

CENTURY
COMMUNICATIONS

THE ATARI 130XE HANDBOOK

*A comprehensive guide
to the functions and
applications of
THE ATARI 130XE*

LUPTON & ROBINSON

**THE
ATARI 130XE
HANDBOOK**

In association with 'Personal Computer World'

Microcomputing for Business: A User's Guide

edited by DICK OLNEY

The Microcomputer Handbook: A Buyer's Guide

edited by DICK OLNEY

The Spectrum Handbook

TIM LANGDELL

The Intimate Machine

NEIL FRUDE

35 Educational Programs for the BBC Micro

IAN MURRAY

Educational programs for the Dragon 32

IAN MURRAY

Educational Programs for the Spectrum

IAN MURRAY

Information Technology Yearbook

edited by PHILIP HILLS

The Database Primer

ROSE DEAKIN

Computer Gamesmanship

DAVID LEVY

Best of PCW: Software for the BBC Micro

Best of PCW: Software for the Dragon

Best of PCW: Software for the Spectrum

The ORIC Handbook

PETER LUPTON and FRAZER ROBINSON

The Commodore 64 Handbook

PETER LUPTON and FRAZER ROBINSON

35 Programs for the Dragon 32

TIM LANGDELL

THE ATARI 130XE HANDBOOK

PETER LUPTON & FRAZER ROBINSON



CENTURY COMMUNICATIONS

LONDON

Copyright © Peter Lupton and Frazer Robinson 1985

All rights reserved

First published in Great Britain in 1985
by Century Communications Ltd
A division of Century Hutchinson
Brookmount House
62 - 65 Chandos Place
London
WC2N 4NW

ISBN 0 7126 97055

Typeset by Spokesman, Bracknell
Printed in England by the Sidmouth and Devon Print Group

CONTENTS

Acknowledgements

Introduction

1	Computers and Programs	1
2	Setting up your 130XE	3
3	First Steps	8
4	Programming	18
5	Program Control	31
6	Data and Programs	45
7	Pieces of Strings	52
8	Functions	63
9	Logical Thinking	70
10	Memory Management	74
11	Sound and Music	77
12	Characters	105
13	Graphics	136
14	Advanced Graphics	153
15	Player-Missile Graphics	173
16	Permanent Storage	189
17	Advanced Techniques	197

APPENDICES

2	BASIC Error Messages	215
3	Speeding Up Programs	223
4	Numbering Systems	225
5	Graphics Modes	230
6	Colour Codes	231
7	Character Codes	232
8	Display List Instructions	234
9	Useful Memory Locations	236
10	Music Frequencies	238
11	Musical Notation	239
12	System Memory Map	243
	Index	244

ACKNOWLEDGEMENTS

Our thanks to all those who in some way or other made this book possible. These include (in no particular order):

Steph for helping to heat up the treacle pudding 5 years ago in Leeds, Alison and Steve for wet weekends in Somerset, Mac Peters for the Irish Moss, Don Gray for being on our side, Bronwyn Doughty at Atari in Slough for our machines, Peter Norton for his sleeves and finally Ian, Paul and Kevin for the salads.

ABOUT THE AUTHORS

Frazer Robinson graduated from Leeds University and works (occasionally) as a technical author and programmer for a leading UK computer company. He always wears his sleeves rolled up except when it's too cold or he's wearing a T shirt.

Peter Lupton graduated from Imperial College, London and is employed as a consultant by the same company. As he owns several short sleeved shirts it is sometimes difficult to tell how he's wearing his sleeves.

INTRODUCTION

This book is an introduction to the ATARI 130XE computer and its predecessors (Atari 400, 800 and the XL range). It takes you from the first steps in BASIC programming, pointing out the pitfalls and explaining how to make the most of the features of ATARI BASIC, through to a detailed understanding of how to write good, structured programs, with many illustrative examples along the way.

A chapter is devoted to the extensive sound capabilities of the XL, including details of the sound generating chip itself which allow you to create a wider range of effects than is possible using the standard commands.

Four chapters cover the incredible graphics facilities of the machines, including many spectacular examples and useful programs. An advanced chapter shows how a detailed knowledge of the way graphics are created leads to even more exciting effects.

A comprehensive set of appendices ensure that the book will remain a valuable reference work long after you have grasped the principles of the machine.

We hope that you will find this book both enjoyable and instructive and that it enables you to exploit the potential of your Atari to the full.

PETER LUPTON and FRAZER ROBINSON 1985

CHAPTER 1

COMPUTERS AND PROGRAMS

Despite everything you may have heard about computers being hyperintelligent machines which are on the point of taking over the world, a computer is not really intelligent at all. A computer is at heart little more than a high-speed adding machine, similar to an electronic calculator, but with more sophisticated display and memory facilities.

The feature that gives computers their immense flexibility and potential is this: they can store lists of instructions which they can read through and obey at very high speed. These lists of instructions are called programs, and most of this book is concerned with how these programs are written.

The computer instructions which form programs must follow certain rules, or the computer will not be able to understand them. The rules for writing programs resemble the rules of a spoken language, and so the set of instructions is often said to form a programming language. There are many different computer programming languages; the one that the Atari 130XE understands (in common with most other personal computers) is called BASIC. (The name BASIC is an acronym for Beginners' All-purpose Symbolic Instruction Code.)

A programming language is much simpler than a human language because computers, for all their power, are not as good at understanding languages as people are. The BASIC language used by the

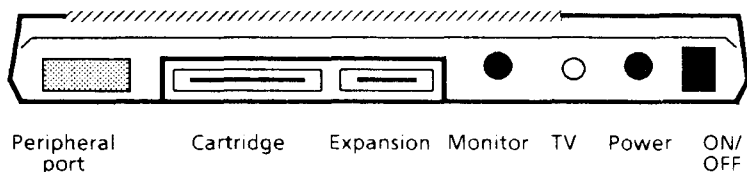
130XE has only about 80 words. The rules for combining the words - the 'grammar' of the language - are much more strict than for a language like English, again because it is difficult to make computers that can use languages in the relaxed sort of way in which we speak English. These may seem like limitations, but in fact as you will discover BASIC is still a powerful language, and it is possible to write programs to perform very complex tasks.

Finally, remember this. Your computer will not do anything unless you tell it to, so whatever happens, you're the boss. Your Atari won't take over the world unless you make it!

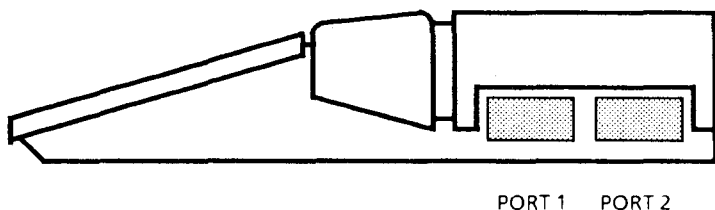
CHAPTER 2

SETTING UP YOUR 130XE

Before you can use your Atari 130XE, you must connect it to a power supply and to a television. To



The Atari 130XE Connectors - Rear View



The Atari 130XE Connectors - Side View

load and save programs you will also need to connect a cassette unit to the 130XE. Before connecting anything, make sure you know what should plug in where. The diagrams above show all the connector sockets of the 130XE.

There are several sockets through which the 130XE passes and receives information, and one through which it gets the electrical power it needs to operate. The sockets are clearly labelled, but refer to the diagram before plugging anything in.

POWER

The 130XE needs a low voltage DC supply, and this is obtained from the power supply unit supplied with your computer. Plug this power supply into the mains, and plug the output lead into the computer at the socket labelled PWR. IN on the back of the computer. Do not switch on yet!

DISPLAY

The Atari 130XE uses a standard domestic TV to communicate with you, and for this almost any TV will do. To get the best results, use a modern, good quality colour TV. If you use a black and white set, the colour displays produced by the computer will appear as shades of grey.

To connect the 130XE to the TV, plug the supplied aerial lead into the aerial socket of the TV, and plug the other end into the socket labelled TV at the back of the computer (check the diagram). The lead has a different type of plug at each end, so take care that you don't try to force in the wrong one.

CASSETTE RECORDER

The Atari 130XE, like most other small computers, uses cassette tapes to save programs or information, so that you don't have to type them in every time you need them.

You cannot use an ordinary cassette recorder with the 130XE: you must use the special Atari cassette unit, which is available at extra cost. The power for

the cassette unit is taken from the adaptor supplied with it. Plug this adaptor into the mains, and plug the lead into the socket labelled **POWER IN** on the back of the cassette unit. To connect the cassette unit to the computer, plug one end of the lead supplied into the socket marked **PERIPHERAL** on the back of the computer, and the other end into either of the sockets marked **I/O CONNECTORS** on the back of the cassette unit.

DISK DRIVE

Programs can also be stored on floppy disks. A floppy disk drive is faster than a cassette unit and is more flexible in use. (It is also much more expensive). The disk drive connects to the socket labelled **PERIPHERAL** at the rear of the computer. The disk unit requires its own power supply and so must also be plugged into the mains.

PRINTER

If you have a printer, its lead plugs into one of the **I/O CONNECTORS** socket at the rear of the cassette unit. If you are using a disk drive you should plug the printer lead into the spare socket at the rear of the disk drive. The printer also has a separate mains lead.

SWITCHING ON

When you have connected everything together, you are ready to switch on. The equipment must be switched on in the right order, or there is a risk of damaging the computer.

First switch on the TV.

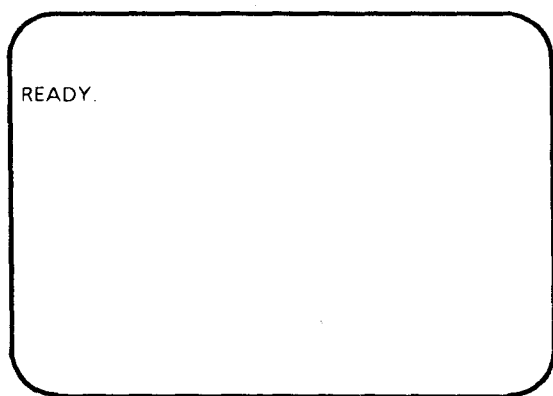
Second, switch on the disk drive and then the printer. Remember that you should never switch a disk unit on or off with disks inside it.

Last, switch on the computer itself, using the switch on the back next to the PWR. IN socket.

Switching off should be carried out in the reverse order: first the computer, then the disk drive and printer, and last of all the TV.

TUNING

To get a display to appear on the TV screen, tune to channel 36, or, on a pushbutton set, use a spare channel and keep tuning until you see this appear on the screen:



If you are unable to tune the television, perform the following checks:

- 1 Check that the aerial lead is connected.
- 2 Make sure the computer is connected to the mains and switched on. The red power indicator should be on.
- 3 Try tuning the TV again, and - if possible - try a different TV.

With a little time and careful tuning, it is possible to get a clear and stable display on nearly all types of TV. If you are unsuccessful, consult your dealer.

If you are thinking of buying a TV especially for use with your 130XE, it's worthwhile taking the computer to the shop, as certain types of TV seem to give better results than others.

For the best quality display, you could buy a monitor. This is a display specifically designed for use with a computer, and contains no circuitry for TV reception. However, a good colour monitor can cost twice the price of the computer!

The 130XE provides a standard output for a monitor, through the socket next to the TV socket. Your dealer can advise you on the connecting leads required.

CHAPTER 3

FIRST STEPS

Before you can make use of your Atari, you must find out how to communicate with it.

Communication is a two-way process: you must give the Atari instructions, and you must be able to find out how it responds to them. You give instructions using the keyboard, and the computer displays its response on the TV screen.

Type the following phrase on the keyboard:

```
PRINT "ATARI XE"
```

You will see the letters appear on the screen as you press the keys. The light blue square – called the *cursor* – will move to indicate where the next letter you type will appear. Nothing else happens though: the computer has not yet obeyed your instruction.

Now press the RETURN key. The words 'ATARI XE' appear on the screen, and the word 'READY' is printed to tell you that the computer is waiting for your next command. If instead of printing 'ATARI XE' the computer prints 0, or prints 'ERROR' and reprints your instruction, it means you have made a typing mistake. Try again!

So, to give the Atari a direct command you type it at the keyboard and press RETURN. Try another command:

```
PRINT "HELLO!"
```

Again, the letters between the quotation marks are printed. You can tell the Atari to print any sequence of letters, but you must remember to put them between quotation marks. Try making it print your name.

You don't have to type **PRINT** in full every time - the command can be abbreviated to **PR.** (the full stop must be included), or to a question mark. Try :

```
PR. "HELLO"
```

and:

```
? "ATARI"
```

(remembering to press **RETURN** after typing each command, of course).

MANAGING THE DISPLAY

As well as typing on to the screen, there are a number of different ways to alter the display.

The Cursor

The cursor can be moved around the screen using the key labelled **CONTROL** at the left of the keyboard and the four keys at the right with arrows marked on them. To move the cursor, hold down the **CONTROL** key and press one of the arrowed keys.

This method of assigning two functions to a key is similar to the use of a **SHIFT** key on a typewriter, except that on the Atari computer it is taken a stage further, with the **CONTROL** key as well as

the SHIFT keys, so any of the other keys may have three different uses.

Try moving the cursor around the screen (it will continue to move for as long as you press the key). Notice that when the cursor reaches one edge of the screen it reappears at the opposite edge.

The TAB key is used in the same way as that on a typewriter. Pressing TAB moves the cursor right to the next preset position. You can set and clear the tab stops by using SHIFT and TAB together to add a new stop at the current horizontal position of the cursor, and CONTROL and TAB together to clear a stop. When you first switch on the Atari there are five TAB stops set automatically, at eight column intervals across the screen.

You will notice after a little experimenting that the TAB key may move the cursor to different positions on successive horizontal lines. This is because, as you will discover later, it is possible to type an instruction which extends over two or even three lines of the screen, and the TAB key is moving to preset points along this 114 character line.

Screen Scrolling

When you move the cursor down to the bottom of the screen with the cursor keys, it will reappear at the top and continue to move downwards. However, if you use the RETURN key to move downwards, when the cursor reaches the bottom the display will begin to move upwards. This movement is called *scrolling*. Anything which disappears from the top of the screen can not be recovered, as the screen will only scroll one way.

Corrections

If you notice - before you press RETURN - that you have made a typing mistake, you can correct the error by using the cursor keys. For example, if you type:

```
PRONT "ATARI XE"
```

and then realise your mistake, position the cursor over the offending O using the cursor keys, and type an I. You can now press RETURN - there's no need to move the cursor to the end of the line. This is a general rule; pressing RETURN will tell the computer to consider the line upon which the cursor rests as a command. This means you must be careful to delete any rubbish from the line before pressing RETURN, or the Atari will not understand the instruction.

To remove large amounts of text, use the DELETE BACK SPACE key which will delete characters to the left of the cursor for as long as it is held down. If you hold down CONTROL and press DELETE BACK SPACE characters to the right of the cursor will be deleted. To insert text in a line, hold down the CONTROL key and press the INSERT key - this will create a space for your additions.

You can delete whole lines from the screen by holding down SHIFT and pressing DELETE BACK SPACE; using SHIFT and INSERT will insert blank lines.

Clearing the screen

This can be done in three ways. You can hold down the SHIFT key or the CONTROL key and press the CLEAR key to simply clear the screen. Pressing the RESET key (in the row at the right hand edge of the keyboard) will clear the screen, return the

screen colour to blue and display the READY message.

NUMBERS

As well as printing words, the Atari 130XE can also handle numbers. Try:

```
PRINT 5          (RETURN)
```

You can do arithmetic:

Addition:

```
PRINT 5+3
```

Subtraction:

```
PRINT 7-4
```

Multiplication:

```
PRINT 3*5
```

Division:

```
PRINT 15/6
```

Powers (exponentiation):

```
PRINT 3^2
```

You can ask the computer to calculate longer expressions, such as:

```
PRINT 3*5 + (11+5)/(9-7) * 3/4
```

(the answer should be 21).

In working out expressions like this, the computer follows strict rules about the order in which the various arithmetical operations are performed.

First The expressions inside brackets are evaluated.

Second Any powers (squares, cubes, etc.) indicated by \wedge are worked out.

Third The computer performs the multiplications and divisions.

Fourth The additions and subtractions are performed.

If you are unsure, put brackets round the bit you want calculated first. Try the following examples, and see if you can work out the answers yourself to check that you know what the computer is doing.

```
PRINT 2 * 2 + 2 * 2
PRINT 2 + 2 * 2 + 2
PRINT 6 + 3 / 5 - 4
PRINT 3 ^ 2 + 4 ^ 3 / 2
PRINT (3 + 4) * 2
```

Both text and numbers can be printed by a single **PRINT** command if they are separated by a semi-colon. For example:

```
PRINT "3+5 = "; 3+5
```

DISPLAYING CHARACTERS

There are several ways in which the characters displayed may be altered.

Upper and Lower Case Letters.

When you first switch on and type, all the letters on the screen are displayed as capitals,(upper case). The Atari can also display letters in lower case.

```
PRINT "THIS IS A TEST"
```

will print the message 'THIS IS A TEST'. Now, press the CAPS key and type the line again:

```
print "this is a test"
```

The Atari will tell you you have made an error because it can not understand commands typed in small letters. You can however print both sizes:

```
PRINT "LARGE AND small letters"
```

is OK. You can use the SHIFT key to get the upper case letters, or press CAPS again to return normal upper case only mode.

Reverse Mode

Up to now, all our characters have been in light blue on a dark blue background. You can invert this by pressing the key at the bottom right of the main part of the keyboard, with the black and white triangle marking - the *reverse* key. From now on, anything you type appears reversed, with dark characters on a light background. To turn this off press the reverse key again. The Atari will not understand instructions typed in reverse video, but reversed characters can be included in **PRINT** commands.

Graphics Characters

The Atari has a set of graphical symbols which can be displayed by holding down the **CONTROL** key

and typing letters. To switch permanently to this mode, press **CONTROL** and **CAPS** together. Any letter key pressed will now display a graphics character. Pressing **SHIFT** and **CAPS** restores the normal upper case mode.

The command:

```
POKE 756, 204
```

will replace these graphics characters with the International character set - a collection of accented letters. The command:

```
POKE 756, 224
```

restores the graphics characters, as does pressing the **RESET** key. The **POKE** command is described in Chapter 10.

CONTROL CHARACTERS

When you use the cursor control keys, or the **TAB** key, or any other of the controlling keys, the effect is usually immediate. It is possible however to write **PRINT** commands which perform the same functions as the control keys. Try the key sequence:

```
PRINT "[ESC][ SHIFT & CLEAR]"
```

(The symbol **[ESC]** means press the key marked **ESC**. Don't type the brackets). Instead of the screen clearing when the **CLEAR** key is pressed, a control character (looking like **↵**) representing the screen clear key is added to the print command. When you press **RETURN**, and this control character is printed, it has the same effect as pressing **CLEAR** usually does.

You can use the ESC key in a similar way to insert cursor controls, INSERT, DELETE BACK SPACE and TAB actions into PRINT commands. The ESC key affects only the next key to be pressed, so to enter a sequence of commands, you must type, for example:

```
PRINT "[ESC][CLEAR][ESC][CONTROL/UP]
```

In the rest of this book we will, where possible, use symbols such as ↑ to represent the typing of ESC and a control character.

If you want to display the control symbols on the screen, pressing ESC twice will produce an ESC control character to precede the control character symbol to be displayed.

MULTIPLE COMMANDS

You can put more than one instruction in one command if you separate the instructions with colons (:). Try this:

```
PRINT "3+4=": PRINT 3+4
```

Or this:

```
PRINT "␣THIS": PRINT "IS": PRINT  
"AN": PRINT "ATARI 130XE": PRINT  
"COMPUTER"
```

It doesn't matter if you run over the end of the line on the screen, as long as there are no more than three screen lines of characters (including the spaces) in the command. A beep will sound to warn you as you approach the maximum number of characters.

Entering a number of instructions together like this can get very cumbersome. In the next chapter

we will discover a way of giving the computer a large number of instructions all at once - in the form of a *program*.

SUMMARY

PRINT instructs the 130XE to print something on the screen.

```
PRINT "ABCDEFGF"
```

prints the characters between the quotation marks.

```
PRINT 3+4
```

prints the result of the sum.

These can be combined:

```
PRINT "3+4*5+7 = "; 3+4*5+7
```

Upper and lower case characters, reversed characters and graphics characters are available.

Control characters to clear the screen and move the cursor can be included in a **PRINT** command.

Multiple Commands

A number of separate instructions may be included on one line but must be separated by colons (:).

CHAPTER 4

PROGRAMMING

At the end of Chapter 3, we saw that it is possible to give the computer a number of instructions at one time by writing them one after another on one line. We will now look at a much more powerful way of doing this - using a *program*.

A computer program is a numbered list of computer instructions which are entered together and stored by the computer to be obeyed later. Let's look at the last example of Chapter 3 and see how we can turn it into a program. In Chapter 3 we had:

```
PRINT " ^ THIS": PRINT "IS": PRINT  
"AN": PRINT "ATARI 130XE": PRINT  
"COMPUTER"
```

Rewritten as a program it would look like this:

```
10      PRINT " ^ THIS"  
20      PRINT "IS"  
30      PRINT "AN"  
40      PRINT "ATARI 130XE"  
50      PRINT "COMPUTER"
```

To enter the program into the computer, type each line in turn, remembering to press RETURN after each. You will see that the Atari does not obey the instructions as they are entered, and that the 'READY' message does not appear after you press RETURN. The computer recognises that each line

is part of a program because each begins with a number, and the program is stored for future use.

When you have typed in all the lines, you can inspect the stored program by typing **LIST** (and pressing **RETURN**, of course). The instructions are listed in numerical order on the screen. You can list parts of the program by specifying line numbers. For example:

LIST 30, 50 lists all lines from 30 to 50

LIST 20 lists only line 20

If a program is very long, it will occupy more lines than there is space for on one screen, so the screen scrolls upwards to display the program. The listing can be suspended by holding down **CONTROL** and pressing **1**. Repeating the **CONTROL 1** sequence will restart the listing.

To order the computer to act on the program you use another **BASIC** command: **RUN**. Type this in (followed by **RETURN**). This tells the computer to read through the stored program and obey each instruction in order. The program gives exactly the same results as the multiple instructions in Chapter 3.

A program may be **RUN** as many times as you like. It will be stored in the computer's memory until you switch it off. Programs can be altered or added to at will. To alter an instruction, retype the line. If you enter:

```
40      PRINT "ATARI"
```

the original line 40 is replaced by the new one.

Extra lines are added by typing them as before:

35 PRINT "AN EXTRA LINE"

Notice that the computer inserts the extra line in the appropriate place: it is *not* put at the end.

NOTE It is a good idea always to number program lines in steps of 10 to allow extra or forgotten lines to be inserted without having to renumber the whole program.

To delete a line, type the line number and press RETURN. The line is deleted from the list in the computer's memory.

Now see if you can alter the example program to **PRINT** different words on the screen.

When you have exhausted the possibilities of this program, you can use the command **NEW** to delete the whole program. You will then be ready for the next program.

This program will demonstrate some of the graphics facilities of the Atari. Don't worry if you don't understand all the BASIC commands, they are explained later in this book.

```
10    GRAPHICS 9
20    SETCOLOR 4,1,0
30    FOR Y=50 TO 0 STEP -10
40    FOR X=0 TO 24
50    LUM = X+3: IF X>11 THEN LUM =
      27-X
60    Y2 = X+Y
70    DIAM = SQR(144-(X-12) *
      (X-12))/2
80    COLOR 15-LUM
90    PLOT Y2, Y+7-DIAM
100   DRAWTO Y2, Y+7+DIAM
110   COLOR LUM
```

```
120    DRAWTO Y2, 180+DIAM-Y
130    NEXT X
140    NEXT Y
200    FOR T=1 TO 1000: NEXT T
210    FOR C=0 TO 15
220    SETCOLOR 4,C,0
230    FOR T=1 TO 500: NEXT T
240    NEXT C
250    GOTO 210
```

ARITHMETIC IN PROGRAMS - VARIABLES

We discovered in Chapter 3 that the Atari will print the answers to arithmetical problems in response to commands such as:

```
PRINT 3+5
```

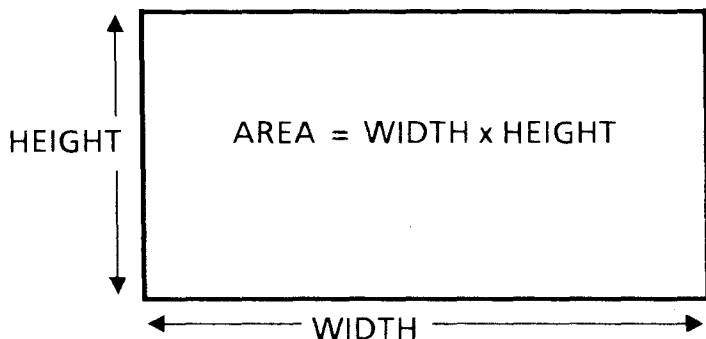
This sort of instruction can be included in a program like this:

```
10  PRINT"↵"
20  PRINT "3+5 = "; 3+5
```

but it would be difficult to make much use of this for working out your personal finances.

