sounded with all possible values of distortion number. Each number is printed on the screen just as you hear the sound. The delay loop in line 40 holds the sound long enough to make an impression. At the end of the program, line 60 cuts off the sound as usual. It's particularly interesting to try this program out with a number of different pitch values.

Now while this will introduce you to some of the effects that can be generated by the use of different distortion numbers, it tells you only part of the story. Distortion numbers of 4 and 8 are very useful for noises, but the use of a single note does not illustrate the remarkable effects that you can get with a distortion number of 12. This can have the effect of generating notes that are outside the normal range of pitch, as Fig. 9.9 illustrates. In this program, a

```

10 ? "}:FOR N=255 TO 0 STEP -1
20 POSITION 5,5: ? "NOTE ";N
30 SOUND 0,N,12,14
40 FOR Z=1 TO 100:NEXT Z: ? "}"
60 NEXT N
70 SOUND 0,0,0,0
80 END

```

Fig. 9.9. The effect of different note numbers when distortion number 12 is used.

distortion number of 12 is used along with the full range of note numbers. As the program proceeds, you will hear very low-pitched notes being produced at certain note numbers. If you want to make use of these notes, which need a large loudspeaker for good reproduction, you may want to place a GET step in line 40 rather than a delay loop. You can then listen to each note more carefully, and compare it with notes sounded on a piano or other instrument.

### More special effects

Back to the effects! Figure 9.10 shows the effect of varying the volume number in a loop that contains a SOUND instruction. The effect can be a very useful one, and it can be extended. For example, the piece of program in Fig. 9.10 can be used as a subroutine, and repeated several times. Alternatively, the volume

```

10 ? "}" :FOR N=0 TO 15
20 SOUND 0,81,14,N
30 FOR J=1 TO 50:NEXT J
40 NEXT N

```

*Fig. 9.10.* Varying the volume number in a loop.

can be decreased instead of being increased, or it can be alternately increased and decreased. As is the case with all of these Atari instructions, there is a lot of room for experimenting to find the effect that you want.

After that introduction, let's go to some more full-blooded sound effects which make use of all that we have discovered so far. Figure 9.11 gives a reasonable impression of surf breaking on a distant shore. The number of waves is given by the outer loop, using Z. The pitch of the sound is regulated by the number N in the next loop, and the type of noise is chosen by using a distortion number of 8 in line 20. The delay loop in line 30 holds the sound long enough to appear realistic, and when all of the loops have finished, line 50 stops it all and brings you back to reality.

```

10 ? "}" :FOR Z=1 TO 5:FOR N=50 TO 0 STEP
-1
20 SOUND 0,N,8,10
30 FOR J=1 TO 50:NEXT J
40 NEXT N:NEXT Z
50 SOUND 0,0,0,0
60 END

```

*Fig. 9.11.* A program that creates a surf sound.

Next item, gunshots. Figure 9.12 attends to this useful sound effect, using a distortion number of 0. The pitch number of 2 is used (there is plenty of scope for experiment here), along with maximum volume. The duration of the shot is fixed by the delay loop in line 30, and the number of shots (six, of course) by the loop that starts in line 10. You can modify this type of sound to give a 'pneumatic drill' effect, as Fig. 9.13 shows. This one uses a distortion number of 2 and a pitch number of 100. The loop in line 30 controls the time for which the drill is operating, and the loop in line 60 controls the off

```

10 ? "}" :FOR X=1 TO 6
20 SOUND 0,2,0,15
30 FOR J=1 TO 100:NEXT J
40 SOUND 0,0,0,0
50 FOR J=1 TO 500:NEXT J
60 NEXT X
70 END

```

*Fig. 9.12.* A gunshot effect.

```

10 ? "}" :FOR X=1 TO 5
20 SOUND 0,100,2,15
30 FOR J=1 TO 500:NEXT J
40 SOUND 0,0,0,0
60 FOR J=1 TO 1000:NEXT J
70 NEXT X

```

Fig. 9.13. A pneumatic drill effect.

```

10 ? "}" :FOR X=1 TO 10
20 GOSUB 1000
30 FOR J=1 TO 100:NEXT J
40 GOSUB 1000
50 FOR J=1 TO 500:NEXT J
60 NEXT X
70 END
1000 SOUND 0,80,10,10
1010 SOUND 1,100,4,10
1020 FOR J=1 TO 100:NEXT J
1030 SOUND 0,0,0,0:SOUND 1,0,0,0
1040 RETURN

```

Fig. 9.14. The sound of an unanswered phone!

time. The actual pulsating noise of the drill is produced by the combination of distortion and pitch numbers that is used.

Another noise is produced by the program in Fig. 9.14. This time, the sound is the one you hear when you dial a number on the telephone. In my case, I hear it too often – I wish someone would answer now and again! Two channels are used to produce this mixture of noises. Channel 0 produces a pure tone, with channel 1 being used to add the noise to it. The two different delay loops produce the correct spacing of sounds for the British telephone system.

Looking for aircraft noises? Try Fig. 9.15 for the sound of a plane

```

10 ? "}"
20 FOR X=20 TO 60
30 SOUND 0,X,8,X/4
35 GOSUB 1000
40 NEXT X
50 FOR X=60 TO 20 STEP -1
60 SOUND 0,X,8,X/8+5
65 GOSUB 1000
70 NEXT X
80 SOUND 0,0,0,0
90 END
1000 FOR J=1 TO 50:NEXT J
1010 RETURN

```

Fig. 9.15. An aircraft taking off and passing overhead.

```

10 FOR D=1 TO 500
20 SOUND 0,20,0,15
40 SOUND 0,20,1,15
60 NEXT D
70 SOUND 0,0,0,0

```

*Fig. 9.16.* Piston-engined sound for the fanatic.

taking off and passing overhead. Both volume and pitch are being changed here in the loop that starts in line 20, and also in the second loop of lines 50 to 70. If you hanker after piston engines, try the sound from Fig. 9.16!

### Melodies for you!

After that barrage of shots and other noises, let's end with a piece of Atari music. The fact that the Atari can make use of four channels of

```

5 OPEN #1,4,0,"K:"
10 ? "}:FOR J=1 TO 16:READ A,B,C,T
20 SOUND 0,A,10,6:SOUND 1,B,10,6:SOUND 2
,C,10,8
30 FOR N=1 TO T:NEXT N
40 FOR N=1 TO 100:NEXT N
50 NEXT J
70 SOUND 0,128,10,10:SOUND 1,204,10,8:SO
UND 2,255,10,8
80 FOR N=1 TO 500:NEXT N
110 SOUND 0,0,0,0:SOUND 1,0,0,0:SOUND 2,
0,0,0
120 END
500 DATA 81,121,193,100,60,121,193,100
510 DATA 96,144,193,100,81,144,193,50,91
,144,193,50
520 DATA 96,144,193,100,102,121,193,100,
96,144,193,100
530 DATA 128,162,217,100,136,173,217,100
540 DATA 128,182,230,100,108,144,217,100
550 DATA 114,162,217,50,128,162,217,50
560 DATA 81,114,193,100,85,121,193,100

```

*Fig. 9.17.* Illustrating the use of SOUND to produce melody and harmony. The first line is put in for fault-finding purposes, as the text explains.

sound allows you an excellent capability for music-making. The ideal way of writing a music program is by following a musical score, but that's not quite so easy if you can't read music! If you have a good ear for music, though, you can get by with some patience and a lot of typing.

Figure 9.17 shows a program which produces a melody with harmony. Line 10 is the important opener, with the number of notes specified in the loop, and the READ instruction. Only three channels are used, and the variables A, B and C are used for the note numbers of the channels. The fourth variable, T, is used to control the time of each note. By reading a set of four numbers for each note, we can change the sound more easily if we want to. The SOUND instructions are then placed in line 20.

The delay in line 30 is controlled by the time numbers that are placed in the data lines. In addition, there is an extra delay in line 40. This is fixed, and it's a useful way of controlling the timing of the whole piece. If you use the variable T to control time completely, then if you want to play the music faster, you will have to alter all of the values of T in the data lines. By using a separate delay loop, it's easy to adjust the timing (unless you want a large change) by a single alteration. The last note is programmed separately, and line 110 restores silence.