What gives a computer the power to perform complex data processing (or 'number crunching') tasks is its ability to do algebra: to use names as symbols for numbers. This means that programs can be written with names to symbolise numbers, and then RUN with different numbers attached to the names. These number-names are called **VARIABLES** because they represent varying numbers. Let's try an example:

The area of a rectangle is equal to the width of the rectangle multiplied by the height. By using



names instead of numbers we can write a program to work out the area of any rectangle:

```
10  WIDTH = 8
20  HEIGHT = 12
30  AREA = WIDTH * HEIGHT
40  PRINT "AREA = "; AREA
```

This program, when **RUN**, will print the area of a rectangle 12 inches by 8 inches. By changing the numbers in lines 10 and 20 we can obtain the area of a rectangle of any size.

VARIABLE NAMES

A variable name can have as many as 114 characters. All variable names must begin with a letter, but the other characters may be numbers, for example A1, EGG7, LAST1. There must not be any spaces in a variable name as this will confuse the computer.

NOTE You cannot use as a variable name any of the BASIC command words, or any name which begins with the letters of a BASIC command word. If you do, the Atari will complain of errors as soon as you type the line. For example, the name LENGTH begins with LEN, and

COST begins with COS. (See Appendix 1 for a full list of all the BASIC reserved words.)

TYPES OF VARIABLE

There are two different types of variable, one represents numbers, the other represents words. The type which represents numbers is called the *numeric* variable; the type representing words is the *string* variable, so called because it contains a sequence or 'string' of characters. The variables in the rectangle program were all numerical variables.

Numeric Variables

These are used to represent numbers and can have any value, whole numbers or fractions. Real variables should be used in all arithmetical programs. Real variables can have any value, for example:

```
REAL = 3.72  
SIZE = 87.3 * 2.5
```

String Variables

A string variable is used to store words, letters or numbers. The variable name for a string variable must be followed by a dollar sign (\$) to distinguish it from the a numeric variable. Examples are:

```
NAME$ = "WINSTON CHURCHILL"  
STRING$ = "ABCDEFGF"
```

Note that the letters defining the variable contents are enclosed by quotation marks.

A string variable can hold almost any number of characters, limited only by the amount of memory

available, but you must set the maximum number of characters allowed for each string variable before using the variable. The **DIM** command (short for *dimension*) is used for this:

```
DIM BONZO$(30)
```

sets the string variable **BONZO\$** to have a maximum length of 30 characters. If you then try to store more than 30 characters in **BONZO\$** an error message will be displayed.

NOTE It is possible to have variables of different types with the same name, such as:

```
NAME    = 87.76
NAME$   = "GEORGE WASHINGTON"
```

The computer will not be confused, but you might, so be careful!

ARRAYS

Numeric variables (but *not* strings) may be used in a special form – an *array*. In normal use a variable name represents one stored number. In an array, the variable name represents a collection of stored information, with one or more reference numbers identifying the individual items. An array can be pictured as a collection of boxes. The table on the next page is a diagram of an array which uses one reference number.

The array has 6 elements numbered from 0 to 5. The array would be used like this:

```
10    DIM BOXES(5)
20    BOXES(0) = 11
30    BOXES(1) = 92.73
40    BOXES(2) = 6
```

REFERENCE NUMBER	CONTENTS
0	11
1	92.73
2	6
3	0.4971
4	-3
5	125.6

The Array BOXES

```
50    BOXES(3) = 0.4971
60    BOXES(4) = -3
70    BOXES(5) = 125.6
100   FOR N = 0 TO 5
110   PRINT N, BOXES(N)
120   NEXT N
```

As mentioned before, an array may have more than one reference number. The number of reference numbers per item is called the number of *dimensions* of the array. Here is a program using an array with two reference numbers per item:

```
10    DIM TABLE(2,1)
20    TABLE(0,0) = 34
30    TABLE(0,1) = 3.7
40    TABLE(1,0) = 7
50    TABLE(1,1) = 6.85
60    TABLE(2,0) = 302
70    TABLE(2,1) = 0.35
100   FOR A = 0 TO 2
110   FOR B = 0 TO 1
120   PRINT TABLE(A,B),
130   NEXT B
```

```
140 PRINT  
150 NEXT A
```

Here again the array can be thought of as a collection of storage boxes, this time laid out in a grid as shown in the table below. The two columns could represent for example the number of items in stock, and the price of each item.

	B = 0	B = 1
A = 0	34	3.7
A = 1	7	6.85
A = 2	302	0.35

The Array TABLE

You can use arrays with one or two dimensions, but no more. The maximum allowed value of each reference number may be anything up to several thousand, but this depends on how much memory is occupied by the program itself, and by the other variables of the program.

A **DIM** command must be included at the beginning of any program which will use arrays. The command tells the computer to reserve memory space for the array. If you try to use an array without setting its dimensions the Atari will give the error message 'ERROR 9'.

GETTING VALUES INTO PROGRAMS

It would be very inconvenient to have to alter a program to make it handle different numbers, so there are a number of instructions which allow

numbers or letters to be given to a program while it is running. The first of these is **INPUT**.

When the computer finds an **INPUT** statement in a program, a question mark is displayed on the screen. The computer waits for you to type in a number or letter string, which it will then store as a variable before continuing with the rest of the program. Type **NEW (RETURN)** to delete any program currently in the computer's memory and then try this example:

```
10 INPUT NUMBER
20 PRINT NUMBER
```

When you **RUN** this program you will see a question mark on the screen. Type a number and press **RETURN**: the number is printed.

The **INPUT** instruction can handle strings too, if you specify a string variable in the **INPUT** command:

```
10 DIM NAMES$(20)
20 INPUT NAMES$
30 PRINT NAMES$
```

This program will accept both numbers and letters and store them as a string variable. Pressing **RETURN** enters a null string but this time nothing is displayed.

It is possible to use one **INPUT** command to input two or more numbers or strings.

```
10 DIM NAMES$(20)
20 INPUT NAMES$, AGE
30 PRINT NAMES$
40 PRINT AGE
```


The variables must have commas separating them in the **INPUT** command. If you press **RETURN** before entering all the items the 130XE will print another question mark and wait for the remaining items. When you type in the information you must use a comma to separate numeric entries, or press **RETURN** to mark the end of each string entry (a comma would be counted as part of the string). If you enter too many items the surplus ones will be ignored.

Let's use what we know about variables and **INPUT** to improve the area program. Clear the computer's memory by typing **NEW (RETURN)**, and then type in this program.

```
10      REM  IMPROVED AREA PROGRAM
20      PRINT "↵"
30      PRINT "ENTER WIDTH";
40      INPUT WIDTH
50      PRINT "ENTER HEIGHT";
60      INPUT HEIGHT
70      AREA = WIDTH * HEIGHT
80      PRINT: PRINT "AREA = "; AREA
```

This program will read the two numbers you type in for width and height and print the area of the rectangle. Using **INPUT** has made the program much more flexible; we don't have to alter the program to use different numbers.

REMARKS

Line 10 of the last program is a *remark* or *comment* statement. These are used to hold comments, notes and titles to make the purpose of a program and the way in which it works clear to someone reading the listing. Remarks are identified by the word **REM** before the remark and have no effect when the program is **RUN**.

You should always put plenty of remarks in programs, because although you may understand how a program works when you write it, three months later you will have forgotten. If you need to modify a program at some time after writing it, remarks will make it much easier to remember how the program works.

SUMMARY

PROGRAMS

A *program* is a numbered sequence of computer instructions.

- | | |
|-------------|---|
| LIST | displays the lines of a program. |
| RUN | starts the execution of a program. |
| NEW | deletes a program from the computer's memory. |

VARIABLES

Variables represent numbers and words in programs.

There are two types of variable:

- | | |
|----------------|--|
| Numeric | representing numbers. |
| String | representing words or numbers, and distinguished by \$ after the name. |
| INPUT | is used to enter numbers or words into variables while a program is running. |

Numeric variables may take the form of *arrays*, with one variable name representing a number of different values, with reference numbers to distinguish between the items.

REMARKS

REM is used to put remarks into programs for the programmer's benefit.

CHAPTER 5

PROGRAM CONTROL

In the last chapter we defined a program as a numbered list of instructions to the computer, which are obeyed in order from beginning to end. In this chapter we will find out how to write programs in which some instructions are executed more than once. This makes programs more efficient, as instructions which must be repeated need to be written only once. We will also discover that the computer can make decisions.

REPETITION

A section of program can be repeated many times using the instructions **FOR** and **NEXT**. Try this example :

```
10   FOR COUNT = 1 TO 10
20   PRINT COUNT
30   NEXT COUNT
```

which prints the numbers from 1 to 10 on the screen. The section of program between the **FOR** and the **NEXT** commands is repeated for each value of **COUNT**, and **COUNT** is automatically increased by one every time the **NEXT** command is met in a loop.

The variable in a **FOR ... NEXT** loop need not be increased by 1; any increase or decrease can be

specified using the command **STEP**. Try changing line 10 to:

```
10   FOR COUNT = 0 TO 30 STEP 3
```

Then **RUN** the program again. **COUNT** is now increased by 3 each time the loop is repeated. The **STEP** can also be negative, so that the numbers get smaller:

```
10   FOR COUNT = 50 TO 0 STEP -2.5
```

In summary, the instructions are used like this:

```
FOR V = X TO Y STEP Z
```

is used to begin a loop (V is a variable and X, Y and Z may be variables or numbers), and

```
NEXT V
```

marks the end of the loop. Note that, unlike some other microcomputer BASICs, in Atari BASIC the variable *must* be specified after **NEXT**.

The variables used in **FOR ... NEXT** loops must be simple numeric variables; string variables, and elements of arrays may not be used.

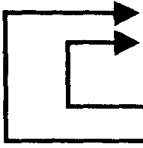
Nested Loops

FOR ... NEXT loops can be *nested* - that is, one loop can contain one or more other loops. Here is an example :

```
10   FOR A = 1 TO 3
20   FOR B = 1 TO 4
30   PRINT A, B
40   NEXT B
```


50 NEXT A

For any nested loops, it is *very important* that the enclosed loop is *completely within* the outer loop, otherwise the program *will not work*. To clarify this, let's examine the order in which instructions are obeyed in the program.



```
START
10 FOR A = 1 TO 5
20 FOR B = 1 TO 5
30 PRINT A, B
40 NEXT B
50 NEXT A
END
```

The two loops are arranged so that one is completely within the other. If lines 40 and 50 were swapped over, the result would be as shown in the next example. You can see that the two loops



```
START
10 FOR A = 1 TO 5
20 FOR B = 1 TO 5
30 PRINT A, B
40 NEXT A
50 NEXT B
END
```

overlap and the order in which the instructions should be obeyed is not clear, and the Atari will give 'ERROR 13'.

This rule always applies to **FOR ... NEXT** loops - the loops must *always* be one within the next.

JUMPS

The computer can be instructed to jump from one program line to another using the command **GOTO**. Enter this program:

```
10      GOTO 30
20      PRINT "LINE 20"
30      PRINT "LINE 30"
```

When it is **RUN** the message 'LINE 20' is not seen because the **GOTO** instruction in line 10 sends the Atari straight to line 30.

The jump can be to a lower numbered line. The following program will go on for ever, unless you stop it by pressing the **BREAK** key:

```
10      PRINT "ON AND ";
20      GOTO 10
```

This is called an endless loop, and is to be avoided!

DECISIONS

The commands **IF** and **THEN** are used in programs to make decisions. A variable is tested, and one of two alternative actions is taken, depending on the result of the test.

```
10      FOR X = 1 TO 10
20      PRINT X
30      IF X = 5 THEN PRINT "X = 5"
40      FOR P = 0 TO 200 : NEXT P
50      NEXT X
```

The format of **IF ... THEN** is:

IF (*condition*) **THEN** (*instructions*)

The condition after **IF** may be one of many possible alternatives. Examples are :

```
IF COUNT = 10
```

Continue when variable (COUNT) equals a number, in this case 10.

```
IF COUNT < 100
```

Continue when variable less than a number.

```
IF COUNT > 100
```

Continue when variable greater than a number.

```
IF NUMBER < > VALUE
```

One variable not equal to another (greater than or less than).

```
IF X >= Y
```

Greater than or equal to.

```
IF X <= Y
```

Less than or equal to.

Two conditions may be combined, as in:

```
IF X = 1 OR X = 2
```

```
IF A = 3 AND NAME$ = "ATARI 130XE"
```

In all of these, the items being compared may both be variables, or one may be a variable and the other a number. (There's not much point in comparing two numbers!) For further details see Chapter 9, Logical Thinking. The instructions after **THEN**

are carried out if the condition is met, otherwise the program continues at the next line.

IF ... THEN can also be used to control jumps:

```
10    PRINT "WHAT NUMBER AM I  
      THINKING OF";  
20    INPUT N  
30    IF N < > 3 THEN PRINT  
      "WRONG";GOTO 10  
40    PRINT "CORRECT!"
```

If the jump is the only instruction after **THEN**, the word **GOTO** can be omitted:

```
10    DIM A$(20)  
20    PRINT "WHAT AM I";  
30    INPUT A$  
40    IF A$ = "ATARI 130XE" THEN 80  
50    PRINT: PRINT A$;"? NO, TRY  
      AGAIN"  
60    PRINT  
70    GOTO 20  
80    PRINT: PRINT "GOOD GUESS!"
```

Variables can be used to direct a **GOTO** command:

```
10    JUMP = 40  
20    GOTO JUMP  
30    PRINT "THIS LINE IS MISSED"  
40    PRINT "JUMPS TO HERE"
```

EXAMPLE PROGRAM - SORTING NUMBERS

As an example of what can be done using loops and decisions, here is a program which sorts ten numbers into ascending order.

```
10    REM    SORTING PROGRAM
```

```
20    DIM NUM(10),SORT(10)
30    REM INPUT 10 NUMBERS
40    PRINT "↵"
50    FOR N=1 TO 10
60    PRINT "ENTER A NUMBER";: INPUT
      NUMBER
70    NUM(N)=NUMBER
80    NEXT N
100   REM COPY NUMBERS TO ARRAY SORT
110   FOR C=1 TO 10
120   SORT(C) = NUM(C)
130   NEXT C
200   REM SORT NUMBERS
210   COUNT = 0
220   FOR N=1 TO 9
230   IF SORT(N+1) >= SORT(N) THEN
      280
240   TEMP = SORT(N+1)
250   SORT(N+1) = SORT(N)
260   SORT(N) = TEMP
270   COUNT = COUNT + 1
280   NEXT N
300   IF COUNT > 0 THEN 210
310   FOR Z=1 TO 10
320   PRINT NUM(Z),SORT(Z)
330   NEXT Z
```

The **REMARKS** indicate the functions of the different sections of the program, which works like this:

Line 20 dimensions the two arrays used in the program.

Lines 40 to 80 input ten numbers from the keyboard and store them in the array **NUM()**, using a **FOR ... NEXT** loop.

Lines 110 to 130 copy the numbers from **NUM()** to a second array **SORT()** which will be sorted, while **NUM** retains the numbers in the order in

which they were typed in. Again, a **FOR ... NEXT** loop is used.

The lines from 210 to 280 sort the numbers in the array **SORT(10)** into ascending order. A **FOR ... NEXT** loop compares each element of the array **SORT** with the next, and swaps them over if they are in the wrong order. A variable, **COUNT**, keeps track of the number of these swaps and, if at the end of the loop this isn't zero, the loop is repeated.

Lines 310 to 330 complete the program by **PRINTing** the two sets of numbers.

PAUSES

To make a pause during the running of a program, we can use a **FOR ... NEXT** loop which does nothing at all except count its way through the steps. The following program is an example :

```
10    FOR T = 0 TO 5000
20    NEXT T
30    PRINT "ABOUT 10 SECONDS"
```

Another way to make a pause is to use the Atari's built in timer. The three storage locations 18, 19 and 20 are incremented about 5 times a second, so for a 10 second wait, try:

```
10    POKE 19,0:POKE 20,0
20    IF PEEK(19) < 2 THEN 20
30    PRINT "[BELL]"
```

(The **BELL** character is obtained by pressing **ESC** then holding down **CONTROL** and pressing 2.)

SUBROUTINES

A subroutine is a section of program which is executed at a number of different times in the course of a program, but is written only once. The computer is diverted from the main program to the subroutine, and returns to the main program at the point from which it left.

The command **GOSUB** followed by a line number or a variable diverts the computer to the subroutine in much the same way as the **GOTO** command. The difference is that the end of a subroutine is marked by a **RETURN** command which sends the computer back to the instruction after the **GOSUB** command. As an example of two very simple subroutines, type in the following program:

```
10      GOSUB 110
20      C = A+B
30      GOSUB 210
40      END
100     REM SUBROUTINE TO INPUT A & B
110     INPUT A
120     INPUT B
130     RETURN
200     REM SUBROUTINE TO DISPLAY
      RESULT
210     PRINT "  "
220     PRINT
230     PRINT A;" + ";B;" = ";C
240     RETURN
```

Lines 10 to 40 are the main program. Line 10 calls the subroutine at line 110, line 20 adds A and B and stores the result as C, and line 30 calls the subroutine at line 210. Line 40 marks the end of the program - as the last line to be executed is not the last line of the program we have to tell the

Atari not to go on to the following lines, which contain the subroutines.

The subroutine at line 110 inputs two numbers and stores them as A and B. The subroutine at line 210 clears the screen and prints the two numbers and their sum.

When the program is **RUN**, the computer begins, as always, at the lowest numbered line, line 10. This line calls the first subroutine, and the computer is diverted to line 110. Line 130 returns the computer to the instruction after the **GOSUB**, which is in line 20. The program proceeds as normal to line 30, which causes another diversion to line 210. Line 240 returns the computer to line 40, and the program ends.

So, a **GOSUB** command causes a diversion from the sequence of a program to a subroutine, and the **RETURN** command ends the diversion.

The use of subroutines saves a lot of effort in writing programs, as the program lines in the subroutine need to be written only once, instead of being retyped at every point in the program where they are needed. There is no limit to the number of times a subroutine may be called.

Another advantage of subroutines is that they can make the design of a program simpler. If you use subroutines for the repetitive and less important parts of a program the main program becomes much easier for you, the programmer, to follow, and is therefore much easier to write.

Subroutines, like loops, may be nested – that is, the subroutines may call other subroutines, which may in turn call others. However, a subroutine may not be called by another subroutine which it has itself called, or an endless loop occurs. The program will

crash (come to an undignified halt) when the Atari runs out of the memory it uses to store all the line numbers for the **RETURNS**, and the message 'ERROR 2 AT LINE xx' will be displayed.

STOPPING AND STARTING

In the last example program the new command **END** was used to indicate the last line of the program to be executed. The command tells the computer to stop running the program. There is a second command, **STOP**, which also halts programs, but the two have slightly different effects. **END** stops the program running, and the 'READY' message is displayed, but when **STOP** is used the program halts and the message 'STOPPED AT LINE xxx' is printed (xxx is the line number of the **STOP** instruction). Unless you need to know where the program halted, **END** is the one to use.

If a program has been halted by one of these instructions, or by pressing the **BREAK** key, it may be restarted with the instruction **CONT**. The program will continue from the instruction after the last one to have been executed. **CONT** may only be used immediately after the program has been stopped. You may not use **CONT** within programs.

ON ... GOSUB (GOTO)

Both **GOSUB** and **GOTO** may be used in a second type of decision command which selects one of a number of destinations depending on the value of a chosen variable.

```
ON N GOTO 100, 200, 300
```

results in:

```
GOTO 100    if N = 1
GOTO 200    if N = 2
GOTO 300    if N = 3
```

Similarly:

```
ON P GOSUB 500,200,500,400,100
```

selects the P'th destination in the list and calls it as a subroutine. If the value of the variable is greater than the number of destinations in the list, or if it is zero, no destination is selected and the program continues at the next instruction. If the value is less than zero, the program stops with the error message 'ERROR - 3'. To avoid this, make sure you only use positive values by testing them first. For example, instead of:

```
250    ON NUM GOTO 1000, 2000, 3000
```

write:

```
250    TEST=NUM
260    IF TEST<0 THEN TEST=0
270    ON TEST GO TO 1000, 2000, 3000
```

SUMMARY

LOOPS

The loop structure has the form:

FOR V = A TO B STEP C ... NEXT in which the instructions between the **FOR** and **NEXT** commands are repeated once for each value of V indicated by the **FOR ... STEP** command. A, B and C may be variables or numbers; V must be a variable. Loops may be nested one within another.

DECISIONS

IF (*condition*) **THEN** (*instruction*) decides between two actions. The condition after **IF** is usually a test of a variable. There may be more than one instruction after **THEN**, in which case colons (:) must be used to separate them. The complete **IF ... THEN** command must be in one line.

GOTO need not be included if it would be the only command after **THEN**, but it must be if there is another command before it:

```
100 IF ACE = 1 THEN 20
```

```
but      100 IF ACE = 2 THEN KING =  
          7:GOTO 20
```

SUBROUTINES

Diversions from a program, using the commands **GOSUB** and **RETURN**. **GOSUB** calls the subroutine, and **RETURN** at the end of the subroutine returns the computer to the instruction after the **GOSUB**.

ON ... GOSUB and ON ... GOTO

Select a destination from a list:

```
ON P GOSUB X, Y, Z
```

causes a **GOSUB** to the P'th item in the list.

```
ON N GOTO A, B, C
```

This causes a jump to the N'th destination - if there is an N'th item in the list.

END halts the running of a program.

STOP halts a program and prints 'STOPPED AT LINE xxx'.

CONT will restart a program.

CHAPTER 6

DATA AND PROGRAMS

So far we have not examined all of the ways in which information can be given to programs, nor those by which programs can print out information on to the screen. There are several ways of getting information into programs which we will examine in this chapter, and we will also examine in more detail the use of **PRINT** and other commands to display data.

READ AND DATA

To enter large amounts of data which will be the same each time the program is **RUN** we use the data entry command **READ**. Look at this program:

```
10    DIM STOCKS(10),PRICE(10)
20    FOR ITEM=1 TO 10
30    READ DTA1,DTA2
40    STOCKS(ITEM)=DTA1
50    PRICE(ITEM)=DTA2
60    NEXT ITEM
70    PRINT"↵"
80    FOR X=1 TO 10
90    PRINT X,STOCKS(X),PRICE(X)
100   NEXT X
110   END

120   DATA 5,1.41,1,5.75,10,2.63,
        100,9.95,0,2.23,5,5.6,123,
        1.49,8,9.12,20,5.55,1,6.59
```

This method of entering information is much easier to use than simply writing it into the program by setting variables. Imagine the typing involved in entering lots of lines like:

```
PRICE(2)=5.75
```

With **DATA** lines you can lay out the information in a clear tabular form and check it much more easily. You can also alter it if you need to without touching the main program.

When a program is running, the computer uses a 'pointer' stored in its memory to keep track of how many **DATA** items it has read. Every time a **READ** command is met, the **DATA** item indicated by the pointer is copied into the variable, and the pointer moves on to the next item. This means you must have a **DATA** entry corresponding to each occurrence of **READ**. The program will stop if there are too many **READ** commands and the Atari will print an error message: **ERROR-6**.

The command **RESTORE** can be used to reset the data pointer to the first **DATA** item in the program. **RESTORE** followed by a line number, such as:

```
50      RESTORE 2000
```

will reset the data pointer to the first **DATA** item which occurs after that line.

The variables used in the **READ** command must match the type of data in the corresponding **DATA** entry. A **READ** with a string variable will read anything as a string, but **READ** with a number variable must find a number in the **DATA** entry or the error message **ERROR-6** will result.

GET

The **GET** instruction provides a second way of feeding information to programs from the keyboard. **GET** is used in handling many different peripheral devices, but here we will consider it only as a keyboard command. (See Chapter 15 for more details on other uses of **GET**).