Now it's one thing to write the program and put in values for the notes, but how do you sort it out? The answer is, as you might expect, a bit at a time. Notice line 5? This is the OPEN instruction that you need for a GET, but there's no GET in the program. The point is that you can add a GET in place of the delay in line 40. This allows you to listen to each sound until you press a key. Having time to listen makes it easier to decide if it's right or not. Another help is to alter line 20. If, for example, you make the volume numbers in channels 1 and 2 equal to 0, then you will hear only the melody in channel 0. You can then make whatever alterations you need to get this right. Having done so, you can then restore channel 1, and concentrate on getting this so that it's a good accompaniment to the melody in channel 0. When you have sorted out these two channels, you can then try adding channel 2. Taking the problem note by note in this way makes it much easier to handle if you are working 'by ear'. If you are working from a score, of course, all you need is the chart in Fig. 9.3. Happy listening!

## Chapter Ten

# Odds And Ends

A computer which is as complicated as the Atari contains lots of secrets. A book which sets out to document all of the possibilities in detail would be of encyclopaedic length, and would have to be added to continually. In this book, I have tried to equip you with enough fundamental knowledge of your Atari to set you on the path of learning. Inevitably, some topics have been omitted, some because they require more knowledge of how the computer works than you may have at this stage. Other topics, however, simply have not fitted in to the arrangement of chapters, and some of this chapter will be devoted to sorting out these omissions. If this book has stimulated you to seek more information on your Atari, then your best step is to join a User Group – one address is given in Appendix B. Members of User Groups interchange information continually, and their efforts result in much more being known about the computer than is available from the manufacturers.

### Editing

The first of the neglected topics is editing. Editing means altering a line that has been entered by pressing the RETURN key. While you are typing a line, you can edit by back-spacing over a mistake, using the BACK SPACE key at the right-hand top row of keys. This is no longer possible once the line has been entered into memory, however, and the process of altering such a line comes under the heading of editing.

All editing must start by having the line that you want to edit visible on the screen. This can be achieved by using LIST, and if you know the number of the line that you want, say 200, then you can use a command such as LIST 200 to get that line by itself. The next step is to shift the cursor to that line by making use of the CTRL key

along with the arrowed keys. These are the keys which are normally used for the arithmetic operations of +, -, =, and \*, and the arrow directions are also marked on them. They operate as editing keys only when the CTRL key is pressed at the same time. Normally, you will get to the start of a line by using the up-arrow key.

When the cursor is on the correct line, it can be moved to the error by using another arrowed key, usually the right-arrow. As before, the CTRL key must be held down while this is being done. When you get the cursor to the error you have the choice of replacing the character, deleting it, or adding more characters. To replace the character, place the cursor directly over it, release the CTRL key, and just type the correct character. This will replace the incorrect character. If you want to delete a character, place the cursor over it, keep the CTRL key pressed, and press the DELETE key, which is the same as the BACKSPACE key. If you want to add more characters, press the CTRL key and the INSERT key (on the > key) and hold down both until you have enough space to put in what you want. You can then release the CTRL key and type as many characters as will fill the space. If you haven't allowed enough space, use INSERT again. If you have space left, you can use DELETE to remove spaces.

Once you have made all the changes that you want, you can confirm the changes by pressing RETURN. If you move the cursor away from the line before you press RETURN, the changes will not be carried out. You have to be careful after pressing RETURN. When you do this, the cursor will appear at the start of the next line. If the rest of this line is blank, then you can type a command, like LIST or RUN, and it will be obeyed when you press RETURN. You may find, however, that the cursor lies over the R of 'READY'. If you press RETURN while the cursor is in this position, you will get an error message. You will also get an error message if you have typed LIST or RUN in this situation, because the word that you type replaces only part of the READY word. You should use the CTRL key along with the arrow keys to place the cursor clear of any lines before you attempt to make any new command.

## **The TRAP instruction**

Among other interesting and useful functions of the Atari that we have not looked at so far is the TRAP instruction. TRAP is a way of detecting errors so that you can avoid the normal error message

system. Normally, when an error occurs during a program, the program halts and displays an error message. This can be a nuisance, particularly if an inexperienced operator is in charge, so that the Atari allows you to write your own error-handling programs! This is done by using TRAP to direct the program to a subroutine which will deal with the error and then return the program to normal running. Obviously, if you are to incorporate this sort of automatic error-handling, you must know what type of errors you are likely to find.

As usual, an example is more useful than a lot of explanation, and Fig. 10.1 shows a very simple example of TRAP in action. Line 10

```

10 ? "3":TRAP 1000
20 FOR J=1 TO 5:READ A
30 ? A:
35 POSITION 10,J: ? SQR(A)
40 NEXT J
50 DATA 4,9,-12,16,20
60 END
1000 E=PEEK(195):REM FIND ERROR
1010 L=PEEK(187)*256+PEEK(186):REM FIND
ADDRESS
1020 IF E=3 THEN A=-A
1030 TRAP 1000
1040 GOTO L

```

*Fig. 10.1.* Illustrating the use of TRAP.

clears the screen, and contains the instruction TRAP 1000. This will cause the program to go to line 1000 if any error is detected, so that 1000 is the start of the error subroutine. At line 1000, the action of PEEKing address 195 will give the code number of the error. For a list of these codes, see your Atari manual. In line 1010, the formula for L will give the line number in which the error was found. Once you have these two pieces of information, you can set about correcting errors. For example, if you know that an error 3 is likely, you can test for it, as is done in line 1020. If, on the other hand, you know that an error can occur in a line whose number you know, then you can use a test such as:

IF L = 150 THEN (whatever you need)

The use of TRAP is self-cancelling – once an error is found, the line TRAP 1000 will have to be executed again if another error is to be trapped. This line therefore used at the end of the trapping subroutine. Finally, line 1040 uses GOTO L to get the program back to the line that caused the trouble. In the example, square roots are



being found, and the TRAP routine deals with the problem of a negative number. Instead of stopping the program, the negative number is converted to positive, and the result of this used.

The important point about using TRAP is that you have to know either what type of error you expect, or in which line it might occur. This simple example has used data taken from a data line, but it's more likely that you would want to use TRAP when an input was being taken from tape or disk, or from the keyboard.

### The games keys

The keys on the extreme right-hand side of the keyboard are not normally used when you are writing your own programs, but are extensively used in games that are in cartridge form. You can, however, make use of the OPTION, SELECT and START keys for yourself. This is done by using PEEK (53279). The number which is produced by this action depends on which of these keys is or are pressed. The number can also detect combinations of these keys, as the table in Fig. 10.2 shows. You can use a test such as:

```
IF PEEK (53279) = 5 THEN LIST
```

to make these keys carry out program actions for you.

---

Values of PEEK(53279)	
Value	Key or Keys pressed
0	OPTION, SELECT and START
1	OPTION and SELECT
2	OPTION and START
3	OPTION
4	SELECT and START
5	SELECT
6	START
7	None of these keys.

---

Fig. 10.2. Making use of the games keys.

## **The Atari add-ons**

One of the particular advantages of the Atari system is that every part of it is available *now*, not simply a 'this year, next year, sometime' promise. The computer has been planned out so as to offer a large selection of add-ons, all of which can be obtained from your Atari dealer.

For example, up to four joysticks or paddles can be plugged into the sockets that are on the front of the computer below the keyboard. They allow for control of games programs, and the BASIC also allows you to write your joystick/paddle programs, using the instruction words PADDLE, PTRIG, STICK and STRIG. The techniques are explained in the Manual.

Other additions to the ATARI system require the use of the Atari 850 interface module. This plugs into the socket that is otherwise used for the program recorder, and the correct cable and plugs are supplied. The program recorder can, in turn, be plugged into the interface. In addition, the interface permits the use of printers. Either Atari printers or printers of other makes can be connected to the interface, and the floppy disk drives are also connected to this unit.

The use of floppy disks is a major step forward for you in the use of your computer. The disk system replaces the program recorder, allowing very rapid storing and loading of programs or data. Details of disk operation are, however, beyond the scope of this book.

## Appendix A

# SAVE and LOAD Problems

A minor problem of the Atari computers is that occasionally a program which has been saved by using the CSAVE instruction will not correctly load again using the CLOAD instruction. The problem is intermittent, and can be overcome by using the LPRINT command before the program is saved. This has the action of completely clearing the part of memory that is used when the CSAVE instruction is carried out.