GET halts the program and waits for a key to be pressed. The ASCII code for the key character is stored in a variable.

Before **GET** may be used, a channel must be opened to the keyboard, by the command

```
OPEN#1,4,0,"K:"
```

The **GET** instruction then takes the form:

```
GET#1,KEY
```

Try this short program to see **GET** in action.

```
10      REM READING KEYS WITH GET
20      OPEN#1,4,0,"K:"
30      GET#1,KEY
40      PRINT KEY,CHR$(KEY)
50      GOTO 30
```

GET is not the ideal command for taking large amounts of information from the keyboard. It is very useful to be able to halt a program until a key is pressed, before, for example displaying a new screen of information. **GET** also allows one-key entries to be made when selecting items from *menus* - lists of options. Try this program:

```
10      REM MENU
20      OPEN#1,4,0,"K:"
```

```
30      PRINT "↖ ↓ ↓ ↓ SELECT OPTION":REM  
        REVERSED TEXT  
40      PRINT " ↓ RED"  
50      PRINT " ↓ YELLOW"  
60      PRINT " ↓ BLUE"  
70      GET#1,KEY  
80      IF KEY=ASC("R") THEN SETCOLOR  
        2,3,6  
90      IF KEY=ASC("Y") THEN SETCOLOR  
        2,2,8  
100     IF KEY=ASC("B") THEN SETCOLOR  
        2,9,4  
110     GOTO 30
```

This kind of menu is useful in many types of program when asking the program user to make simple decisions and selections. The program is made extremely easy to use as keystrokes are reduced to a minimum.

MORE ABOUT PRINTING

So far, we have described the use of **PRINT** (or ?) simply to display single items of data on the screen. The Atari has a number of extra facilities which allow you to control the screen layout and produce clear and orderly output from programs.

Punctuating **PRINT** Statements

You can **PRINT** more than one item on a single line by including 'punctuation' in the **PRINT** command. You can use commas to separate the items, in a **PRINT** statement, and the effect is rather like the 'tab' function of a typewriter. Output on the screen is formatted at every tenth column. For example:

```
PRINT "A", "B", "C", "D"
```

will give a display:

A	B	C	D
---	---	---	---

The first item in the **PRINT** statement appears in the first column. The next item – the first after a comma – appears in column 10. The other items appear in columns 20 and 30.

The semicolon (;) can be used in a similar fashion. This leaves no space between successive items in the **PRINT** statement.

```
PRINT "A"; "B"; "C"; "D"; "E"; "F"
```

will display:

```
ABCDEF
```

The same rules apply when printing numbers, and numerical variables.

The effect of commas and semicolons is not confined to the **PRINT** statement in which they appear. If you end one **PRINT** statement with a comma or semicolon then the data printed by the next **PRINT** command will appear on the same line, with spaces as appropriate.

POSITION

The command **POSITION** can be used to make the Atari begin printing at any point on the screen. To place the cursor at column C, row R, use:

```
POSITION C,R
```

On the standard text display, C must be between 0 and 39, and R must be between 0 and 23.

Cursor Control

Just as we can use the cursor keys to move the cursor round the screen, so can we use a program to move it. As described in Chapter 3, the Atari interprets four characters as control codes governing the cursor movement.

The characters can be used within strings. For example:

```
10    DIM CD$(20): DIM CU$(20)
20    PRINT "↖"
30    PRINT "TOP LINE"
40    CD$ = "↓↓↓↓↓↓↓↓↓↓"
50    CU$ = "↑↑↑↑↑↑↑↑↑↑"
60    FOR T = 0 TO 500:NEXT T
70    PRINT CD$; "MIDDLE LINE"
80    FOR T = 0 TO 500:NEXT T
90    PRINT CU$;"BACK TO THE TOP!";
      CD$
```

prints the three messages in various parts of the screen using the strings CD\$ and CU\$ to move the cursor.

You can use the other control characters in a similar way - to insert or lines and characters, to set and clear tabs, and so on.

The way in which data is presented on the screen is an important part of the program and can make the difference between a program being easy to use, or frustrating and confusing. Make good use of the commands available!

SUMMARY

READ and **DATA** are used to copy large amounts of information into variables without writing lots of program lines.

POSITION C,R places the cursor at column C, row R.

There are a number of special commands which allow control of the format of data **PRINTed** on the screen.

CHAPTER 7

PIECES OF STRINGS

String variables are used to store sequences - 'strings' - of characters. There are a number of BASIC commands which are used to manipulate strings, and these are described in this chapter.

We saw in Chapter 4 that a string variable can store a sequence of characters. For example:

```
NAME$    = "WILLIAM SHAKESPEARE"  
GAME$    = "SPACE INVADERS"  
REFERENCE$ = "ABC123D"
```

A string can almost any number of characters, limited only by the amount of memory space available. The maximum length of any string must be specified in a **DIM** statement before the string is used. The characters in a string may be letters, figures, punctuation marks, spaces - in fact any of the characters the Atari can print. The number of characters in a string can be counted using the function **LEN**. Try this:

```
10  DIM NAME$(25)  
20  NAME$ = "JOHN SMITH"  
30  PRINT NAME$  
40  PRINT LEN(NAME$)
```

This program prints the length as 10, which is the number of characters in **NAME\$** (including the space).

NUMBERS AND LETTERS

Computers cannot handle characters (numbers, symbols and letters) directly – they use numbers to represent the characters. Each letter, and each of the other characters the 130XE can print, is represented by a different code number. There are several systems used in different computers for deciding which code number represents which letter. The Atari uses a system called ASCII – the American Standard Code for Information Interchange – which is the most common system for micros. The table in Appendix 4 shows how the ASCII code relates to letters and numbers. (In fact the Atari code differs from standard ASCII in certain respects, but this is not too important.)

From the table we see that the 130XE uses a code number 65 to represent the letter A, code number 66 for B and so on. Even numerals have codes. For example the character '9' has the code number 57.

There are two functions in BASIC which allow us to convert characters to numbers and numbers to characters.

The function **CHR\$** converts a number to a string containing the corresponding character:

```
10    DIM A$(1)
20    C = 65
30    A$ = CHR$(C)
40    PRINT A$
```

This displays the letter 'A'. You can use a variable with **CHR\$**, as in the example, or a number:

```
PRINT CHR$(42)
```

This displays an asterisk (*).

CHR\$ can only give one character at a time. There are 255 possible characters, and if the number or variable in the brackets is increased beyond 255 the character set is repeated. The value used must be less than 65536, and must not be negative, or an **ERROR-3** message will result.

The function **ASC** complements **CHR\$**. **ASC** finds the ASCII codes of characters.

```
10      B = ASC("B")
20      PRINT B
```

displays '66', which is the ASCII code for the letter B. You can also use string variables with **ASC**:

```
10      DIM B$(1)
20      B$ = "B"
30      PRINT ASC(B$)
```

also displays '66'. If the variable contains more than one character, **ASC** gives the code for the first character only:

```
PRINT ASC("ABCDE")
```

displays '65', the code for 'A'.

FIGURES IN STRINGS

A string can contain any characters - including numerals. There are two functions which can be used to make strings of number characters from actual numerals and to find the numeric value of the number characters in a string.

The function **STR\$** creates from a number a string containing the characters of that number:

```
10      DIM A$(5)
20      A$ = STR$(12345)
```

```
30      PRINT A$
```

This converts the single number 12345 into a string by assigning the characters 12345 to A\$ and displays the string.

The complementary function to **STR\$** is **VAL**. **VAL** evaluates the numerical characters in a string:

```
10      DIM A$(10)
20      A$ = "123456"
30      A = VAL(A$)
40      PRINT A
```

This displays the number '123456'.

If the string evaluated contains letters as well as numbers then only the numbers which appear to the left of all the letters are converted by **VAL**. This means that

```
PRINT VAL("123ABC45")
```

displays '123'.

The signs + and - may appear before the number characters. They will be treated by **VAL** as the sign of the number. If the first characters of the string are not numeric the message **ERROR-18** is displayed by the Atari.

TESTING STRINGS

Strings can be compared with each other, in the same way as numbers, using **IF ... THEN**. A routine like this one can be used to check that an entry is suitable for the program:

```
10      DIM PW$(50)
```

```
20    PRINT "WHAT'S THE PASSWORD";:
      INPUT PW$
30    IF PW$ <> "BEANS" THEN PRINT
      "WRONG !!!":GOTO 20
40    PRINT "↵O.K."
```

As well as testing to see if two strings are the same, you can use the 'greater than' (>) and 'less than' (<) tests to compare strings.

```
10    IF "B" > "A" THEN PRINT "OK"
```

This works because the letters are stored as numbers. The computer is in fact comparing the ASCII codes of the letters in the strings. This is very useful because it allows strings to be sorted into alphabetical order, as we will show later in this chapter.

STRING MANIPULATION

Atari BASIC differs from most other dialects of BASIC in the way in which strings are handled. Characters can be copied from strings, and can be inserted into strings to replace some of the characters already there. To illustrate the various commands we will use a string A\$ which contains the first 10 letters of the alphabet. The string is defined by:

Position	1	2	3	4	5	6	7	8	9	10
Character	A	B	C	D	E	F	G	H	I	J

The string A\$

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

```
A$="ABCDEFGH IJ"
```

Portions of strings can be copied by specifying the positions of the beginning and end of the substring to be removed, like this:

```
B$=A$(FIRST, LAST)
```

So to print the fifth to eighth characters of A\$, use the command:

```
PRINT A$(5,8)
```

New characters may be inserted into a string by specifying the position of the first character:

```
A$(4)="XYZ"  
PRINT A$
```

Notice that end of the string is reset to the end of the new section, despite the fact that the original string was longer. The end of the first string can be recovered by adding a character to the end:

```
A$(11)="Q"  
PRINT A$
```

You can add material to the end of a string by this method as long as the total length does not exceed the maximum set by the **DIM** command. A useful formula to add one string to the end of another is:

```
A$(LEN(A$)+1)=B$
```

It is quite legitimate to add a new string at a position beyond the end of the first string. Try adding more letters to A\$ like this:

```
A$(20)="LMNOP"  
PRINT A$
```

The characters in the middle of the string, which have not been defined by any of your commands, turn out to be hearts. The character code of a heart is zero, and the memory is filled with zeros when the Atari is switched on.

The facility to manipulate strings in this way and the fact that a string may hold a very large number of characters allows a single string variable to be used in a similar fashion to a numeric array variable. An "array" of ten strings, each with a length of 20 characters or less, can be fitted into one string with a length of 200 characters by allowing 20 character positions for each of the substrings. The substrings can then be read and replaced using the commands outlined above.

There are two problems in using strings like this. The first is that the string is initially filled with hearts. This can be overcome by filling the string with spaces before adding any data to it:

```
10    DIM ARRAY$(200)
20    FOR Z=1 TO 200
30    ARRAY$(Z)=" "
40    NEXT Z
```

The second problem is that every time new material is inserted into a string, the position of the end of the main string is reset to the position of the end of the inserted portion. The answer to this is to write a space in the last character space of the main string after every modification:

```
150    ARRAY$(140)="NEW STUFF"
160    ARRAY$(200)=" "
```

There are one or two other points to watch out for. Garbage characters may be left behind if you replace an element with a shorter string. This again can be solved by writing spaces into the

appropriate portion of the string before inserting the new material.

The program below uses these string handling techniques to sort 20 words into alphabetical order. The string ALL\$ with a maximum length of 440 characters is used as the array.

```
10      REM STRING SORTING PROGRAM
20      PRINT "↵"
30      DIM WORD$(20),ALL$(440)
40      FOR L=1 TO 440: ALL$(L)=" ":
      NEXT L
50      REM INPUT 20 WORDS
60      FOR C=1 TO 20
70      PRINT "ENTER A WORD";
80      INPUT WORD$
90      ALL$(C*20)=WORD$
100     NEXT C
110     ALL$(440)=" "
120     REM PRINT UNSORTED LIST
130     GOSUB 1000

190     REM SORT THE WORDS
200     SWAPS=0
210     FOR Z=1 TO 19
220     Z0=Z*20
230     Z1=Z0+20
240     IF ALL$(Z1,Z1+19) >=
      ALL$(Z0,Z0+19) THEN 300
250     WORD$=ALL$(Z1,Z1+19)
260     ALL$(Z1)=ALL$(Z0,Z0+19)
270     ALL$(Z0)=WORD$
280     ALL$(440)=" "
290     SWAPS=SWAPS+1
300     NEXT Z
310     IF SWAPS>0 THEN 200: REM
      REPEAT UNTIL NO SWAPS
320     REM PRINT SORTED SEQUENCE
330     GOSUB 1000
```



```
340      END

990      REM DISPLAY STRINGS
1000     PRINT"↵"
1010     FOR P=1 TO 20
1020     PRINT ALL$(P*20, P*20+19)
1030     NEXT P
1040     RETURN
```

This program works in the same way as the sorting program at the end of Chapter 5. You can input twenty strings, which may contain more than one word.

The "array" string ALL\$ is not used very efficiently. The first 19 characters are not used at all. There is no reason why, with a bit of extra calculation, these characters should not be used. Remember that there is no character at ALL\$(0) – if you try to use this position the message ERROR-9 will be displayed and the program will stop.

If you run this program a few times you will discover how effective string sorting can be. Notice that any characters will be sorted into ASCII order, so strings containing characters such as */@£?% and /@\$&)# will be sorted. You will find that lower case letters are treated as completely different from upper case letters, because of course they are represented by different codes, and the Atari is interested only in the codes. The lower case letters have higher codes than the capitals, so they appear at the end of the sorted list. (You can switch in and out of lower case by pressing the CAPS key).

SUMMARY

A string is a sequence of characters. These can be stored in string variables, which are distinguished by \$ after the variable name.

LEN(string) gives the number of characters in a string.

ASCII code is used by the Atari to symbolise letters. There is a table of the ASCII codes for the characters in Appendix 10.

CHR\$ converts a code number to the equivalent character:

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

displays 'A'.

ASC gives the ASCII code for a character:

```
PRINT ASC("A")
```

displays '65'.

STR\$ turns a number into a string of number characters:

```
10   A$ = STR$(123)
```

```
20   PRINT A$
```

VAL evaluates the numerical characters in a string:

```
PRINT VAL("123ABC789")
```

displays '123'.

COMPARISONS.

Strings can be compared using **IF ... THEN** and the relational operators **=** **<** and **>**. This is useful for testing inputs and sorting strings.

Strings can be modified by inserting characters at any position. Substrings can be read by specifying the beginning and end of the portion required.

CHAPTER 8

FUNCTIONS

A 'function' in computing is an instruction which performs a calculation on a number. There are a number of functions available on the Atari 130XE. For example, in Chapter 7 functions were described which operate on numbers to give strings, or on strings to give numbers. In this chapter we will look at some more of the functions available on the Atari.

SQUARE ROOTS

The square root of a number is calculated by the function **SQR(N)**. The square root of a number or variable N is the number which when multiplied by itself, or *squared*, gives N. Try:

```
PRINT SQR(4)
```

The Atari displays 2, because 2 squared ($2*2$) is 4.

```
10 FOR N = 1 TO 10
20 PRINT N, SQR(N)
30 NEXT N
```

prints the numbers 1 to 10 and their square roots.

ABSOLUTE VALUES

The function **ABS** finds the *absolute value* of a number: the value of the numeric part, disregarding the sign. The function changes negative numbers to positive numbers, but has no effect on positive numbers.

```
PRINT ABS(123.456)
```

displays '123.456' - no change. But

```
PRINT ABS(-543.345)
```

displays '543.345' - the minus sign has been removed.

INTEGER CONVERSION

The function **INT** removes the fractional part of a number and returns the next lower whole number. Try:

```
PRINT INT(123.4567)
```

Only the whole number part - 123 - is displayed.

Be careful with numbers less than zero. The **INT** function finds the first whole number lower than the number you give it. This means that for negative numbers the answer is not the whole part of the initial number, but one less (or minus one more!). Therefore:

```
PRINT INT(-2.875)
```

displays '-3'.

SIGNS

SGN(N) returns a value which indicates the sign of a number or variable **N**. If the number is positive, the result is 1; if the number is zero, the result is zero; and if the number is less than zero, **SGN** returns -1.

```
PRINT SGN (5)      displays '1'
PRINT SGN (0)      displays '0', and
PRINT SGN (-7)     displays '-1'.
```

TRIGONOMETRY

The trigonometric functions of sine, cosine and arctangent are available in Atari BASIC. They are written:

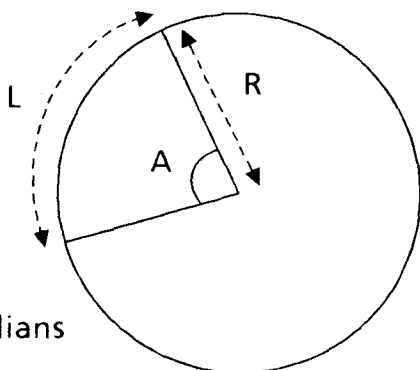
SIN(N)

COS(N)

ATN(N) where **N** is a number or a variable.

The angles may be expressed either in radians or degrees. When first switched on, and after a **RUN** command, the Atari treats all angles as being in radians. To switch to degrees use the command **DEG**. The corresponding command **RAD** returns the Atari to radian mode.

A radian is the ratio of the length of an arc of a circle to the radius. In the diagram overleaf, the angle **A** in radians is L/R . The circumference of a circle is 2π times its radius ($2\pi R$), so it follows that 360 degrees is equivalent to 2π radians.



The angle A is L/R radians

LOGARITHMS

Natural logarithms are provided by the function **LOG**(*number or variable*).

`PRINT LOG(10)` displays '2.30258509'

`PRINT LOG(5000)` displays '8.5171932'

The antilogs of natural logarithms are calculated by **EXP**(N). Try:

`PRINT EXP(LOG(100))`

The answer is 100. The Atari actually prints 99.9999976 – the calculation is not quite accurate.

Logs can be used to calculate roots. This program finds the cube root of 8.

```
10 A = 8
20 R = LOG(A)/3
30 PRINT EXP(R)
```

The answer is 2, because 2 cubed ($2*2*2$) is 8. (Again the Atari is not quite accurate, printing 1.9999999.)

This program calculates fifth roots.

```
10 N = 243
20 R = LOG(N)/5
30 PRINT EXP(R)
```

The program will display '3' (actually 2.999999993), because 3 to the power 5 (3^5 or $3*3*3*3*3$) is 243.

Base ten logarithms are catered for by the function **CLOG**. The antilogs can be calculated by using the formula:

$$\text{Antilog} = 10^{\log}$$

So, if:

$$\text{TENLOG} = \text{CLOG}(A)$$

Then:

$$A = 10^{\text{TENLOG}}$$

RANDOM NUMBERS

The Atari 130XE has a function which provides random numbers. The function is **RND(X)**, which prints a number between 0 and 1. The 'seed' value - X - can be any number or letter, but does directly influence the value of the random number that results. Try **PRINTing RND(1)** a few times. You will get a different number each time.

Dice Throwing

We can use random numbers in programs to imitate the throwing of dice. Try this:

```
10 OPEN#1,4,0,"K:"
20 PRINT "PRESS ANY KEY TO THROW
   THE DICE"
30 GET#1,KEY
40 DICE = INT(RND(0)*6) + 1
50 PRINT
60 PRINT "YOUR NUMBER IS ";DICE
70 FOR DELAY = 0 TO 500:NEXT DELAY
80 GOTO 20
```

This program will throw the die every time you press a key. Pressing the BREAK key will stop the program.

SUMMARY

There are several built-in functions in Atari BASIC which operate on numbers or variables:

SQR(N) calculates the square root of a number N.

ABS(N) returns the absolute value of a number - changing negative numbers into positive ones, and leaving positive numbers unchanged.

INT(N) returns the integer value of N, removing any fractional part.

SGN(N) gives 1 if N is positive, 0 if N is zero and -1 if N is negative.

The trigonometric functions:

SIN(N)

COS(N)

return the values for the angle N (which may be in radians or degrees).

ATN(N)

returns the value of the angle whose Tangent is N.

DEG switches to degrees mode.

RAD switches back to radians.

LOG(N) returns the natural logarithm of a number, N.

CLOG(N) returns the base ten logarithm of N.

RND(N) returns a pseudo random number between 0 and 1.

CHAPTER 9

LOGICAL THINKING

As well as doing arithmetic, the Atari 130XE can perform tests to compare numbers and strings. We have already seen this when using **IF ... THEN**. In this chapter we will examine in more detail the way in which the 130XE makes comparisons.

Consider the program line:

```
220 IF A = 3 THEN 500
```

The line instructs the Atari to branch to line 500 if the variable **A** holds the value 3. What does the Atari do when it encounters this program line?

The first thing the computer must do is decide whether **A** is equal to 3; or to put it another way, whether '**A = 3**' is true. The XE tests this, and if the expression is true, it returns the answer 1; if the expression is false, the answer is 0. If the result is true (1), the commands after **THEN** are obeyed. If the answer is false (0) the program will continue with the next line.

Why are we telling you all this? Because the computer can compare numbers and return true or false values without **IF**. Try this short program:

```
10 A = 3
20 PRINT A = 3
```

The program displays '1'. If you change line 10 to

10 A = 5 (or any other number)
the program will print 0.

This applies to all the other comparisons listed in Chapter 5 as conditions for IF:

- = Equal
- < Less than
- > Greater than
- <> Not equal
- >= Greater than or equal
- <= Less than or equal

Try this:

```
10 A = 5
20 PRINT A <= 7
```

If you try out the other tests, you will find they all behave in the same way.

LOGIC

If two tests are combined, the same true and false answers are still obtained:

```
10 A = 5
20 PRINT ( A < 7 ) AND ( A > 3 )
```

Now, if the computer evaluates simple relational expressions such as $A < 7$, $A > 3$ as 1 or 0, what happens when two are combined, and what does the **AND** do?

There are three BASIC commands which can be used with relational expressions: **AND**, **OR** and

NOT. Forgetting about numerical representations of true and false for the moment, let's look at what these commands do.

Using **AND** to relate two expressions, as in the example above, it seems fairly obvious that the final result will be true only if both smaller expressions are true. This is in fact what happens. Here is a table of the possible combinations, with the two simple expressions represented by X and Y. This type of table is called a 'truth table'.

X	Y	X AND Y
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

Truth Table for AND

The command **OR** also does the obvious thing, returning 1 if X or Y or both are true.

X	Y	X OR Y
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

Truth Table for OR

NOT changes from true to false, and vice versa, so:

```
10  A = 3
20  PRINT NOT A>10
```

prints 1 (true). Here is the truth table for **NOT**:

X	NOT X
TRUE	FALSE
FALSE	TRUE

*Truth Table for **NOT***

SUMMARY

The Atari 130XE can not only handle numerical calculations: it can solve logical problems too. Relational tests such as those performed after **IF** are represented numerically:

True is represented by the number 1.

False is represented by the number 0.

There are three logical operators:

AND A **AND** B is true if both A and B are true.

OR A **OR** B is true if either A is true or B is true or both are true.

NOT **NOT** A is true if A is false.

CHAPTER 10

MEMORY MANAGEMENT

The memory of a computer is built up from a large number of storage units, each of which can hold one number. Each storage unit is called a *memory location*, and each location has a unique reference number called its *address*. In a microcomputer like the Atari 130XE, each memory location can hold an eight-bit binary number, which can have a value between 0 and 255. Larger numbers need more than one location. (If you are not sure about binary numbers, they are explained in Appendix 4.) Eight bits of storage are called a *byte* in computer jargon, so the memory locations of the 130XE hold one byte of data each. The 130XE has a total of 128k of memory, though not all of it may be used easily by BASIC programs.

There are two types of memory used in a computer. Random access memory, or RAM, is used to store data, and its contents can be changed as the data changes. Read only memory, or ROM, cannot be altered and is used to store the programs the computer needs to work at all; the BASIC interpreter for example.

There are two BASIC commands, **PEEK** and **POKE**, which are used to look at individual memory locations. **PEEK** finds out the contents of a location, and **POKE** writes a new number into the memory. You can try out **PEEK** and **POKE** by using them to control the display.