An alternative is to use a different type of saving and loading command. The most useful alternative is SAVE"C:" for saving and LOAD"C:" for loading. In this type of command, the "C:" name refers to the cassette recorder. Both CSAVE and SAVE"C:" save the program in the same abbreviated form, but you cannot load a program, using CLOAD, if it has been saved using SAVE"C:". Similarly, you cannot load with LOAD"C:" a program which has been saved using CSAVE. The use of SAVE"C:" and LOAD"C:" is slightly slower than the use of CSAVE and CLOAD.

There is yet another pair of save and load commands, LIST"C:" and ENTER"C:". These operate in a different way, because what LIST"C:" stores on tape is a representation of the ASCII codes of the program as typed. LIST"C:" can, in fact, be treated in the same way as LIST, so that it is possible to LIST"C:" one line, or a range of lines as well as a complete program. This is done by using the same format as you would use for a LIST to screen. This is of considerable use if you want to save only part of a program, such as a number of subroutines.

The matching loading command, ENTER"C:" causes the ASCII codes of a program recorded on tape to be entered into the Atari just as if they had been entered from the keyboard. Whereas the CLOAD and the LOAD"C:" commands automatically clear memory before they start loading, the ENTER"C:" does not. It is therefore possible to merge one program with another, providing that the

programs have different line numbers. This allows you to keep a 'library' of subroutines which you can place into any other program as needed. The only proviso is that the subroutines must have been recorded on tape using the LIST"C: " command. Material that has been recorded using CSAVE or SAVE"C:" will not successfully load using ENTER"C: ".

## Appendix B

# Useful Addresses

### **Selmor Industries Ltd**

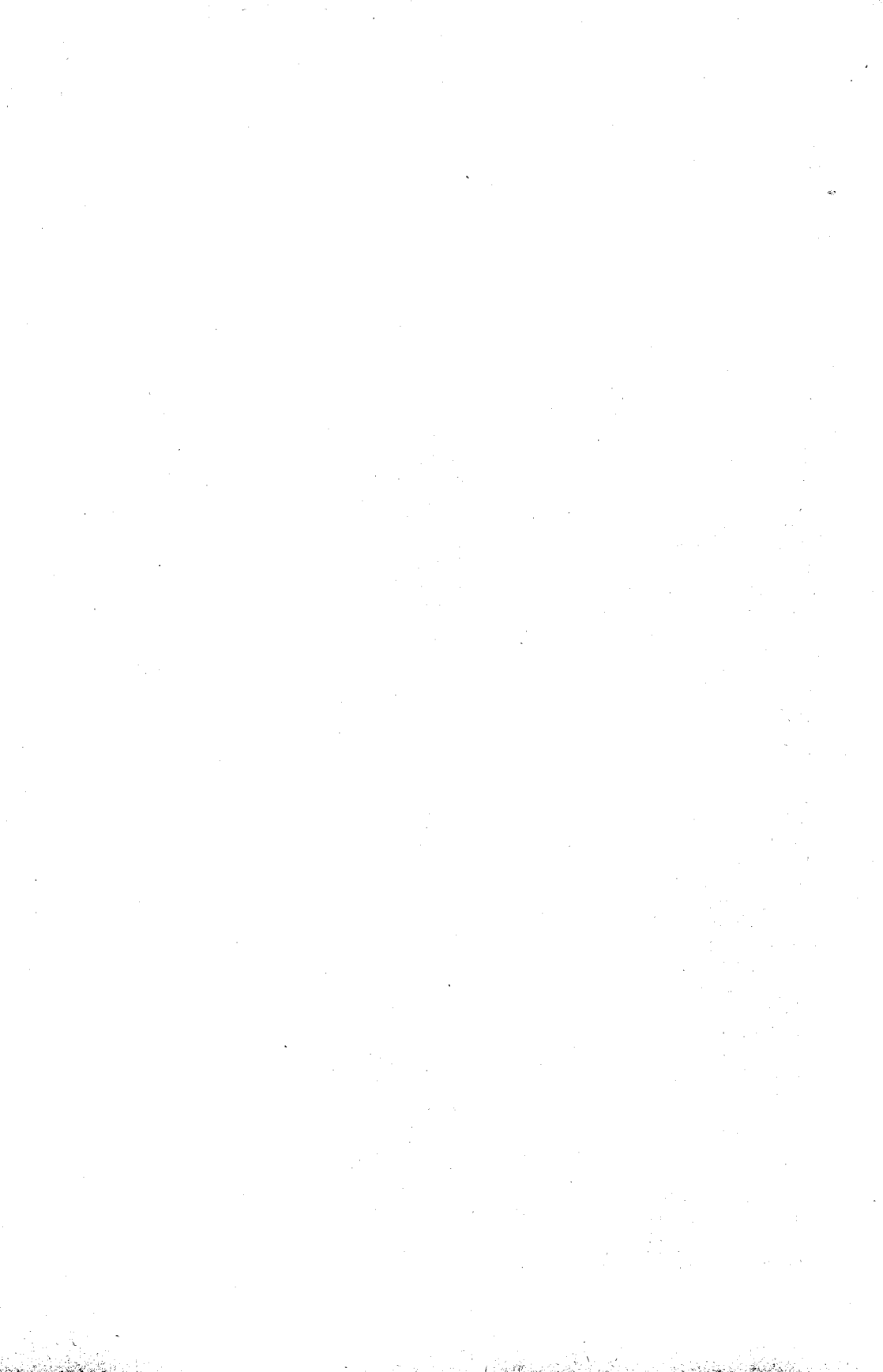
24 Mulberry Street  
Tower Hamlets  
London E1 1EH

Selmor Industries are makers of the computer stand which is mentioned in Chapter 1, and a wide range of other devices, including methods of locking computers to tables. They supply educational authorities, and their prices are very much lower than those asked by firms advertising in the Press.

### **Silica Atari 400/800 Users Club**

Richard Hawes  
1-4 The Mews  
Hatherly Road  
Sidcup  
Kent DA14 4DX

**Note:** The magazine, *Practical Computing* runs a section which deals with specific machines. Useful information on Atari models is now beginning to appear in this section of the magazine.



# Index

- adaptor, 2-to-1, 1
- add-ons, 140
- address for sprite data, 113
- aerial plug, 1
- aircraft noise, 132
- alphabetical order, 59
- alternate character set, 87
- amplitude of sound, 124
- angles, 100
- animating a dot, 110
- animation, 97
- array subroutine, 77
- ASC, 57
- ASCII codes, 45
- assignment, 23
- asterisk, 15
  
- background colour, 101
- backslash, 15
- BACKSPACE, 137
- BASIC, 14
- BASIC cartridge, 4
- becomes sign, 32
- bent arrow shape, 22
- binary fraction, 33
- blank cassettes, 10
- border colours, 83
- BREAK key, 9
- brightness, 82
- built-in speaker, 124
- buzz, 125
- bytes, 24
  
- carriage return, 9
- cassette lead, 10
- centring a title, 21
- channel number, 66
- channel number, sound, 126
- character design grid, 88
- character number, 18
  
- chord, 129
- CHR\$, 57
- circles, 104
- CLEAR key, 13
- CLOAD, 12
- CLOG, 33
- CLOSE#1, 67
- co-ordinates, 99
- coarse-resolution sprites, 113
- COLOR, 84
- colour changes, 12
- colour monitor, 5
- colour of sprites, 117
- colour receiver, 3
- coloured letters, 87
- column number, 99
- columns, 18
- commas, 18
- compare strings, 58
- concatenation, 26
- counting, 32
- creating sprites, 112
- creating string arrays, 63
- CSAVE, 11
- CTRL key, 9
- curly bracket, 22
- cursor, 9
- cycle of wave, 124
  
- DATA, 30
- database programs, 69
- decisions, 42
- decrementing, 32
- DEG, 100
- DELETE, 137
- designing character, 90
- designing programs, 69
- dial tuning, 5
- DIM, 24, 61
- direct mode, 13

## 146 *Index*

- distortion number, 128, 130
- dimensioning string, 24
- DRAWTO, 99, 101
  
- editing, 136
- enabling addresses, sprite, 116
- end of file, 66
- ENTER "C:", 141
- equations, 100
- ESC key, 99
- expanding outline, 71
- expression, 23
- extracting initials, 54
  
- fan patterns, 108
- fine-resolution sprites, 113
- flag, 64
- floppy-disk, 140
- FOR, 37
- forbidden operation, 26
- frequency of sound, 124
  
- games keys, 139
- GET, 44
- GOSUB, 49, 72
- GOTO, 36
- graphics, 80
- graphics blocks, 88
- graphics modes list, 80
- graphs, 100
- grid, character design, 88
- grid, multiple character, 93
- gunshots, 132
  