Type in these instructions:

```
GRAPHICS 0:POKE 82,20
```

The cursor will jump to the middle of the screen, and from now on you will only be able to use the right-hand side of the display. This is because the Atari uses location 82 to control the width of the screen - this location normally contains the value 2, resulting in a 2 column left-hand margin. The **POKE** command placed a 20 in this location, causing the left-hand margin to move to column 20. If you now type:

```
PRINT PEEK(82)
```

the Atari will print the answer 20, telling you that the number in location 82 is 20. (It should be, as you've just put it there!). Before you continue, type:

```
POKE 82,2
```

which will reset the margins to their normal positions.

What makes the **POKE** and **PEEK** commands so important is that not all the memory addresses of the 130XE are used for memory. Some of the addresses are used to point to the control registers of the special chips which control the sound and graphics facilities. This means that the chips can be controlled simply by writing a number into these control registers using the **POKE** command. These special chips will be described in the following chapters.

Two other commands for managing memory are **FRE** and **CLR**. The function **FRE(X)** tells you how much memory is left unused: that is the memory which is not holding the BASIC program or its variables.

The **CLR** command clears all the variables stored by a program, but leaves the program itself intact.

You can then start a program with the maximum amount of memory free.

THE EXTRA MEMORY OF THE 130XE

The microprocessor used in the 130XE, the 6502, is able to make use of 65536 memory locations (65536 is 2^{16} , or 64k), and this is the normal maximum for a microcomputer of this type. The 130XE has twice this - 132k of memory - but needs a special method of accessing it. The extra 64k is arranged in four blocks of 16k each, numbered from 0 to 3, and these may be switched in and out of the normal memory area in the region between addresses 16384 and 32767. It is also possible to arrange for the ANTIC display chip to use the extra memory while the 6502 uses the normal memory, and vice versa.

Unfortunately for the BASIC programmer, the extra memory can not be used easily without a knowledge of machine code, as the 16k-32k region is in the middle of the BASIC program area, and altering the memory could upset your programs. The display data used by the ANTIC is usually stored between 32k and 40k, and so, although in theory you could store screens in the extra memory for instant display, in practice it is not all that easy.

The memory is controlled by location 54017. To select a bank of memory use the command:

```
POKE 54017, 193 + 4*BLOCK +16*MODE
```

The mode number has the following meaning:

MODE	6502 sees:	ANTIC sees:
0	Extra memory	Extra memory
1	Normal memory	Extra memory
2	Extra memory	Normal memory
3	Normal memory	Normal memory

CHAPTER 11

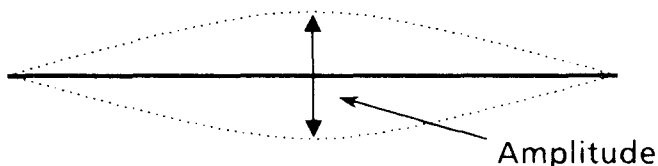
SOUND AND MUSIC

The Atari 130XE has exceptional sound generating facilities. The built-in sound generator (POKEY) is a synthesiser on a chip, with four *voices*, a wide range of octaves, and control of the type of tone produced by each voice.

The Atari has no built-in loudspeaker, but uses the loudspeaker of the television or monitor to which it is connected.

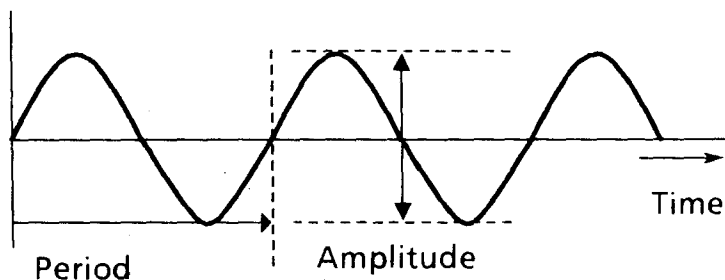
WHAT IS SOUND?

Sound is the effect on our ears of vibrations in the air caused by a vibrating object such as a guitar string or a loudspeaker.



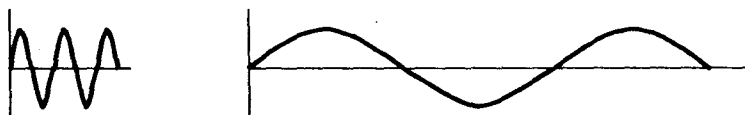
Vibrating Guitar String

If we plot a graph of the way a vibrating object moves over time we get an undulating shape which is called a 'wave'. What you hear depends on the properties of the wave: the rate at which the object vibrates (the frequency of the vibration)

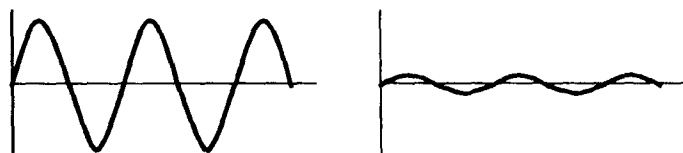


Simple Wave

determines the pitch of the sound, and the amplitude determines the volume.



High and Low Frequencies



Large and Small Amplitudes (Loud and Soft)

Not all sounds have the same shaped waves, and this wave shape, or waveform, has a great effect on the quality of the sound. The shape of the sound waves produced by the 130XE looks like this:



A Square Wave - as produced by the XL

- this is known as a square wave for obvious reasons.

USING THE SOUND GENERATOR

The sound generator chip (known as POKEY) provides four tone channels, each of which can play tones spanning three octaves. The four channels can be set to play independently of each other so that chords and harmonies can be created. The volume of each channel is independently controllable, as is the amount of distortion with which the tones are played.

The chip can be described as 'intelligent', meaning that once data about the sound you wish to hear has been passed to it, the computer can continue running your program while the sound is being generated. For this reason, using the POKEY chip will not slow the running of BASIC programs.

To provide control over the functions of the POKEY chip, there is a BASIC command - **SOUND**

SOUND

The **SOUND** command must be followed by four numbers, separated by commas, which determine:

- 1 The tone channel to be used.
- 2 The period of the note to be played. The number is *not* the frequency of the note, but is inversely proportional to its frequency.
- 3 The distortion to be applied to the sound.

4 The volume of the note on that channel.

A typical **SOUND** command is :

```
SOUND 0,126,10,15
```

where a note of period 126 is played at volume 15 on channel 0. The distortion parameter of 10 specifies no distortion - a pure tone is played.

Try the example, and vary the parameters to get a feel for the noises you can make. You can stop a note from sounding by pressing the RESET key.

The **SOUND** command has the format:

SOUND Channel, Period, Distortion, Volume

and the four parameters work as shown below. The numbers in brackets in the following paragraphs are the allowable range of values for these parameters - if you exceed them an error message is displayed.

CHANNEL (0-3)

This determines which of the four channels is to be used. The remaining parameters in the command pertain to that channel only, and have no effect on the other three channels.

PERIOD (0-65535)

This controls the frequency of the tone, a low number means a high pitched tone; a high number means a low tone. Any number up to 65535 is allowed, but there are only 256 different frequencies which means that the the numbers 0 to 255 represent all possible tones, with these being repeated at higher values so that:

SOUND 0,100,10,14

SOUND 0,356,10,14

SOUND 0,61540,10,14

all produce the same tone.

VOLUME (0-15)

This parameter controls the volume of the tone, with 15 being the loudest possible and 0 being silence. If you use more than one channel at a time, it is advisable to ensure that the total of the volume parameters isn't greater than 32 as exceeding this value may result in distorted sounds which are accompanied by buzzing of the TV speaker. The best way to ensure this is to use a volume parameter of 8 for each channel, which is quite loud enough for most purposes. If not, you can increase the volume using the volume control on the TV set.

DISTORTION(0-65535)

This controls the amount by which the pure tone is distorted. Numbers between 0 and 65535 are allowed but only even numbers have any effect, and there are only eight possible settings – for numbers higher than 16 the effect repeats. There are only seven different degrees of distortion, selected by the numbers 0, 2, 4, 6, 8 and 12; the numbers 10 and 14 both specify a pure tone.

The term *distortion* requires further explanation.

As we have seen, the sound waves generated by the POKEY chip are square waves:



A Square Wave

In order to create distorted tones, this regular pattern is modified by another square wave, this one having a random pattern like this:



A Random Square Wave

When you specify a distortion parameter other than 10 or 14, the square wave generated according to the three other **SOUND** command parameters is modified by the random square wave, such that the only pulses of the original wave you hear are those which coincide with a pulse of the random wave. The diagram opposite shows how this works:

The result of the combination of the two waves would look like the lowest wave in the diagram. As well as distorting the quality of the sound, the pitch of the sound you hear is lower, as some of the cycles of the wave are removed. The different values of the distortion parameter select different random waves, producing different degrees of distortion.

The following program plays a tone while stepping through the different distortion values.

```
10     FOR DISTORT=0 TO 14 STEP 2
20     SOUND 0,100,DISTORT,14
30     FOR DELAY=0 TO 1000:NEXT DELAY
40     NEXT DISTORT
```



SQUARE WAVE



RANDOM WAVE



RESULTANT SOUND WAVE

The Distortion process

50 SOUND 0,0,0,0

Notice how some of the values of the distortion parameter produce more regular sounding noises than others. This is because, despite the name, the pattern of random pulses which comprise the random waves is repetitive. The difference between the various random waves is the time over which the pattern repeats. If this time is short, the noises produced appear repetitive and are useful for engine noises, for example. The most random noises are good for explosions and gun shots.

SOUND CHECKER PROGRAM

The following program will allow you to experiment with the various **SOUND** parameters on each channel.

```
5        REM SOUND CHECKER
10       GOSUB 10000
20       GOSUB FUNKEY
```



```

30      GOSUB KEYBOARD
40      GOTO 20
993     REM
994     REM *****
995     REM *                                     *
996     REM * SUBROUTINE FUNKEY *
997     REM *                                     *
998     REM *****
999     REM
1000    K=PEEK(53279)
1010    IF K=3 THEN GOSUB OPTION
1020    IF K=5 THEN GOSUB SELECT
1030    IF K=6 THEN GOSUB START
1040    RETURN
1993    REM
1994    REM *****
1995    REM *                                     *
1996    REM * SUBROUTINE KEYBOARD *
1997    REM *                                     *
1998    REM *****
1999    REM
2000    K=PEEK(764)
2010    IF K=54 THEN POKE 764,255:
        GOSUB DECREASE:GOSUB PLAY:GOTO
        2100
2020    IF K=55 THEN POKE 764,255:
        GOSUB INCREASE:GOSUB PLAY:GOTO
        2100
2030    IF K=28 THEN GOSUB START:PRINT
        "↵":END
2100    RETURN
2993    REM
2994    REM *****
2995    REM *                                     *
2996    REM * SUBROUTINE OPTION *
2997    REM *                                     *
2998    REM *****
2999    REM
3000    POSITION 2,4:PRINT"CHANNEL
        FREQUENCY DISTORTION VOLUME"
3005    P=P+2:IF P=6 THEN P=0

```

```

3010  X=((P=0)*11)+((P=2)*22)+
      ((P=4)*34)
3020  IF P=0 THEN M$="FREQUENCY":REM
      THIS TEXT IN REVERSE
3030  IF P=2 THEN M$="DISTORTION":
      REM THIS TEXT IN REVERSE
3040  IF P=4 THEN M$="VOLUME":REM
      THIS TEXT IN REVERSE
3050  POSITION X,4:PRINT M$
3060  GOSUB KEYBOARD
3070  RETURN
3993  REM
3994  REM *****
3995  REM *
3996  REM * SUBROUTINE SELECT *
3997  REM *
3998  REM *****
3999  REM
4000  POSITION 3,7+2*CHAN
4010  PRINT " "
4020  CHAN = CHAN+1
4030  IF CHAN=4 THEN CHAN=0
4040  POSITION 3,7+2*CHAN
4050  PRINT ">"
4060  RETURN
4993  REM
4994  REM *****
4995  REM *
4996  REM * SUBROUTINE START *
4997  REM *
4998  REM *****
4999  REM
5000  FOR Z=0 TO 3:SOUND Z,0,0,0:
      NEXT Z
5010  GOSUB 12010
5020  RETURN
5993  REM
5994  REM *****
5995  REM *
5996  REM * SUBROUTINE DECREASE *
5997  REM *

```

```

5998 REM *****
5999 REM
6000 S(CHAN,P)=S(CHAN,P)-1-(P=2)
6010 IF S(CHAN,P) <0 THEN S(CHAN,P)
    =S(CHAN,P+1)
6020 RETURN
6493 REM
6494 REM *****
6495 REM *
6496 REM * SUBROUTINE INCREASE *
6497 REM *
6498 REM *****
6499 REM
6500 S(CHAN,P)=S(CHAN,P)+1+(P=2)
6510 IF S(CHAN,P)>S(CHAN,P+1) THEN
    S(CHAN,P)=0
6520 RETURN
7993 REM
7994 REM *****
7995 REM *
7996 REM * SUBROUTINE PLAY *
7997 REM *
7998 REM *****
7999 REM
8000 SOUND CHAN,S(CHAN,0),S(CHAN,2)
    ,S(CHAN,4)
8005 XP=X+((P=0)*3)+((P=2)*4)+
    ((P=4)*2)
8010 POSITION XP,7+2*CHAN
8020 PRINT "    ":REM 3 SPACES
8030 POSITION XP,7+2*CHAN
8040 PRINT S(CHAN,P)
8050 RETURN
9993 REM
9994 REM *****
9995 REM *
9996 REM * SET UP DISPLAY *
9997 REM *
9998 REM *****
9999 REM
10000 PRINT "↵"

```

```
10010 POSITION 13,1
10020 PRINT "XE SOUND CHECK":REM IN
      REVERSE
10030 POSITION 2,4
10040 PRINT "CHANNEL  FREQUENCY
      DISTORTION  VOLUME"
10050 POSITION 12,5:PRINT "(0-255)
      (0-14)      (0-15)"
10060 FOR CHAN=0 TO 3
10070 POSITION 5,7+2*CHAN
10080 PRINT CHAN:NEXT CHAN
10090 CHAN=0:POSITION 3,7+2*CHAN
10095 PRINT ">"
10993 REM
10994 REM *****
10995 REM *                                     *
10996 REM * SUBROUTINE ADDRESSES *
10997 REM *                                     *
10998 REM *****
10999 REM
11000 FUNKEY = 1000
11010 KEYBOARD = 2000
11020 OPTION = 3000
11030 SELECT = 4000
11040 START = 5000
11050 DECREASE = 6000
11060 INCREASE = 6500
11070 PLAY = 8000
11080 POKE 752,1:REM TURN OFF CURSOR
11993 REM
11994 REM *****
11995 REM *                                     *
11996 REM * INITIALISE *
11997 REM *                                     *
11998 REM *****
11999 REM
12000 DIM S(3,5),M$(20)
12010 FOR CHAN=0 TO 3
12020 FOR P=0 TO 4 STEP 2
12030 S(CHAN,P)=0
```

```
12035 XP=((P=0)*14)+((P=2)*26)+  
      ((P=4)*36):GOSUB 8010  
12040 READ D:S(CHAN,P+1)=D  
12050 NEXT P:RESTORE  
12060 NEXT CHAN:P=-2:CHAN=0  
12070 DATA 255,14,15:REM MAX DATA  
12080 GOSUB OPTION  
20000 RETURN
```

The program creates a display of the current Frequency, Distortion and Volume parameters for each of the four tone channels, and allows you to change them to experiment with the **SOUND** command.

To select one of the four channels, press the **SELECT** key on the function keypad - a > symbol indicates which channel is selected.

To alter one of the parameters on that channel, use the **OPTION** key to make your choice - the heading of the currently selected column will be highlighted. Use the < and > keys to increase or decrease the value displayed. The sound you hear is changed with the changing parameters, and in this way you can tailor the four channels to create the noise you require and make a note of the parameters. If you make a mess of things, pressing the **START** key sets all the parameters back to zero.

HOW THE PROGRAM WORKS

The program consists of a number of modules, defined in lines 11000 to 11070.

The display is created by lines 10000 to 10080 and lines 12000 to 12080 create an array which contains the parameters for each channel (initially zero) and the maximum value for each parameter.

The program selects Channel 0 and Frequency options as a starting point by setting the variables P and CHAN in line 12060, then calling the routine normally called when the OPTION key is pressed.

When running, the program constantly cycles through the two routines FUNKEY and KEYBOARD which read the two keyboards, until a keypress is detected. The appropriate subroutine is called and SOUND parameters for the selected channel changed by either the DECREASE or INCREASE subroutines.

After each keypress is processed, the PLAY routine at 8000 is called to update the noise, and the program awaits the next keypress.

PLAYING TUNES

To play tunes, we must 'feed' a SOUND command with a series of frequency parameters in turn, and one way of doing this is shown in this example:

```
10      SOUND 0,INT(100*RND(1)),10,8:
        GOTO 10
```

As mentioned earlier, all four channels are independant - allowing us to play four different tones at once on each channel.

```
5        REM CACOPHONY!
10       SOUND 0,INT(10*RND(1)),10,8
20       SOUND 1,INT(50*RND(1)),10,8
30       SOUND 2,INT(100*RND(1)),10,8
40       SOUND 3,INT(250*RND(1)),10,8
50       GOTO 10
```

Neither of these examples could be described as music, for that we need a more controlled way of calculating frequencies. The table in Appendix 10 lists the frequency values corresponding to musical

notes (if you are unfamiliar with musical notation, refer to Appendix 12). Using this table we can build up a series of numbers corresponding to notes but we need a way of storing them. One way is to store them in a string, reading them one at a time as parameters for a **SOUND** command. This technique is illustrated in the following program.

```
10    DIM TUNE$(11)
20    TUNE$ = "QQQQDHQQUQ"
30    FOR Z=1 TO LEN(TUNE$)
40      N = ASC(TUNE$(Z,Z))
50      SOUND 0,N,10,8
60      FOR D = 0 TO 100:NEXT D
70      SOUND 0,0,0,0
80    NEXT Z
```

The main limitation with this method is that the notes are all the same length, which renders it impractical for all but the simplest music.

There are several ways of overcoming this: for example, a second string could be used to hold the data for the length of notes. Add these lines to the last program to hear this technique in action.

```
15    DIM L$(11)
25    L$="32132121214"
60    FOR D=0 TO 100*VAL(L$(Z,Z)):
      NEXT D
```

This method is adequate for simple tunes but a better way is to hold information concerning notes and note lengths in **DATA** statements like this:

```
10    FOR Z=1 TO 13
20      READ NO,LE
30      SOUND 0,NO,10,8
40      FOR DUR=0 TO 100*LE:NEXT DUR
50      SOUND 0,0,0,0
```

```
60      NEXT Z
70      SOUND 0,0,0,0:END
100     DATA 81,2,81,2,88,1,81,1,72,
           2,96,4,108,4,122,2,122,2,128,1
           ,122,1,108,2,144,8
```

The next program uses **DATA** statements to play a tune in four part harmony. There are five **DATA** items for each group of notes - the periods of the four voices, and a value for the note length. The program will play a tune you should recognise if you have run the Audio-Visual self checking routine on the Atari.

It will take you quite a time to type in all the **DATA**, but the end result is well worth all the effort.

You will notice that the values used for note periods are not the same as those given in the Atari 130XE Owner's Manual supplied with your computer. We found that the values given by Atari were not well tuned, and that the note values given in Appendix 11 sounded much better! Even so, the tuning is not perfect, as the 255 available periods are not sufficient to give a perfect sound. Later in the chapter we will describe how to increase the range and tuneability of the Atari at the expense of reducing the number of voices from four to two.

```
10      D=10
20      V0=0:V1=1:V2=2:V3=3
30      VOL=4:VOL0=10:VOL3=6
100     READ P0,P1,P2,P3,T
110     IF T=255 THEN END
120     SOUND V0,P0,D,VOL0
130     SOUND V1,P1,D,VOL
140     SOUND V2,P2,D,VOL
150     SOUND V3,P3,D,VOL3
```



```
155   FOR DEL=0 TO T*110:NEXT DEL
160   GOTO 10
1000  DATA 85,0,0,0,2,95,0,0,0,2,
      71,0,0,0,2,63,0,0,0,1,47,0,0,0
      ,1,56,0,0,0,2
1010  DATA 63,0,0,0,1,47,0,0,0,1,
      56,0,0,0,2,71,0,0,0,2,63,0,0,0
      ,2,85,0,0,0,2,95,0,0,0,2
1020  DATA 85,113,143,171,2,
      95,127,151,191,2,71,113,143,17
      1,2,63,75,127,191,1,47,75,127,
      191,1,56,75,95,227,2
1025  DATA 63,75,127,191,1,47,75,
      127,191,1,56,71,95,143,2,71,85
      ,113,171,2,63,85,101,255,2,85,
      127,171,203,2
1026  DATA 95,127,151,191,2
1030  DATA 95,0,0,0,2,85,0,0,0,2,
      113,0,0,0,2,95,0,0,0,1,85,0,0,
      0,1,127,0,0,0,2
1040  DATA 85,0,0,0,1,75,0,0,0,1,95,
      0,0,0,2,47,95,191,0,2,56,95,14
      3,0,2,63,95,171,0,1,71,95,171,
      0,1,95,191,0,0,2
1050  DATA 95,0,0,0,2,85,0,0,0,2,
      113,0,0,0,2,95,0,0,0,1,85,0,0,
      0,1,105,0,0,0,2
1060  DATA 71,0,0,0,1,63,0,0,0,1,79,
      0,0,0,2,39,79,159,0,2,47,79,11
      9,0,2,52,79,143,0,1,59,79,143,
      0,1,79,0,159,0,2
1062  DATA 0,0,0,0,0
1070  DATA 79,105,159,0,2,71,105,
      143,0,2,79,105,159,0,2,71,105,
      159,0,1,63,105,127,0,1,52,105,
      0,0,1
1072  DATA 71,105,143,0,1,79,105,
      159,179,2
1080  DATA 59,79,95,191,1,52,63,79,
      211,1,47,59,79,239,1,39,0,0,23
```

9,1,44,52,71,211,1,47,59,79,19
 1,1
 1082 DATA 52,63,79,159,1,44,0,85,
 159,1,47,59,71,143,1,59,0,113,
 143,1,52,63,79,159,2
 1090 DATA 79,105,159,0,2,71,105,
 143,0,2,79,105,159,0,2,71,105,
 143,0,1,63,105,127,0,1,52,105,
 0,179,1,71,0,143,0,1
 1100 DATA 63,85,127,143,2,56,85,
 113,143,2,63,85,127,151,2,56,8
 5,113,143,1,47,85,95,143,1,42,
 85,0,143,1
 1102 DATA 56,85,113,143,1,63,85,
 127,143,2
 1110 DATA 47,63,95,151,1,42,50,63,
 171,1,37,47,63,191,1,31,0,63,1
 91,1,35,42,56,171,1,37,47,63,1
 51,1,42,50,63,127,1
 1112 DATA 35,0,56,127,1,37,47,75,
 113,1,47,0,95,113,1,42,50,85,1
 27,2
 1120 DATA 37,63,75,151,1,50,63,74,
 151,1,47,56,95,143,2,37,63,75,
 191,2,56,71,101,171,2,37,63,75
 ,191,2,56,71,101,171,2
 1130 DATA 47,75,95,227,1,63,75,95,
 227,1,56,71,95,171,2,47,75,95,
 227,2,56,71,95,171,2
 1132 DATA 47,75,95,227,1,63,75,95,
 227,1,56,71,95,171,2
 1140 DATA 63,85,105,255,1,63,85,
 101,255,1,75,95,127,191,2,71,9
 5,113,171,2,63,85,105,255,1
 1142 DATA 63,85,101,255,1,75,95,
 127,191,2,71,95,113,171,1,56,9
 5,113,171,1
 1150 DATA 63,85,101,255,2,75,95,
 113,191,2,63,85,101,255,2,47,9
 5,0,191,2,52,71,75,171,1,56,0,
 113,171,1

```
1152 DATA 63,75,127,151,1,71,95,  
113,143,1  
1160 DATA 63,95,127,151,2,56,95,  
113,143,2,47,63,75,151,2,42,50  
,71,171,1,35,0,71,171,1,47,0,9  
5,191,2,42,0,85,171,2  
1170 DATA 47,0,95,191,2,52,71,85,  
171,1,56,0,113,171,1,63,75,95,  
151,1,71,95,113,143,1,63,95,12  
7,151,2  
1172 DATA 56,95,113,143,2,47,63,  
95,151,2  
1180 DATA 42,50,71,171,1,35,0,71,  
171,1,47,95,0,191,2,42,85,0,17  
1,2,47,95,0,191,2,0,0,0,0,0  
1182 DATA 85,85,171,0,2,95,95,191,  
0,2  
1190 DATA 42,50,71,255,1,35,71,0,  
255,1,47,95,0,191,2,42,85,0,17  
1,2,47,95,0,191,2,0,0,0,0,0  
1192 DATA 85,105,143,211,2,95,127,  
151,191,2  
1200 DATA 71,95,143,227,2,63,75,  
127,255,1,47,0,95,255,1,56,71,  
95,143,2,63,75,95,151,1,47,0,9  
5,151,1  
1220 DATA 56,71,95,171,2,71,95,113,  
171,2  
1230 DATA 63,85,101,255,2,85,101,  
127,203,2,95,127,151,191,2,0,0  
,0,0,0,85,113,143,171,2  
1232 DATA 95,127,151,151,2,71,95,  
143,171,2  
1240 DATA 63,75,95,191,1,47,0,95,  
191,1,56,71,95,143,2,71,85,113  
,171,2,53,63,85,255,2,63,75,95  
,191,2,71,95,113,143,3  
10000 DATA 0,0,0,0,255
```

ADVANCED TECHNIQUES

The effects you can achieve using the **SOUND** command can be quite spectacular, but it is possible to do more, since this single BASIC command doesn't offer control over all of the functions of the POKEY chip. With more understanding of how the chip operates you can have a greater influence on the noises it creates

SOUND REGISTERS

Like other chips in the 130XE, the POKEY chip is controlled by a number of registers, whose addresses and functions are shown here.

The **SOUND** command has a direct influence on the first six of these registers, and loads those pertinent to the specified channel with the appropriate data, for example:

```
SOUND 0,100,10,15
```

can be duplicated by the commands

```
POKE 53760,100
```

```
POKE 53761,175
```

The first **POKE** command sets the frequency register for channel 0 to 100, which is fairly obvious, but the contents of the channel 0 control register (53761) require some explanation.

REGISTER	FUNCTION
53760	Channel 0 Frequency
53761	Channel 0 Control
53762	Channel 1 Frequency
53763	Channel 1 Control
53764	Channel 2 Frequency
53765	Channel 2 Control
53766	Channel 3 Frequency
53767	Channel 3 Control
53768	POKEY Control

POKEY Chip Registers

The eight bits of the control register control different aspects of the sound.

Bits 0, 1, 2 and 3 control the volume – four bits can represent the numbers 0 to 15, hence the restriction of 0 to 15 for the volume parameter in **SOUND** commands.

Bits 5, 6 and 7 control distortion. Three bits means eight different combinations or eight different degrees of distortion.

We **POKEd** a value of 175 into this register, and obtained a pure tone at volume 15. The volume is obtained by setting each of the four volume bits to 1. A pure tone involves setting bits 5 and 7 to 1, the value of bit 6 being irrelevant (that's why the

numbers 10 and 14 give the same pure tone in a **SOUND** command).

Converting these bits to a decimal number gives:

$$128 + 32 + 8 + 4 + 2 + 1 = 175$$

You can achieve the same effects as those obtainable with the **SOUND** command in this way, but this knowledge isn't much use. But what about bit 4 of the control register? This is the volume only bit and when set to 1, the value of the volume bits (0 to 3) is used to directly control the speaker of the TV.

Bits 0 to 3 select a voltage which is applied to the TV speaker if bit 4 is set. The higher the value of bits 0 to 3, the higher the voltage and the greater the deflection of the TV speaker.

For example type:

```
POKE 53763,31
```

When you press the **RETURN** key, you will hear the beep as you press the key, followed by a click from the speaker.

```
POKE 53763,16
```

will select a zero voltage and the speaker will return to its rest position, with another click. This is how the 130XE creates noises with square waves, each pulse of which moves the speaker out then back to rest again.

Type in the following program, which steadily increases the voltage applied to the speaker.

```
10      FOR VOLTS=1 TO 15
```

```
20    FOR Z=1 TO 100
30    POKE 53763,16+VOLTS
40    POKE 53763,16:REM 0 VOLTS
50    NEXT Z
60    NEXT VOLTS
```

Each increase in voltage is alternately applied then switched off 100 times, causing a buzzing sound, which increases in volume with each increase in voltage. For each voltage the distance travelled by the speaker cone increases, displacing an increased amount of air, and so producing a louder sound.

We can change the frequency of the noise by inserting a delay loop:

```
35    FOR D=0 TO 30:NEXT D
```

With this delay, you can hear the individual clicks as the speaker moves in and out.

Unfortunately the speed limitations imposed by BASIC mean that this technique cannot be exploited fully without resorting to machine code.

THE POKEY CONTROL REGISTER

This register provides additional control of the POKEY chip, and opens up many more sound generating possibilities.

The frequency of the square waves generated by the POKEY chip is governed by another square wave generated by the computer. The frequency of this stream of pulses (known as *clock pulses*) has a bearing on the range of frequencies you can generate. You can control the frequency of these clock pulses and thereby greatly increase the range

of frequencies generated by the POKEY chip. Try this example:

```
SOUND 0,255,10,8
```

will generate the lowest frequency tone possible with a **SOUND** command. If you now type:

```
POKE 53768,1
```

the frequency of the note produced will drop dramatically. This is because bit 0 of the POKEY control register (PCR from now on) controls the frequency of the clock pulses applied to the POKEY chip. Bit 0 is usually set to 0, but when set to 1, the frequency of the clock pulses is reduced by a factor of about four - resulting in a much lower output frequency.

This effect applies to all four tone channels.

The other seven bits in PCR are used to control other functions, as shown in this table.

Bits 5 and 6 provide a more flexible way of changing the clock frequency applied to Channels 2 and 0, without affecting other channels. The clock frequency is increased by a factor of nearly 28 - try this example.

```
SOUND 2,200,10,15
```

```
POKE 53768,32
```

Note that the frequency of the output tone is much higher after setting bit 5 of PCR.

```
10    FOR Z=1 TO 12
20    READ N
30    SOUND 0,N,10,8
```


BIT	FUNCTION
0	Switch clock from 64kHz to 15 kHz
1	High Pass Filter on Channel 1 (controlled by Channel 3)
2	High Pass Filter on Channel 0 (controlled by Channel 2)
3	Adds Channel 2 to Channel 3
4	Adds Channel 0 to Channel 1
5	Channel 2 clock = 1.79 MHz
6	Channel 0 clock = 1.79 Mhz
7	Increases repeat rate of random waves

POKEY Control Register Functions

```

40    SOUND 2,N,10,8
50    FOR P=0 TO 200:NEXT P
60    NEXT Z
70    SOUND 0,0,0,0:END
100   DATA 251,230,217,204,193,182,
      173,162,153,144,136,128

```

This program plays ascending notes on two channels at once. If you add the line:

```

45    POKE 53768,32

```

bit 5 of the PCR is set, increasing the clock frequency applied to Channel 1 – try it and see!

This method of increasing the frequency of the output tone is rather crude, and the PCR provides facilities for more accurate control of frequencies.

In normal use we are restricted to 256 different frequencies per channel, since the frequency register for each channel consists of only eight bits. This restriction means that many frequencies cannot be achieved. However, PCR bits 3 and 4 allow you to 'join' two channels together, forming one channel with a 16 bit frequency register.

Bit 3 joins channel 2 to channel 3, and bit 4 joins channel 0 to channel 1. So if you are prepared to reduce the number of independent tone channels, you will be able to generate a much wider range of frequencies.

The following program demonstrates the increased frequency range by joining channels 0 and 1.

```
10      SOUND 0,0,0,0
20      POKE 53768,64+16
30      POKE 53761,160
40      POKE 53763,175
50      FOR Z=0 TO 65535
60      HI = INT(Z/256)
70      LO = Z-(HI*256)
80      POKE 53760,LO
90      POKE 53762,HI
100     NEXT Z
110     SOUND 0,0,0,0
```

The program will take a long time to run, so you might like to try adding a **STEP** to the **FOR ... NEXT** loop to speed things up a bit.

The range of frequencies can be altered by setting other bits in PCR which control the clock frequency

applied to the POKEY chip. Try changing line 20 to:

```
20      POKE 53768,16
```

This will select the normal clock frequency instead of 1.79 MHz (1 MHz is one million cycles per second) making the lowest frequency about 1 Hz (or one click per second). The lowest frequency can be reduced still further, by setting bit 1 of PCR to reduce the clock frequency to 15 kHz (1 kHz is one thousand cycles per second):

```
20      POKE 53768,17
```

FILTERING

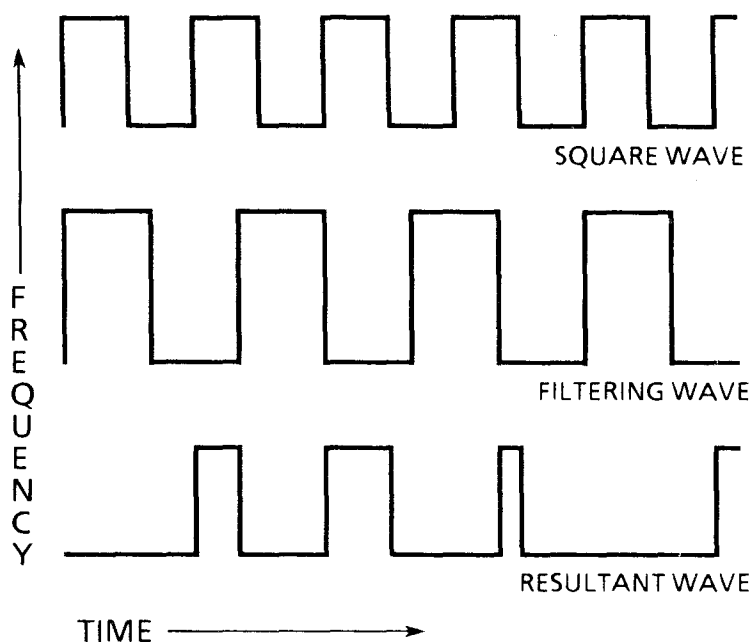
Bits 1 and 2 of PCR control what are known as *high pass filters*. These are filters which cut out frequencies lower than a certain value. In the POKEY chip, the filtering frequency (that below which frequencies are cut off) is set on one of the tone channels and acts on another channel.

As shown in the table of POKEY control registers at the beginning of this section, if bit 1 of PCR is set, any frequency on channel 1 which is lower than that on channel 3 is filtered out. If bit 2 is set, channel 2 acts as a filter on channel 0.

The principle is as follows:

At any instant, if the frequency on the tone channel is greater than that of the filter, the tone will be heard, otherwise it will not. This gives rise to an effect similar to, but much more regular than, distortion as illustrated in the diagram opposite. Try this example to hear it in action.

```
10      SOUND 0,0,0,0
```



```

20      POKE 53768,4
30      POKE 53760,240
40      POKE 53761,168
50      POKE 53764,120
60      POKE 53765,160
70      GOTO 70

```

If the volume bits of the filter channel are set, that tone channel will be heard as well – change line 60 to:

```

60      POKE 53765,168

```

If you delete line 20, the two tone channels alone will be heard.

Finally, bit 7 of the PCR controls the rate at which the pattern of the square wave controlling the

distortion repeats, and when set to 1, makes any distorted sound much more regular, as the following example shows. You will hear the sound change after about two seconds, as line 50 sets bit 7 in the PCR.

```
10      SOUND 0,10,8,8
20      SOUND 1,100,8,8
30      SOUND 2,255,8,8
40      FOR D=0 TO 1000:NEXT D
50      POKE 53768,128
60      GOTO 60
```

As you can imagine, the number of possible combinations of the techniques mentioned in this chapter is virtually infinite and a good deal of experimentation is required to get the best out of the POKEY chip.

CHAPTER 12

CHARACTERS

Up to now all the displays we have created have comprised light blue characters on a dark blue background. You will probably be aware that the 130XE is capable of more impressive displays than this, and the following four chapters will explain how to make the most of the graphics potential of your computer.

CHANGING COLOURS

The 130XE is able to display 16 primary colours, and you can choose any one of these to be the screen or border colour using a BASIC command - **SETCOLOR**

The syntax of the **SETCOLOR** command is :

SETCOLOR R,C,L

The three parameters specify the *register* or store location into which we want to place colour information, the code specifying the *colour* we require, and the *luminance*, or brightness, of the colour.

COLOUR REGISTERS

The XE reserves five memory locations for storing colour information and these *colour registers* are numbered 0 to 4. The colour of the screen is controlled by the contents of register 2, so to change

the screen colour we use the **SETCOLOR** command to alter the contents of register 2.

The sixteen colours from which we make our choice are represented by code numbers 0 to 15, as shown in this table:

CODE	COLOUR
0	GREY
1	GOLD
2	ORANGE
3	RED
4	PINK
5	VIOLET
6	PURPLE
7	LIGHT BLUE
8	DARK BLUE
9	BLUE GREEN
10	BLUE
11	DARK BLUE
12	GREEN
13	DARK GREEN
14	OLIVE GREEN
15	ORANGE

XL Colour Codes

The actual colour displayed depends on the third parameter in the **SETCOLOR** command - luminance. This controls the brightness of the colour and can have a value between 0 and 14. The

higher the luminance number the brighter the colour; so grey at luminance 0 will be black, but at luminance 14 will appear white. Odd values for luminance have the same effect as the lower even number, so there is a choice of eight.

To see **SETCOLOR** in action, type in the following program which cycles through the possible screen and border colour combinations.

```
10    FOR COLR=0 TO 15
20    FOR LUM=0 TO 14 STEP 2
30    SETCOLOR 2,COLR,LUM
40    FOR DELAY=0 TO 200:NEXT DELAY
50    NEXT LUM
60    NEXT COLR
```

The program changes the contents of Register 2 which controls screen colour, if you change line 30 to:

```
30    SETCOLOR 4,COLR,LUM
```

the border colour will be affected. It is also possible to alter the luminance of the characters, by altering register 1 as in this example, where characters are made to flash by controlling their luminance.

```
10    SETCOLOR 2,3,0
20    LUM = 14
30    SETCOLOR 1,3,LUM
40    FOR PAUSE=0 TO 100:NEXT PAUSE
50    LUM = 14-LUM
60    GOTO 30
```

38 OR 40 COLUMN SCREEN ?

All our programs so far have used a screen display of 38 columns by 24 rows, leaving a margin of two characters at the left of the display. This margin is

preset to cater for the variety of TVs which might be connected to the computer. It is possible to regain the two columns and allow a full 40 column display by altering the contents of one of the two memory locations that the 130XE uses to control the width of the display. Location 82 sets the position of the left hand margin, and location 83 the right hand margin. When the 130XE is switched on, a value of 2 is placed in location 82, setting the lefthand edge of the display at column 2, so to regain the two 'lost' columns, simply:

```
POKE 82,0
```

You will then have access to all 40 columns until you switch off the computer or press RESET.

DIFFERENT DISPLAYS

As we have said, the Atari is capable of a variety of display types, and the different displays are referred to by a number - the MODE number. The 40 by 25 display is known as mode 0, and is the default mode which is selected when you switch on the computer. There is a BASIC command which is used to change modes - **GRAPHICS**.

The syntax of the **GRAPHICS** command is:

```
GRAPHICS M
```

where M is the number of the mode required. So typing:

```
GRAPHICS 0
```

would select the standard display - but there isn't much point in doing that since that's the mode we are using! There are two other modes which can be

used to display characters similar those in mode 0; modes 1 and 2.

GRAPHICS MODES 1 AND 2

The characters in mode 1 are twice as wide as those in mode 0, so the number displayable on one line is reduced to 20 (the preset margins do not apply to Graphics 1 and 2 displays). Mode 2 characters are twice as high and twice as wide, bringing the number of characters per screen or *resolution* down to 20 by 12.

As well as the different sized characters the screen is split into two parts - a small section at the bottom is reserved for a four line mode 0 display while the top portion is reserved for mode 1 or 2 characters. The mode 0 window means that the number of rows of characters that can be displayed in modes 1 and 2 is reduced. To see these modes type in the following program:

```
5      REM GRAPHICS 1 AND 2 DEMO
10     MODE=1
20     GRAPHICS MODE
30     PRINT:PRINT"GRAPHICS 0 WINDOW"
40     POSITION 1,5
50     PRINT#6;"GRAPHICS ";MODE;"
        SCREEN"
60     GOTO 60
```

As you can see from the listing, **PRINT** commands cause characters to appear in the mode 0 window, but another technique is required to put characters on the mode 1 section of the screen.

To cause characters to appear on the mode 1 and 2 screens, the **PRINT** command is modified to **PRINT #6**. This instructs the computer to send the data following the **PRINT** command to the destination specified after the **#** symbol. In this

case the destination is 6, which represents the screen. This technique is commonly used to send data to peripheral devices such as the cassette unit (see Chapter 16) and here we are treating the model screen as if it were a peripheral device.

The colour and luminance of the message displayed by the last program is controlled by colour register 0, and can be changed with the **SETCOLOR** command. However it is possible to display characters in up to four different colours on a mode 1 or 2 screen, as the following program demonstrates.

```
10    GRAPHICS 1
20    PRINT#6;"REGISTER ZERO"
30    PRINT#6
40    PRINT#6;"register one"
50    PRINT#6
60    PRINT#6;"REGISTER TWO":REM
      THIS TEXT IN REVERSE
70    PRINT#6
80    PRINT#6;"register three":REM
      THIS TEXT IN REVERSE
```

Notice that regardless of the case specified in the program, the messages on the screen are all in upper case and those which would appear in inverse video in mode 0 do not do so in mode 1. This is because the additional information which would affect case and inverse video in mode 0, is used to select one of four colour registers to determine the colour of the characters. The following table shows which registers are selected by which type of text.

Characters can be positioned on the mode 1 and 2 screens using the **POSITION** command as in mode 0, but remember that because the characters are larger than those in mode 0, the limits of screen coordinates are correspondingly reduced.

TEXT	REGISTER
UPPER CASE	0
lower case	1
INVERSE UPPER CASE	2
inverse lower case	3

Colour Register use in Modes 1 and 2

There is another way to put characters on the mode 1 and 2 screens using the commands **COLOR** and **PLOT**.

With this method, characters are placed on the mode 1 or 2 screens with a **PLOT** command, the syntax of which is:

PLOT X, Y

where X and Y are screen coordinates, and X=0, Y=0 specifies the top left-hand corner of the display.

The character to be plotted is selected using the **COLOR** command. The syntax of the **COLOR** command depends upon the graphics mode in use, and in modes 1 and 2, the syntax is:

COLOR C

where C is a code which specifies not only a character but also the register in which colour information for that character is stored. For each character there are four possible code numbers, each specifying the same character, but a different colour register. For example, the code 42 represents the asterisk, and specifies register 0 for colour information. The number 10 also represents an asterisk, but its colour is determined by the

contents of register 1. A full table of all these codes is given in Appendix 7.

The following program demonstrates how multi-coloured messages can be displayed on the mode 1 and 2 screens using the **COLOR** and **PLOT** commands.

```
5      REM MODE 1 & 2 COLOR/PLOT DEMO
10     MODE=2
20     GRAPHICS MODE
30     FOR CHAR=0 TO 18
40     READ COLR
50     COLOR COLR:PLOT CHAR,8/MODE
60     NEXT CHAR
70     GOTO 70
500    DATA 52,32,100,201,230,70,101
510    DATA 210,229,78,116,32,195,239
520    DATA 76,111,213,242,83
```

The **DATA** statements in lines 100 to 120 contain codes for the characters and their associated colour registers taken from the table in Appendix 7, and these are selected one at a time and plotted by the loop between line 30 and 60. The program will work equally well in mode 1.

The colours in which the characters are displayed are the default values for each of the colour registers, as shown in the table opposite.

The program can be made more interesting by changing these registers to give different colours and luminances. If you do change the value in a register with the **SETCOLOR** command, all characters whose colour is determined by that register will instantly take on the new colour. You can see this by deleting line 70 and adding the following lines to the last program:

```
100    FOR COLR=0 TO 15
```

LOC.	REG.	COLR.	LUM.	COLOUR
708	0	2	8	ORANGE
709	1	12	10	GREEN
710	2	9	4	BLUE
711	3	4	6	PURPLE
712	4	0	0	BLACK

Default Colour Register Values

```

110   FOR REG=0 TO 3
120   SETCOLOR REG,COLR+2*REG,12
130   NEXT REG
140   FOR DELAY=0 TO 50:NEXT DELAY
150   NEXT COLR
160   GOTO 100

```

The colours will continue to cycle until you press BREAK.

You can also use the **POKE** command to directly alter the contents of the colour registers. Add these lines to the last program and type GOTO 200 to cycle through all the colour and luminance combinations for the screen colour, by **POKE**ing the colour register location (given in the table) with every number from 0 to 255.

```

200   REG=710
210   FOR COL=0 TO 255
220   POKE REG,COL
230   FOR PAUSE=0 TO 100:NEXT PAUSE
240   NEXT COL
250   GOTO 250

```

When you change character colours, you will notice that the colour of the mode 0 window also changes, because this is controlled by one of the registers controlling character colours (register 2). This is a

bit distracting, and there is a way of removing the mode 0 window, allowing the entire screen to be given over to mode 1 or 2 displays. To achieve this simply add 16 to the mode number, so that mode 1 without a mode 0 window is obtained by typing:

GRAPHICS 1+16

Try changing line 20 in the previous program to see this. The penalty is that if the program is halted for any reason, the Atari will return to mode 0 - losing your display.

One drawback of being able to specify one of four colours for a mode 1 or 2 display is that only 64 characters can be represented by codes 0 to 255. This restricts us to using only upper case letters in the previous examples, which is only half of the available character set. To display the other half of the character set, we must change the place from which the Atari obtains its characters by **POKE**ing the character set pointer (location 756) with the number 226 (its default value is 224). Don't worry too much about what this means - all will become clear later in this chapter.

The following program, in a romantic vein, demonstrates the use of the other half of the character set in mode 2.

```
10    MODE=2+16
20    GRAPHICS MODE
25    SETCOLOR 0,3,8
30    POKE 756,226
35    FOR CHAR=2 TO 17
40    READ DATA
50    COLOR DATA:PLOT CHAR,4
60    NEXT CHAR
65    REM LOOP TO FLASH MESSAGE
70    FOR LUM=0 TO 14 STEP 2
80    SETCOLOR 1,12,LUM
```

```
85     FOR DELAY=0 TO 30:NEXT DELAY
90     NEXT LUM
100    GOTO 70
110    DATA 104,97,112,112,121,32,
        32,118
120    DATA 97,108,101,110,116,105,
        110,101
```

The screen full of hearts occurs because the character corresponding to space in the first half of the character set is a heart, so everywhere which would normally be blank contains a heart. The colour of the hearts is set in line 25, and the second half of the character set specified in line 30.

If you change line 30 to:

```
30     POKE 756,224
```

and run the program again, the first half of the character set will be used but the resulting display doesn't have the same romantic appeal!

CHARACTER DEFINITIONS

The shapes of all the displayable characters are defined by numbers held in a special area of memory. The XE allows you to create new symbols by entering new definitions for the characters to replace those held in memory. Each symbol in the character set is defined by the contents of eight memory locations, which correspond to a grid of eight rows; each row consisting of eight dots which may be set to the foreground or the background colour. (Each of these dots is called a *pixel*, which is a contraction of *picture element*. The display is built up from 69120 of these pixels.) Each location defines one row of the character, and each bit of the number in the location represents one pixel. If the bit is set to 1, the pixel is displayed in the

Memory Location:	0								
	1								
	2								
	3								
	4								
	5								
	6								
	7								
Bit:		7	6	5	4	3	2	1	0

The Character Grid

foreground colour; if the bit is set to 0 the pixel remains in the background colour.

In normal use, the character definitions are held in 2K of read-only memory (ROM), beginning at the location determined by the contents of location 756. The start address of the character set ROM is obtained by multiplying the contents of location 756 by 256. There are actually two sets of 128 characters, the first set including punctuation and maths symbols, numbers and letters and graphics characters. The second set is a copy of the first except that the 29 graphics symbols (normally obtained by holding down the CONTROL key and pressing a key) are replaced by a European character set, which contains characters like the £ sign. To switch between the two character sets use the command:

`POKE 756,204`

and to switch back:

`POKE 756,224`

INTERNAL CHARACTER CODES

The 130XE makes reference to the eight bytes defining a character by assigning a code, known as the *Internal Character Code* to each one. This is different from the ASCII code, so to find the eight bytes defining a character we must first convert the ASCII code of that character to internal code. This is done with the following formula:

$$\text{INTERNAL} = \text{ASCII} + ((\text{ASCII} < 32) * 64) - ((\text{ASCII} > 31 \text{ AND } \text{ASCII} < 96) * 32)$$

What this means is that the internal code is the same as ASCII for codes greater than 95; for codes less than 32 the internal code is greater by 64, and for ASCII codes between 31 and 96, the internal code is 32 less than ASCII.