- hard copy, 16
- harmony, 134
- hertz, 124
- high resolution, 81
- higher modes, 108
  
- identity sign, 58
- impressive shapes, 93
- incrementing, 32
- indented word, 19
- INPUT, 28
- INSERT, 137
- instruction words, 14
- instructions, 78
- INT, 43
- interface module, 140
- internal codes, 92
- inverted commas, 16
  
- jellyfish, 120
  
- joining strings, 26
- joysticks, 140
  
- leader of tape, 10
- LEFT\$, 56
- LEN, 52
- line number, 14
- line numbers, 11
- LIST"C:", 141
- LIST, 11
- LOAD"C:", 141
- LOCATE, 110
- LOG, 33
- long variable names, 25
- loops, 36
- loudspeaker, 4
- low resolution, 80
- lower-case, 8
- LPRINT, 16
- luminance, 82
- luminance of sprites, 117
  
- mains sockets, 3
- melody, 134
- memory allocation, sprite, 114
- menu, 45
- MID\$, 56
- Middle C, 128
- missile memory allocation, 114
- missile widths, 116
- missiles, 111
- mistuning faults, 7
- Mode 1 characters, 86
- Mode 2 characters, 86
- Mode 8, 107
- Mode zero, 81
- modes 3-8, 98
- modes of graphics, 81
- modulator, 5
- monitor, 5
- mugtrap, 43
- multiple-character grid, 93
- multistatement line, 18
  
- name, 24
- neat printing, 17
- nested loops, 37
- NEW, 11
- NEXT, 37
- not-equal sign, 41
- number abilities, 31
- number comparisons, 41
- number functions, 33, 34
- number of variables, 59



- number totalling program, 40
- octave, 128
- OPEN, 44
- open channel, 66
- OPTION, 139
- paddles, 140
- painting in colour, 105
- passing a variable, 53
- PEEK, 87
- pictures, 80
- piston engine sound, 132
- pitch, 124
- pitch number, 126
- pixels, 80
- planning chart, sprite, 122
- planning pattern, 102
- player demonstration, 119
- player widths, 116
- player-missile graphics, 97
- players, 111
- playfield, 120
- PLOT, 99
- pneumatic drill, 132
- POKE, 87
- POSITION, 20
- positive integer, 14
- power input socket, 4
- power pack, 3
- precision of number, 33
- pressure changes, 124
- print modifiers, 17
- PRINT#, 66
- PRINT#, 85
- PRINT@, 22
- PRINTAT, 22
- printer, 140
- priority, 120
- program, 13
- program mode, 13
- program outline plan, 70
- program recorder, 10
- prompt, 14
- pure sound, 129
- push-button tuning, 7
- question and answer game, 79
- quotes, 16
- radians, 100
- random number, 43
- READ, 30
- READY, 14
- recording programs, 10
- recording array values, 68
- relative volume, 128
- REM, 72
- repeat actions, 36
- replaying array values, 68
- reserved words, 14
- resolution, 80
- RETURN, 9
- RIGHT\$, 56
- rising pitch note, 129
- RND, 43
- ROM, 89
- rounding up, 33
- row number, 99
- SAVE"C:", 141
- saving variables, 65
- scale, music, 128
- scientific form, 34
- score subroutine, 76
- scoring system, 70
- screen limits, 20
- screen splitting, 84
- SELECT, 139
- Selmor, 1
- semicolon, 17
- semitone, 128
- SETCOLOR, 81
- setting up, 1
- SHIFT key, 8
- short length tapes, 10
- single key reply, 44
- skipping stages, 73
- slash-mark, 15
- slicing, 54
- socket strip, 3
- SOUND, 126
- sound, 124
- sound effects, 132
- split screen, 98
- sprite, 111
- sprite design summary, 118
- sprite graphics, 111
- sprites, 97
- SQR, 33
- stands, 1
- START, 139
- STEP, 38
- STR\$, 56
- string, 16
- string arrays, 61
- string functions, 51
- string insertion, 51

## 148 *Index*

- string slicing, 54
- string variables, 24
- STRINGS, 52
- subroutine design, 73
- subroutines, 49
- subscript, 60
- subscripted variable, 60
- surf breaking sound, 132
  