Having worked out the internal code of a character, to find the eight bytes which define it, we multiply the internal code by eight, and add the start address of the character set:

$$\text{CHARACTER LOCATION} = (\text{INTERNAL CODE} * 8) + (256 * \text{PEEK}(756))$$

So the definition of the letter X (ASCII Code = 88; Internal Code = 56) in the standard character set is to be found at location:

$$56 * 8 + (256 * (\text{PEEK}(756))) = 57792$$

This short program will display the eight numbers defining any character.

```

1      REM DISPLAY CHARACTER DATA
5      OPEN#1,4,0,"K:"
10     PRINT "κ"
```

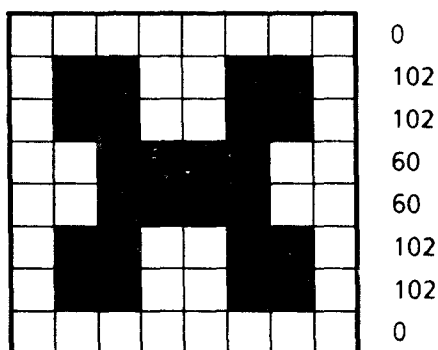
```

20   CSET = PEEK(756)*256
30   PRINT"ENTER ASCII CODE OF
    CHARACTER";
40   INPUT ASCII:PRINT"↵ "
45   PRINT "CHARACTER";CHR$(ASCII):
    PRINT
50   CODE=ASCII+((ASCII<32)*64)-
    ((ASCII>31 AND ASCII<96)*32)
60   PRINT"ASCII CODE ";ASCII;"
    INTERNAL CODE ";CODE:PRINT
70   CHAR = CODE*8 + CSET
80   FOR Z=CHAR TO CHAR+7
90   PRINT PEEK(Z):NEXT Z
100  PRINT:PRINT "PRESS ANY KEY TO
    CONTINUE"
110  GET#1,X:GOTO 10

```

For example, taking the letter X, the program should display the numbers 0, 102, 102, 60, 60, 102, 102, 0. Having found the eight numbers, we can

128 64 32 16 8 4 2 1



The Character X

convert them to binary form and fill in the grid to produce a letter X.

Notice that the top and bottom lines of the character are left blank. This is to leave a space between lines of text on the screen.

REDEFINING CHARACTERS

You may be wondering how it is possible to create new characters when the standard character definitions are held in ROM - it's difficult to write new numbers into read-only memory! The answer is that we have to tell the computer to look somewhere else for the character shapes, and define the new characters in RAM. We have used this technique before to swap between Standard and European character sets. By **POKE**ing the high byte of the start address of the character set into location 756 we changed the memory from which the character definitions were taken.

Location 756 is the place where the Atari stores the high byte of the start address of character data (ie to find the address of the start of character data you must multiply the contents of location 756 by 256). This information is used by the chip which controls all the display of the Atari, known as the ANTIC chip. Location 756 is a copy of one of the registers on this chip, known as a *shadow register*. The contents of the shadow register are copied to the chip every sixtieth of a second and used to dictate whereabouts in ROM the character set is to be found.

This pointer can just as easily be made to point to an area of RAM - try this:

LIST the program to show some characters on the screen, then type:

```
POKE 756,2
```

The program listing turns into garbage because the ANTIC chip is taking data for characters from an area of RAM which contains system variables, *not* character data. Press RESET to return to normal viewing!

To make use of user defined characters you must first copy all the characters in a set into the RAM, and tell the ANTIC chip where to find them. You can then alter some (or all) of the characters to your own designs.

The first step in redefining characters then is to copy the entire character set into RAM, in order that we can modify some or all of the data. The following program does this for you.

```
10      REM COPY CHARACTER SET
20      POKE 106,PEEK(106)-4
30      GRAPHICS 0
40      FOR Z=0 TO 124
50      PRINT CHR$(Z);
60      IF Z=26 THEN Z=31
70      NEXT Z
75      FOR DELAY = 0 TO 500:NEXT
      DELAY
80      CSET=PEEK(756)*256
90      RAMSET = 256*PEEK(106)
100     POKE 756,PEEK(106)
110     FOR Z=0 TO 1023
120     POKE RAMSET+Z,PEEK(CSET+Z)
130     NEXT Z
140     PRINT:PRINT "COPYING COMPLETE"
```

Line 20 reserves space in RAM for the character set; location 106 contains the page number of the top of memory, i.e.

$(256 * \text{PEEK}(106)) - 1$

is the address of the top of free RAM. By **POKE**ing a value of four less into location 106, we protect the top four pages of memory (1024 bytes) from use by BASIC. It is here that we will store our character set. Line 70 sets mode 0, and in doing so forces the computer to check location 106 to find out how much memory is available. Lines 40 to 70 put the characters set on the screen, omitting the control characters like screen clear and cursor characters. After a short delay the character set pointer is moved to our reserved area of RAM by line 80, and the character data copied from ROM to RAM. As it is copied, you will see the garbage which used to be the character set return to normal. The computer will behave as usual after copying, the difference being that character data is now being taken from RAM. Now that the character set is in RAM we can change it to suit ourselves. As an example, if you add these lines to the last program and then **GOTO** 1000, all the characters will be turned upside down - you have redefined them!

```
200    END
1000   REM  INVERT CHARACTERS
1010   FOR Z=0 TO 1023 STEP 8
1020   FOR Y = 0 TO 7
1030   POKE RAMSET+Z+Y,PEEK(CSET+
      Z+(7-Y))
1040   NEXT Y
```

Although this is a good example to illustrate redefined character sets, if you ever do want to invert the character set, there is a way of doing so which requires much less programming effort:

```
POKE 755,6
```

This will invert the character set, and

POKE 755,2

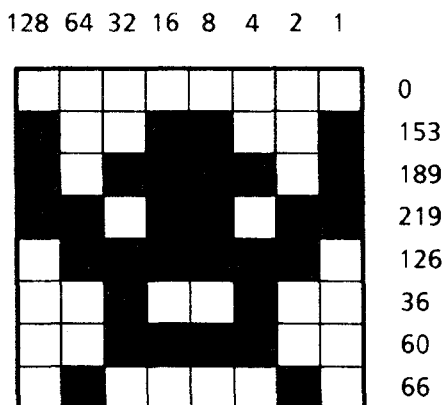
will return it to normal.

SPACE INVADERS

A more typical use of user defined characters would be to create a set of invader characters for a game, or a foreign character set such as Russian or Greek, or even a set of symbols for use in some specialist application.

Let's take the most exciting of these and create some invaders!

Firstly we take the eight by eight grid from the beginning of this chapter and draw our design. Now



Space Invader Character

to calculate the data statements required to specify the invader character, take one row at a time, and if a bit is set, write down the value of that bit (the numbers at the top of the grid). Continue for all eight bits in the row, and add up the numbers you

obtained. For example, in the bottom row, the sum is:

$$64 + 2 = 66$$

and for the top row where no bits are set the answer is 0.

In this way, you can obtain eight numbers which define your character.

All that remains is to chose the character you wish to reprogram, find the start address of the data defining it, using the formula on page 117, and replace the data with your own.

If you add the following lines to the program on page 120, the @ symbol will be turned into an invader.

```
200 CHAR = 32*8+RAMSET
210 FOR Z=0 TO 7
220 READ D
230 POKE CHAR+Z,D
240 NEXT Z
250 DATA 0,153,189,219,126,36,60,
      66
```

From now on, until you press RESET, enter another graphics mode or redefine it, the @ symbol will appear as an invader.

While on the subject of invaders, it is quite easy to achieve their limited form of animation - by reprogramming several character sets, each containing a slightly different version of the character. To animate, simply move the character set data pointer (location 756) to each set in turn.

CHARACTER GENERATOR PROGRAM

This program allows you to design new characters on the screen and add them to the character set. The characters are created by filling in a large grid on the screen, which is then used to program the character.

```

10      REM *****
11      REM *
12      REM * CHARACTER GENERATOR *
13      REM *
14      REM *****
15      REM
100     GOSUB 20000:REM SET UP
110     GOTO 10000:REM MAIN PROGRAM
190     REM
191     REM *****
192     REM *
193     REM * REDEFINE CHARACTERS *
194     REM *
195     REM *****
196     REM
200     PRINT "↵"
210     PRINT "ENTER ASCII CODE OF
CHARACTER (0-127)":INPUT ASCII
220     IF ASCII<0 OR ASCII>127 THEN
200
225     CODE=ASCII+((ASCII<32)*64)-
((ASCII>31 AND ASCII<96)*32)
230     CHARDATA=CODE*8+RAMSET
295     REM STORE CHARACTER DATA IN
ARRAY
300     PRINT"↵":GOSUB 900
310     FOR R=1 TO 8:GOSUB 1000:NEXT R
395     REM DISPLAY CHARACTER
400     GOSUB 2000
410     X=1:R=1
500     B=PEEK(CC)
510     REM MODIFY CHARACTER

```

```

520   POKE 764,255:GOSUB 3000:POKE
      764,255
525   IF EXIT=1 THEN EXIT=0:RETURN
530   IF CL=1 THEN CL=0:GOTO 400
540   CC=GRIDST+X+40*R
550   GOTO 500
890   REM
891   REM *****
892   REM *
893   REM * DISPLAY CHARACTER SET *
894   REM *
895   REM *****
896   REM
900   FOR Z=0 TO 124
910   PRINT CHR$(Z);
920   IF Z=26 THEN Z=31
930   NEXT Z
940   RETURN
990   REM
991   REM *****
992   REM *
993   REM * STORE CHARACTER DATA *
994   REM *
995   REM *****
996   REM
1000  N=PEEK(CHARDATA+R-1)
1010  X=8
1020  A(R,9)=N:REM ROW TOTAL
1030  REM STORE BIT PATTERN
1040  IF N/2=INT(N/2) THEN A(R,X)=0:
      GOTO 1050
1045  A(R,X)=1
1050  N=INT(N/2):X=X-1
1060  IF N>=1 THEN 1040
1070  FOR Z=X TO 1 STEP -1
1080  A(R,Z)=0
1090  NEXT Z
1100  RETURN
1990  REM
1991  REM *****
1992  REM *

```

```
1993 REM * DISPLAY CHARACTER *
1994 REM * *
1995 REM *****
1996 REM
2000 PRINT
2010 FOR R=1 TO 8
2020 FOR X=1 TO 8
2030 IF A(R,X)=1 THEN PIXEL=84:GOTO
      2040
2035 PIXEL=14
2040 POKE GRIDST+X+(R*40),PIXEL
2050 NEXT X
2060 POSITION 18,6+R:PRINT A(R,9)
2070 NEXT R
2075 POSITION 2,20:PRINT"ESC
      RETURN - = + * CLEAR
      PROGRAM":REM BOLD CHARACTERS
      IN REVERSE
2080 RETURN
2990 REM
2991 REM *****
2992 REM * *
2993 REM * MOVE CURSOR *
2994 REM * *
2995 REM *****
2996 REM
3000 REM CHECK KEYBOARD
3010 K=PEEK(764):REM INTERNAL CODE
      OF KEY PRESSED
3020 REM FLASH CURSOR
3030 CH=PEEK(CC):C=CH-((CH>128)*
      128)+((CH<128)*128)
3040 POKE CC,C:FOR PAUSE=0 TO
      10:NEXT PAUSE
3050 REM CHECK KEY INPUT
3060 IF K=255 THEN 3010
3080 IF K=6 AND X>1 THEN POKE CC,B:
      X=X-1:RETURN:REM LEFT
3090 IF K=7 AND X<8 THEN POKE CC,B:
      X=X+1:RETURN:REM RIGHT
```

```

3100 IF K=15 AND R<8 THEN POKE CC,B
      :R=R+1:RETURN:REM DOWN
3110 IF K=14 AND R>1 THEN POKE CC,B
      :R=R-1:RETURN:REM UP
3120 IF K=33 THEN A(R,X)=1-A(R,X):
      GOTO 3200:REM CHANGE PIXEL
3130 IF K=10 THEN GOSUB 4020:REM
      REPROGRAM CHARACTER
3140 IF K=118 THEN GOSUB 3300:REM
      CLEAR CHARACTER
3150 IF K=12 THEN X=1:R=1:POKE CC,B
      :RETURN:REM TOP LH CORNER OF
      GRID
3160 IF K=28 THEN EXIT=1:RETURN
3170 POKE 764,255:RETURN
3190 REM
3191 REM *****
3192 REM *
3193 REM * CHANGE PIXEL *
3194 REM *
3195 REM *****
3196 REM
3200 IF B=14 THEN POKE CC,84:RETURN
3210 POKE CC,14:RETURN
3290 REM
3291 REM *****
3292 REM *
3293 REM * CLEAR CHARACTER *
3294 REM *
3295 REM *****
3296 REM
3300 FOR R=1 TO 8:FOR X=1 TO 9
3310 A(R,X)=0
3320 NEXT X
3325 POSITION 18,R+6:PRINT"
      <<<<" ;:PRINT A(R,9):NEXT R
3330 CL=1:RETURN
3990 REM
3991 REM *****
3992 REM *
3993 REM * REPROGRAM CHARACTER *

```

```

3994 REM *
3995 REM *****
3996 REM
4020 FOR RT=1 TO 8
4030 FOR XT=1 TO 8
4040 D=D+(2^(8-XT))*A(RT,XT)
4050 NEXT XT
4060 POKE CHARDATA+RT-1,D
4065 A(RT,9)=D
4070 D=0
4080 POSITION 18,RT+6:PRINT"
      <<<<" ;:PRINT A(RT,9):NEXT RT
4090 RETURN
9990 REM
9991 REM *****
9992 REM *
9993 REM * MAIN MENU *
9994 REM *
9995 REM *****
9996 REM
10000 PRINT"↵":CLOSE#1
10010 POSITION 9,2:PRINT"CHARACTER
      GENERATOR"
10020 POSITION 2,5:PRINT"CHANGE
      CHARACTERS":REM BOLD
      CHARACTERS IN REVERSE
10030 PRINT:PRINT:PRINT"LOAD
      CHARACTER SET FROM TAPE"
10040 PRINT:PRINT:PRINT"SAVE
      CHARACTER SET TO TAPE"
10050 PRINT:PRINT:PRINT"ESC TO END
      PROGRAM"
10060 OPEN#1,4,0,"K:"
10065 GET#1,K
10070 IF K=67 THEN GOSUB 200:GOTO
      10000:REM C
10080 IF K=76 THEN GOSUB 11000:GOTO
      10000:REM L
10090 IF K=83 THEN GOSUB 12000:GOTO
      10000:REM S
10100 IF K=27 THEN END:REM ESC

```

```
10110 GOTO 10065
10990 REM
10991 REM *****
10992 REM * *
10993 REM * LOAD CHARACTER SET *
10994 REM * *
10995 REM *****
10996 REM
11000 CLOSE#1:PRINT"↵":OPEN#1,4,
      0,"C:"
11010 FOR BYTE=0 TO 1023
11020 GET#1,TEMP
11030 POKE RAMSET+BYTE,TEMP
11040 NEXT BYTE
11050 CLOSE#1:RETURN
11990 REM
11991 REM *****
11992 REM * *
11993 REM * SAVE CHARACTER SET *
11994 REM * *
11995 REM *****
11996 REM
12000 CLOSE#1:PRINT"↵":OPEN#1,8,
      0,"C:"
12010 FOR BYTE=0 TO 1023
12020 PUT#1,PEEK(RAMSET+BYTE)
12030 NEXT BYTE
12040 CLOSE#1:RETURN
11990 REM
11991 REM *****
11992 REM * *
11993 REM * SETTING UP ROUTINES *
11994 REM * *
11995 REM *****
11996 REM
20000 DIM A(8,9)
20010 POKE 106,PEEK(106)-4
20020 GRAPHICS 0
20025 SETCOLOR 2,2,6:SETCOLOR
      1,2,0:SETCOLOR 4,2,8
20026 POSITION 7,5
```

```
20030 PRINT"RELOCATING CHARACTER
      SET"
20035 POSITION 5,22:PRINT "Please
      wait approx. 25 seconds"
20040 CSET=256*PEEK(756)
20045 POKE 752,1:REM TURN OFF CURSOR
20050 RAMSET=256*PEEK(106)
20060 FOR BYTE=0 TO 1023
20070 POKE RAMSET+BYTE,PEEK
      (CSET+BYTE)
20080 NEXT BYTE
20085 POKE 756,PEEK(106)
21000 DLIST=PEEK(560)+256*PEEK(561)
21010 SCRNST=PEEK(DLIST+4)+256*
      PEEK(DLIST+5)
21020 GRIDST=SCRNST+245
25000 RETURN
```

The program allows you to redefine any character, and to save character sets on tape and reload them. You can use the new character sets with other programs.

The first thing the program does is create some space for the character set and then copy the character set into RAM. Lines 20020 to 20080 do this. The rest of the lines in the setting up routine find the start of screen memory (SCRNST) and decide whereabouts on the screen the character grid is to be placed (GRIDST).

Lines 10000 to 10110 then display the menu, giving you the option to change a character or to load or save a character set. Pressing the ESC key stops the program.

The redefining of characters is controlled by the routine from 200 to 550. When you input the code of the character you wish to alter, the line 310 calls the routine at lines 1000 to 1100 to store the data of

the character in an array. The character is then displayed on the screen by the routine at 2000.

The modification of the character is performed by the section of program at 3000. You may move the cursor around the character grid with the cursor keys and use the space bar to set or clear the pixels of the character. A menu of the options open to you is displayed at the bottom of the screen. The ESC key returns you to the main menu, while the RETURN key moves the cursor to the top left corner of the grid. To erase all the data for a character, hold down SHIFT and press the CLEAR key.

When you have finished designing the new character, pressing P will store the new definition in character memory. Lines 4000 to 4090 perform this function.

Saving a character set onto tape is performed by the subroutine at 12000. The character sets are reloaded into the machine by the routine at 11000.

WARNING

Do not reset the Atari by using the RESET key or the new characters will be lost.

MULTICOLOUR CHARACTERS

There are two more graphics modes which display characters, and with these the colour of the individual pixels which make up a character can be specified. This means you can have up to four different colours within each character.

The two multicolour character modes are modes 12 and 13, and are obtained in the usual way. A four line mode 0 window is obtained but as with modes 1 and 2, adding 16 to the mode number in a

GRAPHICS command will allow the entire display to be used for multicolour characters.

Mode 12 allows 40 columns by 24 rows of characters – the same as mode 0. Mode 13 allows 40 columns by 12 rows of multicolour characters, the characters being twice as tall as those in mode 12.

Type in the following program to see some multicolour characters.

```
10    GRAPHICS 12
20    PRINT#6;"MULTICOLOUR
      CHARACTERS"
```

You may just be able to discern the message – it is certainly multicoloured, but hardly legible!

This is because in mode 12 (and mode 13) the information defining a character is interpreted differently by the computer. A multicolour character is still defined by eight bytes of character memory, but each pixel comprising the character is represented by two bits rather than one. This means that multicolour characters are only four pixels wide, but the pixels are double the normal width so characters are the same size as in other modes. The extra information available in two bits is used to select the colour of that pixel.

The multicolour characters can be designed on a similar eight by eight grid to that used for standard characters, as shown opposite.

There are four possible combinations of two bits, 00, 01, 10 and 11, and each of these represents a different colour or rather specifies a different colour register.

Press the **RESET** key (to return the colour registers to their default values) and type in the following

Memory Location:	0								
	1								
	2								
	3								
	4								
	5								
	6								
	7								
Bit:		7	6	5	4	3	2	1	0

The Multicolour Character Grid

program, which puts a letter Q in the centre of a mode 13 screen. Mode 13 is used for clarity, the principles apply equally to mode 12.

```

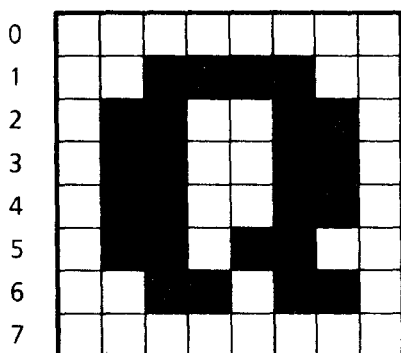
10    GRAPHICS 13
20    POSITION 18,4
30    PRINT#6;"Q"

```

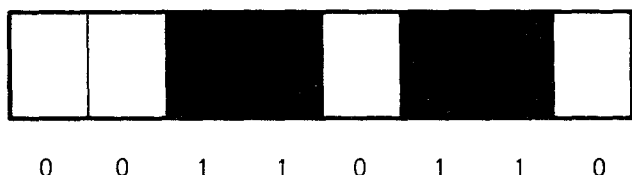
The Q is just recognisable, and if you look closely comprises three different colours. To see how these colours originate, let's look at the eight bytes defining the letter Q in character memory.

The diagram shows how the data for letter Q is stored in the 130XE. In mode 0, each pixel is represented by a single bit. In modes 12 and 13 however, the computer treats this data as four groups of two bits, each specifying the colour register from which information concerning that pixel is to be taken.

If we take one row (one byte) and examine it in more detail we will see how colour information is

*The Character Q*

derived. Take row 6 as it has an example of each of the four possible bit combinations.

*Row 6 of the character Q*

The first bit pair is 00, which means the pixel represented by that bit pair takes its colour from register 4. The second pair, 11, specifies register 2, the third register 0 and the fourth register 1. So the four pixels comprising the seventh row of the letter Q in our example should each be in the default colours of those four registers. If you look closely at the screen you will find this to be the case.

A slight variation on this is that if the characters placed on a mode 12 or 13 screen are in inverse video (i.e. if bit seven of the character code is set), the colour of pixels specified by the bit pair 11 is taken from register 3 rather than register 2.

To see this try typing:

```
GRAPHICS 12:PRINT#6;"ABCDEFGFGABCDEFGF"
```

with the second half of the message in inverse video. Areas of the first half of the message appearing blue (register 2 default colour) are violet in the second half of the message (the register 3 default colour).

You will have realised by now that to make full use of mode 12 and 13 requires a special character set which will have to be defined in the usual way, but taking into account the extra considerations of colour register requirements. The applications are wide ranging and this is an economical way (in terms of memory requirements) to achieve colourful displays.

CHAPTER 13

GRAPHICS

In addition to the five text displaying modes described in the last chapter, the Atari 130XE has eleven graphics modes. The different modes offer many variations in the number of colours available, and in resolution (the number of rows and columns in the display). The same commands are used to control displays in all the graphics modes.

Graphics Modes

The modes are selected by the command **GRAPHICS** followed by the number of the desired mode. The display modes are numbered from 0 to 15; modes 0, 1, 2, 12 and 13 display text, as described in Chapter 12; modes 3 to 11 and 14 and 15 are the graphics displays. Try the command

GRAPHICS 5

You will see the screen turn black, except for a four line high blue area at the foot of the screen. The black area is for drawing graphics, and the blue region is a text 'window' - a smaller version of the mode 0 text display. As in modes 1 and 2, you can remove the text window by adding 16 to the number of the mode following the **GRAPHICS** command, but you cannot control windowless displays directly from the keyboard as the XE will snap back into mode 0 as soon it has obeyed the **GRAPHICS** command so that it can display the 'Ready' message.

Colour in Graphics Modes

The different graphics modes use differing numbers of colours, but they are all controlled in the same way. As described in Chapter 12, there are five colour registers which control the display colours. The SETCOLOR command is used to select the colour produced by each register.

The colours of objects displayed on the screen are set by indicating which colour register is to control their colour. The command COLOR selects the colour to be used. The command is followed by a number which indicates the colour selection, but the number is not always that of the required colour register, which can be a little confusing.

In mode 5, four colors are used. The background colour is controlled by colour register 4, register 3 is not used, and registers 0, 1 and 2 hold the 'foreground' colours. The colours to be used are selected by COLOR as follows:

COLOR 0 Selects the background colour.

COLOR 1 Selects the register 0 colour.

COLOR 2 Selects the register 1 colour.

COLOR 3 Selects the register 2 colour.

When you first enter mode 5, the colour selected in the background colour (COLOR 0) held in colour register 4. Type

```
GRAPHICS 5  
COLOR 1
```

so that you are ready to start drawing.