- TAB stops, 19
- table of notes, 127
- tabulation, 19
- terminator, 40
- testing strings, 42
- title, 79
- tone, 128
- totalling numbers, 39
- TRAP, 137
- TV cable, 1
- TV receiver, 1
- TV tuning, 5
  
- twinkling pattern, 105
- two-colour modes, 106
- typewriter keyboard, 8
  
- uncontrolled loop, 37
- underlining, 52
- upper-case, 8
- user defined graphics, 89
- user group, 136
- user-defined characters, 88
  
- VAL, 56
- variable name, 23
- VCR tuning, 5
  
- warbling note program, 130
- wave of sound, 124
- working copy, 72
  
- XIO, 105

## **The TI 99/4A**

### **GET MORE FROM THE TI99/4A**

Garry Marshall  
0 246 12281 1

## **The VIC 20**

### **GET MORE FROM THE VIC 20**

Owen Bishop  
0 246 12148 3

### **THE VIC 20 GAMES BOOK**

Owen Bishop  
0 246 12187 4

## **The ZX Spectrum**

### **THE ZX SPECTRUM And How To Get The Most From It**

Ian Sinclair  
0 246 12018 5

### **THE SPECTRUM PROGRAMMER**

S. M. Gee  
0 246 12025 8

### **THE SPECTRUM BOOK OF GAMES**

M. James, S. M. Gee and K. Ewbank  
0 246 12047 9

### **INTRODUCING SPECTRUM MACHINE CODE**

Ian Sinclair  
0 246 12082 7

### **SPECTRUM GRAPHICS AND SOUND**

Steve Money  
0 246 12192 0

### **THE ZX SPECTRUM How to Use and Program**

Ian Sinclair  
0 586 06104 5

## **The ZX81**

### **THE ZX81 How to Use and Program**

S. M. Gee and Mike James  
0 586 06105 3

## **Which Computer?**

### **CHOOSING A MICROCOMPUTER**

Francis Samish  
0 246 12029 0

## **Languages**

### **COMPUTER LANGUAGES AND THEIR USES**

Garry Marshall  
0 246 12022 3

### **EXPLORING FORTH**

Owen Bishop  
0 246 12188 2

## **Machine Code**

### **Z-80 MACHINE CODE FOR HUMANS**

Alan Tootill and David Barrow  
0 246 12031 2

### **6502 MACHINE CODE FOR HUMANS**

Alan Tootill and David Barrow  
0 246 12076 2

## **Using Your Micro**

### **COMPUTING FOR THE HOBBYIST AND SMALL BUSINESS**

A. P. Stephenson  
0 246 12023 1

### **DATABASES FOR FUN AND PROFIT**

Nigel Freestone  
0 246 12032 0

### **SIMPLE INTERFACING PROJECTS**

Owen Bishop  
0 246 12026 6

### **INSIDE YOUR COMPUTER**

Ian Sinclair  
0 246 12235 8

## **Programming**

### **THE COMPLETE PROGRAMMER**

Mike James  
0 246 12015 0

### **PROGRAMMING WITH GRAPHICS**

Garry Marshall  
0 246 12021 5

## **MAKE IT HAPPEN WITH ATARI!**

The ATARI is one of the most sophisticated micros available. It is exceptionally well designed, and both the ATARI 400 and 800 are well established and free from the faults that plague many machines. Plenty of software is available, not only for games but also for educational and business purposes, and the ATARI offers superb colour and excellent sound on three channels. Really stunning visual displays and sound effects can be achieved.

Though it has always had a justifiably high reputation as a games machine, the ATARI's remarkable capabilities have rarely been set out in a suitable way for beginners who want to program the machine themselves. Here at last is a book which enables you to get more from the ATARI 400 or 800 than you would ever have imagined possible. It shows you how to program, how to create your own special effects and how to get the most from the ATARI's remarkable range of facilities. Learning is made easy, enjoyable and fascinating.

### *The Author*

Ian Sinclair is a well-known and regular contributor to journals such as *Personal Computer World*, *Computing Today*, *Electronics and Computing*, *Hobby Electronics* and *Electronics Today International*. He has written some forty books on aspects of electronics and computing, mainly aimed at the beginner.