Points and lines

To light up a point (or *pixel*) on the screen the command **PLOT** is used. The command is followed by two numbers – the row and column in which the point lies.

```
PLOT 20, 10
```

lights a pixel on the screen in the colour selected by the previous **COLOR** command. Try **PLOT**ting a few points around the screen. Mode 5 has 80 columns, numbered from 0 to 79, and 40 rows – 0 to 39. If you use numbers which are too large the XE will print 'ERROR – 141'; if you use numbers less than zero the Atari prints 'ERROR – 3'.

Try changing the colour of the points you **PLOT** by using **COLOR 2** or **COLOR 3**. **COLOR 0** will give invisible points as it selects the background colour. This means that you can erase pixels by returning them to the background colour.

When you have finished experimenting with the **PLOT** command clear the screen again by typing:

```
GRAPHICS 5
```

The command **DRAWTO** draws lines on the display. Set a pixel by typing:

```
COLOR 1  
PLOT 5,5
```

and then try:

```
DRAWTO 60,5  
DRAWTO 60,35  
DRAWTO 5,5
```

the XE draws a line from the position of the last **PLOT** or **DRAWTO** to the point specified by the coordinates after the **DRAWTO** command.

You now know enough to produce line drawings of all kinds. Try a few experiments.

This program produces a lacy pattern by drawing lines close together on the mode 7 display, which has twice as many rows and columns as the mode 5 display but is otherwise the same.

```
10      REM LACE
20      GRAPHICS 7
30      COLOR 1
40      FOR X=0 TO 159 STEP 2
50      PLOT 79, 0
60      DRAWTO X, 79
70      NEXT X
```

Filling in the Colours

As well as drawing points and lines on the screen, you can fill in areas of colour. The command used is **XIO**, but to fill in an area you must first define the outline of the shape. The sequence is as follows:

- 1 **PLOT** the bottom right-hand corner of the shape to be filled.
- 2 **DRAWTO** the top right-hand corner.
- 3 **DRAWTO** the top left-hand corner.
- 4 Use **POSITION** to define the bottom left-hand corner of the shape.
- 5 **POKE** the number of the colour you are using into location 765.

6 Then use the command **XIO 18, #6, 0, 0, "S:"**.

This program draws a solid rectangle on the mode 5 display in colour number 1, with its corners at (5, 5), (75, 5), (5, 75) and (75, 75):

```
10      REM FILL WITH XIO
100     GRAPHICS 5
110     COLOR 1
120     PLOT 75, 35
130     DRAWTO 75, 5
140     DRAWTO 5, 5
150     POSITION 5, 35
160     POKE 765, 1
170     XIO 18, #6, 0, 0, "S:"
```

You can fill shapes other than rectangles, within certain limits:

- 1 The bottom of the shape must be horizontal, and
- 2 The top must be horizontal, or slope down to the right.

This is because of the way the shapes are filled. The XE fills in the colour by drawing horizontal lines to the right from a line between the top left-hand corner and the bottom left-hand corner, as far as the line defining the right-hand edge. This means that if the top of your shape slopes up to the right it won't all get filled. If the bottom left-hand corner is higher than the bottom right-hand corner then not all the shape is filled. If the bottom left-hand corner is lower than the bottom right-hand corner then the colour spills out across the full width of the screen.

The next program uses **XIO** to fill random blocks on the screen, producing instant abstract art. The

program uses mode 10, which is a nine-colour mode. Mode 10 is explained later in the chapter.

```
10    REM ABSTRACT XIO
20    GRAPHICS 10
30    XMAX=79: YMAX=191
40    POKE 704, 56
50    POKE 705, 24
60    POKE 706, 168
70    POKE 707, 88
80    SETCOLOR 0, 4, 10
90    SETCOLOR 1, 10, 8
100   SETCOLOR 2, 12, 4
110   SETCOLOR 3, 13, 14
120   SETCOLOR 4, 14, 6

200   X = RND(0) * XMAX
210   Y = RND(0) * YMAX
220   X2 = X+RND(0) * (XMAX - X)
230   Y2 = Y+RND(0) * (YMAX - Y)
240   C = RND(0) * 8
250   COLOR C
260   PLOT X2, Y2
270   DRAWTO X2, Y
280   DRAWTO X, Y
290   POSITION X, Y2
300   POKE 765, C
310   XIO 18, #6, 0, 0, "S:"
320   GOTO 200
```

The final graphics command is **LOCATE**. This command moves the invisible graphics cursor to a point on the screen and reads the colour of the pixel at that point into a variable.

LOCATE X, Y, C

stores the colour (the colour register number) at X, Y into C. The next program plots a point in a

randomly selected colour, and then finds its colour using **LOCATE**.

```
10    REM LOCATE DEMONSTRATION
20    GRAPHICS 3
30    COLOR RND(0)*3
40    PLOT 30, 15
40    LOCATE 30, 15, COL
60    PRINT COL
```

This program uses mode 3, which is similar to mode 5 but with only half the number of rows and columns.

THE GRAPHICS DISPLAY MODES

So far we have used only modes 3, 5 and 7. We will now describe in detail all the graphics modes available on the 130XE. You can use **PLOT**, **DRAWTO**, **LOCATE**, **XIO** and **COLOR** commands in all the modes, the only differences being in the available colours, and in the resolution of the display.

Mode 3

This is a four colour mode. The background colour is held in colour register 4 and selected by **COLOR 1**, the other three colours are held in registers 0, 1 and 2 and are selected by **COLOR 1**, **COLOR 2** and **COLOR 3** respectively.

The display resolution is 40 columns and 24 rows, or 20 rows if the text window is shown.

Mode 4

This mode displays 48 rows of 80 columns, or 40x80 if the text window is shown.

Two colours may be used; the background colour is controlled by colour register 4 and selected by COLOR 0, the second colour is controlled by colour register 0 and selected as COLOR 1.

Mode 5

Four colours are available in this mode, controlled in the same way as those in mode 3.

The resolution is the same as that of mode 4 - 48 rows of 80 columns, or 40x80 with the text window.

Mode 6

This mode has two colours, controlled as in mode 4.

The resolution is 96 rows of 160 columns, or 80x160 if the text window is displayed.

Mode 7

This mode has four colours, controlled as in mode 3, with the same resolution as mode 6.

Mode 8

Mode 8 has the highest resolution of all the display modes, 192 rows of 320 columns are allowed, or 160x320 with the text window.

The colour available is similar to that used in mode 0: only one colour can be used, with two possible luminances. The colour and the luminance of the background are controlled by colour register 2, and selected by COLOR 0. The second luminance is controlled by the luminance of colour register 1, and may be selected by COLOR1.

Because this mode uses the colour registers in the same way as the mode 0 text display, the

background does not appear black when you select this mode, but instead retains the colour of the mode 0 screen.

Mode 9

This mode, like mode 8, uses only one colour. However, unlike any other display mode, this mode gives 16 different luminances. The colour is controlled by colour register 4, but the luminance is controlled by the **COLOR** command and the luminance set in register 4 has no effect. The background is set to a luminance of 0.

The resolution in this mode is 192 rows of 80 columns. The text window may not be displayed in this mode.

The example program shows all 16 luminances of colour number 3 on the screen.

```
10      GRAPHICS 9
20      SETCOLOR 4, 3, 0
30      FOR LUM=0 TO 15
40      COLOR LUM
50      FOR Z=0 TO 11
60      PLOT 0, LUM*12 + 2
70      DRAWTO 79, LUM*12 + 2
80      NEXT Z
90      NEXT LUM
100     GOTO 100
```

Mode 10

This mode has a similar resolution to mode 9; 192 rows of 80 columns. Again the text window may not be displayed. Nine colours are displayable, each in eight luminances. Five of the colours are specified by the colour registers, and can be set by **SETCOLOR**, the other four are controlled by registers which are usually reserved for use with

Player-Missile graphics (see Chapter 15). These registers are memory locations 704, 705, 706 and 707, and cannot be set using **SETCOLOR**; instead you must use **POKE** to place a value for colour and luminance into the register. The value to use is:

$$(\text{Colour Number} * 16) + \text{Luminance}$$

Eight of the nine colours can then be selected by **COLOR**, the ninth sets the colour of the border of the screen and can not be used for drawing. The registers and their **COLOR** codes are:

COLOR No.	Register
0	Location 705
1	Location 706
2	Location 707
3	Colour Register 0
4	Colour Register 1
5	Colour Register 2
6	Colour Register 3
7	Colour Register 4
Border	Location 704

Mode 11

This mode is the 'opposite' of mode 9, giving sixteen colours but only one luminance, instead of one colour at sixteen luminances. The resolution is again 192 rows of 80 columns, and the text window is once again unavailable.

To control the colours you must first set the colour in register 4 to 0, or the full set of sixteen colours will not be displayed. The colours are selected using **COLOR** and the colour number.

This example program draws the Atari logo in sixteen colours on a mode 11 display.

```
10    REM ATARI LOGO
20    GRAPHICS 11
30    FOR Y=0 TO 191
40    COLOR Y/16
50    X = SQR(191-Y)*2
60    PLOT X,Y
70    DRAWTO X+7, Y
80    PLOT 36, Y
90    DRAWTO 43,Y
100   PLOT 72-X, Y
110   DRAWTO 79-X, Y
120   NEXT Y
200   GOTO 200
```

Modes 12 and 13

These modes display text and are described in Chapter 12.

Mode 14

Mode 14 is similar to modes 4 and 6, but has higher resolution. There are two colours, used in the same way as those in modes 4 and 6: the background is controlled by register 4 and is selected by COLOR 0, the foreground is controlled by register 0 and is selected by COLOR 1. A text window may displayed with colours controlled as in mode 0.

The resolution of this mode is 192 rows of 160 columns, or 160x160 if the text window is used.

Mode 15

This mode is a four colour version of mode 14, with the colours controlled as in modes 3, 5 and 7.

Memory Use

The various displays require different amounts of memory, with the higher resolution modes needing as much as 8k of storage. This may cause problems if you have a long program, particularly on earlier machines such as the 600XL with only 16k of RAM. Be careful when writing long programs, and keep checking the available store using `FRE(0)`. For example, if you want to run a program with a mode 8 display, you will need at least 7200 free bytes in mode 0. There is a table showing the memory used by the display modes in Appendix 5.

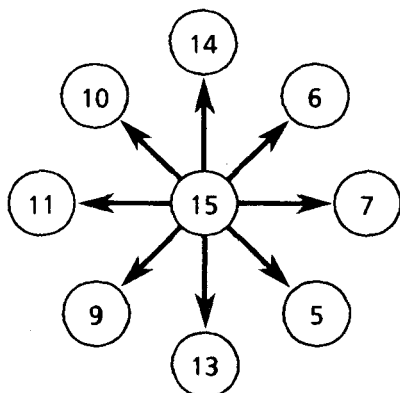
JOYSTICKS AND PADDLES

The Atari 130XE can accomodate two joysticks, or four paddles, which are plugged into the ports on the right-hand side of the computer. Both joysticks and paddles may easily be used to control BASIC programs.

There are two BASIC commands used to read joysticks. The joystick ports are numbered 0 and 1, *not* 1 and 2 as is stamped on the computer's casing, with port 0 at the front. To find the position of a joystick, use the function `STICK()`, which returns a value for the position of the joystick in the specified port. A second function, `STRIG()` gives a value of 0 if the Fire button of the selected joystick is pressed, or 1 if it is not. The values returned by `STICK()` are illustrated in the diagram below.

Paddles are used in a similar way. The a pair of paddles may be plugged into each port; the paddles in port 0 are numbered 0 and 1, those in port 1 are numbered 2 and 3.

The function `PADDLE()` returns a value of between 0 and 224 depending on the rotation of the



The Joystick Positions given by STICK()

controller of the specified paddle. PTRIG() is used to read the Fire button of a paddle.

The program listed below uses a joystick to control the drawing of lines on a graphics display.

```

10    REM JOYSTICK ARTIST
20    REM SET UP VARIABLES AND
      LABELS
100   POKE 752,1: PRINT
      "↖↓↓↓▶SKETCHPAD"
110   GOSUB 20000
192   REM FIND JOYSTICK PORT
200   PRINT"↓↓↓↓▶PRESS FIRE TO
      START"
210   PORT=0
220   IF STRIG(PORT)=1 THEN PORT=
      PORT=0: GOTO 220

992   REM PICTURE
1000  GRAPHICS 7
1010  X=0:Y=0
1020  GOSUB MESSAGE
1030  POKE 82,0
1040  GOSUB COLOUR
  
```

```
1092 REM JOYREAD
1100 JOY=STICK(PORT)
1110 IF JOY=15 THEN GOTO JOYREAD
1120 XNEW=X+DXY(JOY,0)
1130 YNEW=Y+DXY(JOY,1)
1140 IF XNEW<0 OR XNEW>159 OR
    YNEW<0 THEN GOTO JOYREAD
1150 IF YNEW>79 THEN GOSUB TEXT:
    GOTO JOYREAD

1190 REM
1200 IF STRIG(PORT)=0 THEN GOTO
    1250
1210 COLOR BACKGR
1220 PLOT X,Y
1230 COLOR COL
1240 LOCATE XNEW,YNEW,BACKGR
1250 PLOT XNEW,YNEW
1260 X=XNEW:Y=YNEW
1300 GOTO JOYREAD

1992 REM TEXT
2000 COLOR BACKGR
2010 PLOT X,Y
2020 X=INT(X/4):Y=0:IF X<2 THEN X=2
2030 POKE 752,0:REM TURN ON CURSOR
2040 POKE 656,Y:POKE 657,X:POKE
    658,0
2050 PRINT"↔";

2092 REM CHECK JOY
2100 IF STRIG(PORT)=0 THEN GOSUB
    SWITCH
2110 JOY=STICK(PORT)
2120 IF JOY=15 THEN GOTO CHECKJOY
2200 XNEW=X+DXY(JOY,0)
2210 YNEW=Y+DXY(JOY,1)
2220 IF XNEW<0 OR XNEW>39 OR YNEW>3
    THEN GOTO CHECKJOY
2230 IF YNEW<0 THEN GOTO SCREEN
2240 POKE 656,YNEW:POKE 657,XNEW
```

```

2245 PRINT"↔"
2250 X=XNEW:Y=YNEW
2255 FOR T=1 TO 10:NEXT T
2260 GOTO CHECKJOY

2292 REM SCREEN
2300 X=X*4+2:Y=79
2310 POKE 752,1:PRINT"↔";
2320 PLOT X,Y
2330 RETURN

2492 REM SWITCH
2500 IF X<7 THEN COL=0:GOSUB
MESSAGE:GOSUB CHCURSOR:RETURN
2510 IF X>8 AND X<15 THEN COL=1:
GOSUB MESSAGE:GOSUB CHCURSOR:
RETURN
2520 IF X>16 AND X<23 THEN COL=2:
GOSUB MESSAGE:GOSUB CHCURSOR:
RETURN
2530 IF X>24 AND X<31 THEN COL=3:
GOSUB MESSAGE:GOSUB CHCURSOR:
RETURN
2540 IF X>32 AND X<38 AND Y=1 THEN
POP:COL=2:GOTO PICTURE
2550 RETURN

2592 REM CHCURSOR
2600 POKE 656,Y
2610 POKE 657,X
2620 POKE 752,0
2630 PRINT"↔";
2650 RETURN

2992 REM MESSAGE
3000 POKE 752,1
3010 POKE 82,0
3020 PRINT"↖"
3030 PRINT"  ERASE    RED    YELLOW
        BLUE    CLEAR"

```

```
3040 IF COL=0 THEN POKE 656,1:POKE
      657,2:PRINT"ERASE":REM TEXT IN
      REVERSE
3050 IF COL=1 THEN POKE 656,1;POKE
      657,9:PRINT" RED ":REM TEXT IN
      REVERSE
3060 IF COL=2 THEN POKE 656,1;POKE
      657,17:PRINT"YELLOW":REM TEXT
      IN REVERSE
3070 IF COL=3 THEN POKE 656,1;POKE
      657,25:PRINT" BLUE ":REM TEXT
      IN REVERSE
3100 RETURN
```

```
3992 REM COLOUR
4000 SETCOLOR 0,3,8
4010 SETCOLOR 1,1,12
4020 SETCOLOR 2,9,4
4030 SETCOLOR 4,0,12
4100 RETURN
```

```
19992 REM SET UP
20000 DIM DXY(15,1)
20010 FOR J=5 TO 15
20020 FOR K=0 TO 1
20030 READ DXY
20040 DXY(J,K)=DXY
20050 NEXT K
20060 NEXT J
20070 COL=2
```

```
20092 REM DEFINE LABELS
20100 PICTURE=1000
20110 JOYREAD=1100
20120 TEXT=2000
20130 CHECKJOY=2100
20140 SCREEN=2300
20150 SWITCH=2500
20160 CHCURSOR=2600
20170 MESSAGE=3000
20180 COLOUR=4000
```

```
21000 RETURN
22000 DATA 1,1,1,-1,1,0,0,0,-1,1,-1,
          -1,-1,0,0,0,0,1,0,-1,0,0
```

To use this program you will need a joystick, which may be plugged into either of the ports. The program is controlled entirely by the joystick, with no keyboard actions at all.

After the setting up routines have been run, pressing the Fire button on the joystick will display a blank graphics 7 screen, with a text window. The cursor will be visible as a yellow spot in the top left corner of the screen. Use the joystick to move the cursor around the screen, and hold down the Fire button while moving the cursor to draw lines.

To change colour or clear the screen, move the cursor down into the text window, position it over the command you require, and press Fire. There are four colours to choose from: red, yellow and blue, and the background colour (selected by ERASE).

If you find that the display is too fine in mode 7, you can change the program to run in mode 5 by altering line 1000, which specifies the mode, and also by altering the tests in lines 1140 and 1150 which check whether the cursor is running off the edge of the display. You will also need to alter line 2020, which calculates the cursor position in the text window after it crosses from the graphics display, and line 2300, which calculates the cursor position in the graphics display as the cursor leaves the text window.

CHAPTER 14

ADVANCED GRAPHICS

This chapter deals with the way in which the 130XE generates the various displays discussed in the last three Chapters. It describes how the computer controls the TV and how you can alter this mechanism to produce yet more varied displays.

THE ANTIC AND GTIA CHIPS

The Atari uses two special chips to handle the requirements of generating a display, called ANTIC and GTIA. The ANTIC chip takes the data concerning a display from display memory and passes it to the GTIA chip in the correct form for the display type currently selected.

The GTIA chip uses the data to control the TV and so create the display. To understand how this process works, you must understand some of the principles used to generate TV pictures.

TV PICTURES

The pictures on a TV screen are created by an electron beam which is directed at the phosphor coated inner surface of the screen. Where this beam strikes the screen, the phosphor glows. To create a full screen picture, the electron beam scans across the screen in rows, varying in intensity as it goes. This variation in intensity is dictated by information from the TV transmitter and produces a corresponding variation in the intensity with

which the phosphor glows. When the electron beam reaches the edge of the screen it has completed one *Scan Line* and is turned off and moved back to a position just below the last starting position, before the process begins again. This scanning continues until the electron beam reaches the bottom of the screen, at which point it starts all over again at the top. This process must happen many times a second to create a picture that doesn't flicker.

To create a colour picture, the screen is coated with three different types of phosphor which emit the colours red, blue or green when struck by the electron beam. The different phosphors are distributed in tiny dots or blocks over the screen and the colour TV signal must contain information about which of these points to direct the electron beam at, as well as luminance information.

In creating its displays the 130XE uses the same principle as the TV transmitter, with the ANTIC chip feeding data to the GTIA chip, which controls the electron beam and supplies colour and luminance information for each dot on the display.

THE DISPLAY LIST

The ANTIC is a microprocessor, like the 6502, and has its own program in RAM telling it how the data in screen memory is to be organised to create a display. This program is called the *display list*, and is placed in memory (immediately below screen memory) by the 6502 processor. The contents of the display list depend upon the graphics mode currently selected.

The Atari display is divided into groups of scan lines known as *display blocks*. The number of scan lines per display block depends upon the graphics mode. In mode 0 a display block is eight scan lines high, since a graphics 0 character is eight scan

lines high. In mode 8 a display block is only one scan line high, resulting in the 192 line vertical resolution in mode 8.

The size of a display block governs the way in which ANTIC reads data from the display memory, and hence the way in which it is displayed. Let's look at the display list and see how it works. Type in the following program which will print the display list on the screen.

```

10    GRAPHICS 0
20    DL=PEEK(560)+256*PEEK(561)
30    FOR Z=0 TO 31
40    POSITION 2+((Z>15)*18),Z-
      ((Z>15)*16)
50    PRINT DL+Z,PEEK(DL+Z)
60    NEXT Z
70    GOTO 70

```

The location of the start of the display list is stored in two memory locations: 560 and 561. The program calculates the address of the start of the display list and then prints the contents of the next 32 bytes of memory which comprise the display list. (Line 40 ensures that all the information is displayed clearly on the screen.)

If you run the program you will see a display like the one below.

The five figure numbers in columns one and three of the display are memory locations in which are stored the display list instructions given in the second and fourth columns.

15392	112	15408	2
15393	112	15409	2
15394	112	15410	2
15395	66	15411	2

15396	64	15412	2
15397	60	15413	2
15398	2	15414	2
15399	2	15415	2
15400	2	15416	2
15401	2	15417	2
15402	2	15418	2
15403	2	15419	2
15404	2	15420	2
15405	2	15421	65
15406	2	15422	32
15407	2	15423	60

The ANTIC processor reads the display list from top to bottom and the instructions in it control the display in that order.

The three 112 instructions at the start of the list instruct ANTIC to generate three eight-line display blocks of border colour.

Of the next three bytes, only the first (66) is an instruction; this tells ANTIC that display memory starts at the location given in the next two bytes of the display list. In a 16k machine these numbers will be as shown and tell ANTIC that display memory starts at

$64 + 256 \times 60$ or 15424.

To see that this is the start of display memory, stop the program and type:

POKE 15424,1 (or whatever the address is for
your machine)

You will see an exclamation mark (!) appear in the top left hand corner of the screen.

Following this instruction there are 23 '2's. This is the ANTIC instruction for a mode 0 display block. When the 130XE interpreted the **GRAPHICS 0** command in the program, it allocated sufficient memory for the display, and created the display list accordingly. The 23 '2's indicate that the display should comprise 24 display blocks of 8 lines each.

There is a discrepancy here, since there are only 23 code 2s but mode 0 is a 24 line display. There is another code 2 at the beginning of the display list but it isn't obvious. We said earlier that the 66 code indicated that the next two bytes in the list are to be read as a 16 bit address. In fact the code for this is 64 and this has been added to the code 2 specifying the first mode 0 display block of the screen (the top line of the display), in much the same way as you add 16 to a **GRAPHICS** instruction in BASIC to modify its effect. The 64 added to the '2' instruction is called a *modifier* and is discussed more fully later in this chapter.

Only the first of the last three bytes of the display list is an instruction. The code 65 tells ANTIC to jump back to the location specified in the following two bytes - the start of the display list.

The display list we have examined is fairly short and simple, being for a mode 0 display. For mode 8 it would be much longer, containing 192 instructions specifying one line display blocks, and some other codes

A list of the display list instructions is given in Appendix 9.

Armed with this information we can now make changes to the display list which will allow us to create more varied displays than have previously been possible. One of the things we can do is to scroll the display in all directions.

SCROLLING

We have met the term scrolling in Chapter 3 as the name given to the way in which characters on the screen move upwards and disappear off the top - as happens when a large program is listed.

However, scrolling is not restricted to this one direction. We can scroll the display both vertically and horizontally from within a program, by changing the address which ANTIC uses as the start of display memory.

As mentioned earlier, the address of the start of display memory is held in the fifth and sixth bytes of the display list. To make the display scroll horizontally we must add to (to scroll to the left) or subtract from (to scroll to the right) the data stored in these bytes.

The following program demonstrates horizontal scrolling; use the + key to scroll left and the * key to scroll right.

```
5      REM LEFT AND RIGHT SCROLLING
10     GRAPHICS 0
20     DIM MESSAGE$(40):MESSAGE$=
      "LEFT AND RIGHT SCROLLING"
30     DISPLIST = PEEK(560)+256*
      PEEK(561)
40     DISPMEM = PEEK(DISPLIST+4)+
      256*PEEK(DISPLIST+5)
50     POSITION 0,11:PRINT MESSAGE$
60     OPEN #1,4,0,"K:"
```

```
70     GET #1,X
80     IF X=43 THEN DISPMEM=DISPMEM+
        1:GOSUB 100
90     IF X=42 THEN DISPMEM=DISPMEM-
        1:GOSUB 100
95     GOTO 70
100    DISPHI = INT(DISPMEM/256)
110    DISPLO = DISPMEM-(DISPHI*256)
120    POKE DISPLIST+4,DISPLO
130    POKE DISPLIST+5,DISPHI
140    RETURN
```

This program makes the display scroll because it alters the address from which the ANTIC chip reads display data, but it does *not* alter the addresses into which the BASIC routines write the picture data. In normal use, the pointer to display memory is set by the operating system to the address at which BASIC begins storing display data when the graphics mode is set, or the machine is switched on.

When you run this program, you will notice that as you scroll the display to the right, a series of graphics and other characters appear at the top of the screen. This is the lower end of the display-list, and appears because to scroll to the right we subtracted one from the display memory pointers. This tells ANTIC to take display data from the address one lower than that where display memory starts, and as the display list is always stored immediately before display memory, it is treated as screen data and displayed.

Vertical scrolling can be achieved in much the same way, but the display memory pointers must be incremented by 40, as the following program demonstrates.

5 REM UP AND DOWN SCROLLING

```
10  GRAPHICS 0
20  DIM MESSAGE$(40): MESSAGE$=
    "WHAT GOES UP MUST COME DOWN!"
30  DISPLIST = PEEK(560)+256*
    PEEK(561)
40  DISPMEM = PEEK(DISPLIST+4)+256
    * PEEK(DISPLIST+5)
45  DISPMAX = DISPMEM+880:DISPMIN
    =DISPMEM
50  POSITION 0,23:PRINT MESSAGE$
55  REM SCROLL UP
60  DISPMEM = DISPMEM+40
65  IF DISPMEM > DISPMAX THEN 80
70  GOSUB 100:GOTO 60
75  REM SCROLL DOWN
80  DISPMEM = DISPMEM-40
85  IF DISPMEM < DISPMIN THEN 60
90  GOSUB 100:GOTO 80
100  DISPHI = INT(DISPMEM/256)
110  DISPLO = DISPMEM-(DISPHI*256)
120  POKE DISPLIST+4,DISPLO
130  POKE DISPLIST+5,DISPHI
140  RETURN
```

This technique is simple to use but has the disadvantage that the scrolling is 'coarse', that is the motion is not smooth because the display is moved by a whole character height or width of eight pixels at a time. With a bit more programming, we can cause the display to scroll by a single pixel at a time within a single character space. A combination of both techniques will allow smooth scrolling over the entire screen.

SMOOTH SCROLLING

Smooth scrolling is achieved with the use of two GTIA registers:

VERTICAL SCROLL

HORIZONTAL SCROLL

To cause an area of the screen to scroll smoothly, we must modify the display list for the area of the screen we are interested in. To do this, a modifier instruction must be added to the display list opcode for each display block which is to be smooth scrolled.

The modifiers are:

HORIZONTAL SCROLL add 16

VERTICAL SCROLL add 32

The modifier indicates to ANTIC that a display block is to be treated differently from normal. When such a modifier is encountered, ANTIC checks the appropriate scroll register and uses the number it finds to determine how many (if any) scan lines of that display block to display.

An example will make this clear.

Suppose we wanted to vertically smooth scroll just one line of a mode 0 display - the following steps are required:

Add 32 to the display list opcode for that display block, to indicate to ANTIC that this block is to be vertically scrolled.

Write a number between 0 and 7, corresponding to the number of scan lines of that block to be displayed initially, into the vertical scroll register.

A value of 7 would display the entire block whereas 0 would result in none of the block being displayed.

Increment or decrement the scroll register to smooth scroll that display block up or down.

The following program illustrates this technique by scrolling two lines of text in the middle of the screen.

```
5      REM SMOOTH SCROLLING
10     VERTSCROLL=54277
20     DISPLIST=PEEK(560)+256*
      PEEK(561)
30     GRAPHICS 0
40     FOR Z=1 TO 10
50     PRINT "NON-SCROLLING LINE ";Z
60     NEXT Z
70     PRINT "HIDDEN LINE 11"
80     PRINT "S-M-O-O-T-H S-C-R-O-L-
      L-I-N-G-!"
90     PRINT "HIDDEN LINE 13"
100    FOR Z=14 TO 23
110    PRINT "NON-SCROLLING LINE ";Z
120    NEXT Z
200    POKE DISPLIST+15,2+32:POKE
      DISPLIST+16,2+32
300    FOR Y=0 TO 7
310    POKE VERTSCROLL,Y
320    FOR DELAY=0 TO 100:NEXT DELAY
330    NEXT Y
400    FOR Y=7 TO 0 STEP-1
410    POKE VERTSCROLL,Y
420    FOR DELAY=0 TO 100:NEXT DELAY
430    NEXT Y
440    GOTO 300
```

The program works like this:

Lines 10 to 120 create a display to illustrate the scrolling mechanism.

Line 200 adds a modifier of 32 to the display list codes for lines 11 and 12 of the graphics 0 screen to indicate vertical scrolling. (The code for a graphics 0 display block is 2, as described earlier). Line 13, which has no modifier, is initially hidden to allow compensation for the disappearing scan lines of line 11 when scrolling starts. This prevents the whole of the rest of the screen from moving up with line 11.

Lines 300 to 330 and 400 to 440 increment and decrement the vertical scrolling register, with a delay, to gradually increase or decrease the vertical offset of lines 12 and 13 on the screen and produce the effect of smooth scrolling.

This technique can be applied to any area of the screen, and also works for modes 1 and 2.

GRAPHICS MODE '0.5'

It is possible to modify the entire display list to create a display mode which the BASIC command **GRAPHICS** cannot give. If you look at the table of ANTIC instruction codes in Appendix 9, you'll notice that code 3 isn't used. This is used to select another text mode which is very similar to mode 0, but is composed of display blocks which are 10 scan lines tall. The only way to use this mode is to modify the display list, and the following program will do just that:

```
5      REM MODE 0.5
10     DL=PEEK(560)+256*PEEK(561)
20     POKE DL+3,67
30     FOR Z=6 TO 23:POKE DL+Z,3
40     NEXT Z
50     FOR Z=24 TO 26:POKE DL+Z,
      PEEK(DL+Z+5)
```

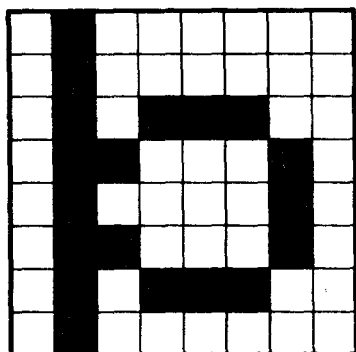

60 NEXT Z

List the program, then try running it. You will see the screen 'stretch', leaving more space between the lines of the listing than usual. This is because each display block is now 10 scan lines high instead of the normal 8. The number of lines on the screen is reduced to 19.

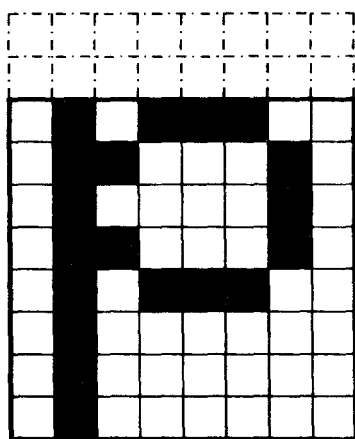
Now try typing some lower case characters; you will observe that some of them appear slightly different from normal - they have the pixel normally in the top row of the character displayed on the bottom. This is because in this mode, ANTIC takes character data from memory in a slightly different order from usual, but only for characters with ASCII codes between 96 and 128. You may be wondering what use there is for this - it allows you to create a more legible character set, in which characters like g, y and j can have descenders (the part of the character that normally appears below the line of the text). This involves reprogramming the character set as described in Chapter 12. If you want to try this you should be aware that the data in the top two rows of the character grid is displayed in the bottom two rows on the screen, so to create a letter p with a descender, your character grid should look like this:

ANTIC will rearrange the data so that when displayed, the character appears like this:
and will insert two blank scan lines before the character is displayed.

You will need to use the previous program as a subroutine to set up the display list in any program you write to use this mode.



*The Character **p** as Stored in Memory*



2 'EXTRA' SCAN
LINES

*The Character **p** as Displayed in Mode 0.5*

MIXING MODES

With more programming, it is possible to mix several graphics modes in the screen at once. You have seen examples of such mixed modes when a mode 0 text window is displayed at the foot of a graphics display, and also in the displays used in self-test mode.

As an introduction, let's examine the display list of a mode 2 screen, using the following program.

```

10    DIM LST(25)
20    GRAPHICS 2
30    DL=PEEK(560)+256*PEEK(561)
40    FOR Z=0 TO 23
50    LST(Z)=PEEK(DL+Z)
60    NEXT Z
70    GRAPHICS 0
80    FOR Z=0 TO 23
90    PRINT LST(Z); "  ";:NEXT Z

```

The display instructions are stored in the array LST(), and are displayed on a mode 0 screen. This is what you should see (the data is shown here in a list for clarity - not as it appears on the screen):

```

112    8 blank scan lines
112    8 blank scan lines
112    8 blank scan lines

71     7 + 64 - 1 mode 2 display block
112    Display memory at
62     112 + 62*256

7      Mode 2 display block
7      Mode 2 display block
7      Mode 2 display block
7      Mode 2 display block
7      Mode 2 display block
7      Mode 2 display block
7      Mode 2 display block
7      Mode 2 display block
7      Mode 2 display block
7      Mode 2 display block

66     2 + 64 - 1 mode 0 display block
96     Display memory at
63     96 + 63*256

2      Mode 0 display block
2      Mode 0 display block
2      Mode 0 display block

```

```
65      Jump back to
88      88 + 62*256
62      at Vertical Blank
```

This display list is more complex than that examined previously, but you should be able to see that it specifies 10 mode 2 display blocks, followed by four mode 0 display blocks. The thing to note is that there are two different pointers to display memory. At the start of the list, ANTIC is told that display memory is at $112 + 62 \times 256 = 15984$ (for a 16k machine), and this is where data for the mode 2 display is stored. After completing the 10 mode 2 display blocks, ANTIC is directed elsewhere in memory, to $96 + 63 \times 256 = 16224$ (for a 16k machine), where it finds the data for the mode 0 display.

You can apply the same principle to creating your own mixed mode displays.

As an example, the following program alters the display list of a mode 0 screen to cause any data appearing on the lower half of the screen to appear in multicolour character mode (mode 12).

```
10      GRAPHICS 0
15      SETCOLOR2,3,14:SETCOLOR4,3,14
20      DL=PEEK(560)+256*PEEK(561)
30      FOR Z=18 TO 29
40      POKE DL+Z,4:NEXT Z
```

Line 10 sets up the mode 0 display list and the contents of colour registers 2 and 4 are set equal so that the background of both displays appears the same.

If you run the program, it will turn the entire screen pink. If you now list the program all will

This is a fairly straightforward and not particularly useful demonstration of mixed mode displays. A better but more complex one is to create a mode 8 graphics display, and label it in a combination of text modes. The following program creates a display comprising a picture in mode 8, with titles in modes 2 and 0.

```

5      REM MIXED MODES DEMO
10     DIM T1$(20),T2$(20),T3$(40),
      A$(40)
15     GRAPHICS 8+16
20     ST=PEEK(560)+256*PEEK(561)
30     DM=PEEK(ST+4)+256*PEEK(ST+5)
40     SETCOLOR 4,3,12
50     SETCOLOR 1,0,2
60     SETCOLOR 2,3,12
99     REM ALTER DISPLAY LIST
100    POKE ST+3,71
110    POKE ST+6,7:POKE ST+7,2
120    POKE ST+160,PEEK(ST+199)
130    POKE ST+161,PEEK(ST+200)
140    POKE ST+162,PEEK(ST+201)
199    REM SET UP TITLES
200    T1$="          THE CUBE          "
210    T2$="          mmmmmmmmmmm      ":
      REM m IN REVERSE VIDEO
215    T3$="          ":
      REM 20 SPACES
220    A$=T1$:GOSUB 500:MEM=MEM+20
230    A$=T2$:GOSUB 500:MEM=MEM+20
240    A$=T3$:GOSUB 500
299    REM DRAW CUBE IN MODE 8
300    COLOR 1
310    PLOT 100,50:DRAWTO 150,50:
      DRAWTO 150,100:DRAWTO 100,100:
      DRAWTO 100,50

```

```
320  DRAWTO 125,25:DRAWTO 175,25:
      DRAWTO 150,50
330  PLOT 150,100:DRAWTO 175,75:
      DRAWTO 175,25
340  GOTO 340
499  REM TRANSLATE TITLES
500  FOR Z=1 TO LEN(A$)
510  IF A$(Z,Z)=" " THEN A$(Z,Z)=
      CHR$(0)
520  IF ASC(A$(Z,Z))<>0 THEN
      A$(Z,Z)=CHR$(ASC(A$(Z,Z))-32)
530  NEXT Z
549  REM POKE INTO DISPLAY MEMORY
550  FOR Z=1 TO LEN(A$)
560  POKE DM+MEM+(Z-1),ASC(A$(Z,Z))
570  NEXT Z
580  RETURN
```

The program will create a display comprising:

2 lines of mode 2 text +

1 line of mode 0 text +

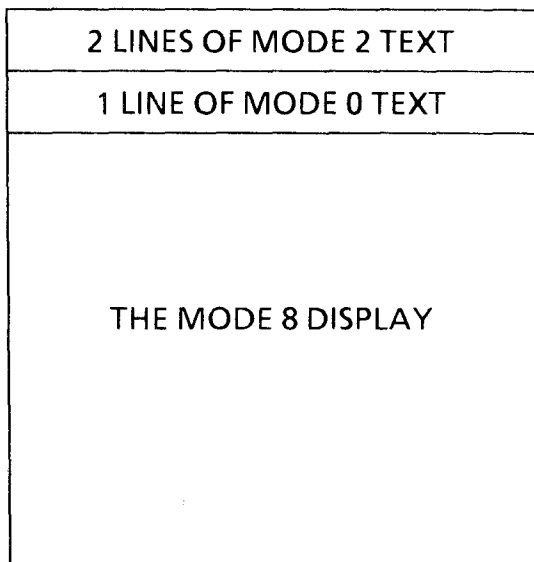
many lines of mode 8 graphics

looking like this:

How The Program Works

Lines 10 to 60 dimension arrays for the titles, select mode 8 with no text window, and find the start of the display list (ST) and display memory (DM). The colour registers for background, border and foreground are also set up.

Lines 100 to 140 alter the display list as follows:



The Mixed Mode Display

- 100 Creates a mode 2 display block. The code 71 is actually $64+7$ - this is the first display block in the list, and so contains the modifier pointing to the display memory address which is 64).
- 110 Creates the second mode 2 display block, followed by a mode 0 display block.
- 120-140 Look to the end of the display list, where the 'jump at vertical blank' instruction which sends ANTIC back to the start of the display list is found, and move this instruction and the following 16 byte address further up the display list. This reduces the effective length of the display list by 37 scan lines. The reason for this is that in adding the three text display blocks to the top of the list, we added an extra 37 scan lines, because

the first three instructions in the list would normally have specified 3 mode 8 display blocks, which are only one scan line high. To ensure that no more than 192 scan lines are specified by the display list, we must 'chop off' the extra scan lines now at the bottom of the list. If you don't take this precaution, some strange effects will result (something for you to try!).

Lines 200 to 240 set up the characters for the titles, which are converted to internal code and **POKEd** into display memory by subroutine 500. The characters are put into display memory starting at the top, and the variable **MEM** keeps track of the position in memory of each string of characters.

Finally, the picture is drawn in the usual way by lines 300 to 340.

You can use the principles covered here to create your own mixed mode displays, by observing the following rules.:

- 1 Start with the display list of the mode which uses the most memory (in our case this was mode 8). See Appendix 5 for memory requirements of the various modes.
- 2 Change the codes to create text for labelling in the appropriate part of the screen.
- 3 Calculate the number of scan lines required for the new display, and if greater than 192 transfer the jump from the end of the display list to a point such that the display requires no more than 192 scan lines.
- 4 Calculate the memory requirements for each text mode display block, and **POKE** the

internal codes for the characters into display memory.

5 Draw the mode 8 picture in the normal way.

These rules and some patience are all that's needed to create a very complex mixed mode display.

CHAPTER 15

PLAYER-MISSILE GRAPHICS

The 16 graphics modes of the 130XE allow you to create superb displays, but there is one thing missing; it is very difficult to produce animated displays using the normal graphics facilities. The Atari has a second kind of graphics feature to overcome this problem: *player-missile* graphics.

Consider the problems of producing an animated display, even if you have only one object moving over a fixed background. As well as redrawing the moving object every time its position changes, you must redraw the background, or the moving object will leave a trail of damage in its wake. Atari have solved this problem by providing movable graphics objects – called *players* and *missiles*, which can easily be moved across the screen without any risk of upsetting the background image.

Players and missiles are similar in nature. A player can extend over the full height of the screen and be up to eight pixels wide; a missile is the same height but only two pixels wide. Four players and four missiles are available, or the missiles may be combined and used as a fifth player. The players and missiles are completely independent of the background display, and can be used in conjunction with displays in any of the 130XE's graphics modes.

A further feature, which is useful in many applications, is that the Atari hardware (the GTIA chip) automatically detects the 'collisions' between pairs of players, players and missiles, and players or missiles and background features when they are

overlapping each other on the display. You can in addition control the relative priorities of the players, missiles and the background to control which will show when two objects overlap, and thus obtain three-dimensional effects.

This simple program shows one player, which appears as a vertical band on the mode 0 screen.

```
10      REM SET ADDRESSES FOR PLAYER 0
20      P0GRAPH=53261
30      P0HPOS=53248
40      P0COL=704
50      P0SIZE=53256
90      REM SHOW PLAYER
100     POKE P0GRAPH, 255
110     POKE P0COL, 54
120     POKE P0SIZE, 3
130     POKE P0HPOS, 120
```

The player (Player 0) is set to a vertical red band by this program. The properties of the player are controlled by the contents of the four registers. The pattern of bits in location 53261 controls the appearance of the player - the program sets it to 255 to give a solid vertical band (255 is 11111111 in binary). Try:

```
POKE 53261, 170
```

This gives four thin stripes down the screen (170 is 10101010 in binary - only the positions corresponding to the 1s are shown).

The colour is controlled by location 704, which works in the same way as the colour registers for normal graphics. The number in the register indicates the colour and luminance as follows:

Number = (Colour No.*16) + luminance
Gold is colour 1, so for a bright gold player, try

```
POKE 704, 16*1 + 12
```

The value in location 53256 controls the width of the player. A value of 0 gives normal width, 1 gives double width, and 3 quadruples the player's width. To shrink the player a little, try:

```
POKE 53256, 1
```

The fourth control register has the most dramatic effect. Location 53248 controls the horizontal position of the player on the screen. The value may range from 0 to 255, and the corresponding player positions range beyond the edges of the screen on both sides, allowing players to be 'hidden'. Try adding these lines to the first example program:

```
130   FOR X=0 TO 255
140   POKE POHPOS, X
150   NEXT X
160   GOTO 130
```

The player will move across the screen from left to right as the value in the horizontal position register is increased.

CONTROLLING THE SHAPES OF PLAYERS

The four players may all be controlled in the same way as Player 0. The registers to be used for the players are shown in Table 15.1. These registers all control the GTIA (the special purpose chip which controls the display).

So far you have seen nothing very impressive. Moving bands, yes, but these are hardly the stuff of

REGISTER	EFFECT	LOCATION
PM0COL	Player/Missile 0 colour	704
PM1COL	Player/Missile 1 colour	705
PM2COL	Player/Missile 2 colour	706
PM3COL	Player/Missile 3 colour	707
P0HPOS	Player 0 position	53248
P1HPOS	Player 1 position	53249
P2HPOS	Player 2 position	53250
P3HPOS	Player 3 position	53251
M0HPOS	Missile 0 position	53252
M1HPOS	Missile 1 position	53253
M2HPOS	Missile 2 position	53254
M3HPOS	Missile 3 position	53255
P0SIZE	Player 0 size	53256
P1SIZE	Player 1 size	53257
P2SIZE	Player 2 size	53258
P3SIZE	Player 3 size	53259
MSIZE	Size of all missiles	53260
P0GRAPH	Player 0 graphics	53261
P1GRAPH	Player 1 graphics	53262
P2GRAPH	Player 2 graphics	53263
P3GRAPH	Player 3 graphics	53264
MGRAPH	Graphics for all missiles	53265

Player-Missile Control Registers

which exciting video games are made. To find out

how to turn players into spaceships, we must study the GTIA in a bit more detail.

As you have seen, the data in the GRAPH register controls the appearance of the player. As the GTIA builds up the video pictures, scanning across the screen from left to right, it is constantly checking the HPOS registers to see if there are players to be switched on. When the GTIA finds it has reached the specified horizontal position for a player, the data in the appropriate GRAPH register is used to generate the next part of the horizontal line instead of the normal data in the screen memory. This process is repeated as each line of the display is scanned: as the GTIA reaches a player's HPOS setting, it takes data from the GRAPH register for that player to generate the next part of the line.

Because the display is built up a line at a time, if we could change the contents of the GRAPH register between lines, we could create much more interesting shapes for the players.

It would be impossible to modify the registers quickly enough in BASIC, but it can be done. The Atari has another special chip, the ANTIC, which is dedicated to feeding the GTIA registers with the right information at the right time. All we have to do is set up a table in memory of the data for each line of a player, and tell the ANTIC chip to use this data to feed the GRAPH register of the GTIA. The next program uses this technique to turn Player 0 into a sunflower.

```
10      REM SUNFLOWER
20      PMBASE = 54279
30      POCOL = 704
40      POSIZE = 53256
50      POHPOS = 53248
60      SDMCTL = 559
70      GRCTL = 53277
```

```
100  START = INT(PEEK(742)/8)-1
110  FOR X=1 TO 25
120  READ F: POKE START*2048 + 1024
      + 120 + X, F
130  NEXT X
150  POKE P0COL, 28
160  POKE P0SIZE, 1
170  POKE PMBASE, START * 8
180  POKE SDMCTL, 58
190  POKE GRCTL, 2

200  FOR X=20 TO 180
210  POKE P0HPOS, X
220  FOR T=1 TO 10: NEXT T
230  NEXT X
500  DATA 30, 42, 42, 93, 93, 93,
      93, 42, 42, 30, 8, 8, 8, 8, 8,
      8, 8, 8, 8, 8, 8, 107, 42, 42,
      28
```

Three new control registers are used by this program: PMBASE (54279), SDMCTL (559) and GRCTL (53277).

PMBASE is a control register in the ANTIC chip, telling it where in memory to find the data tables for the players and missiles. SDMCTL controls the memory access of the ANTIC, and GRCTL is a GTIA register, again controlling memory access.

The GRCTL (graphics control) register is used as follows:

Bit 1 is set to 1 to use memory tables for player graphics.

Bit 0 is set to 1 to use memory tables for missile graphics.

The other bits of GRACTL have no meaning.

SDMCTL is more complicated:

Bit 5 must be set to 1 to switch ANTIC on - otherwise there will be no display at all.

Bit 4 is set to 1 to give players and missiles a vertical resolution of 1 screen line to each data line; and set to 0 to give a resolution of two screen lines to each line of data.

Bit 3 is set to 1 to enable player memory access.

Bit 2 is set to 1 to enable missile memory access.

Bits 1 and 0 control the width of the background screen. Set them to 0,0 for no display, 0,1 for a width of 32 columns in graphics mode 0, 1,0 for 40 columns (the normal setting) and 1,1 for 48 columns.

The value in PMBASE should be the high byte of the beginning of the player-missile graphics data area. The data area is laid out as shown below. There are two possible configurations of the data area; which is used depends on the resolution indicated by bit 4 of the SDMCTL register. If a resolution of 1 data line per screen line is indicated, 256 bytes are needed to store the data for each player, one byte for each screen line. If the resolution selected is 2 screen lines for each data line then only half as much memory is needed, 128 bytes for each player, and the whole table can be fitted into half the memory.

The address indicated by PMBASE must be a multiple of 2048 (1024 in reduced resolution mode). If PMBASE was an eight-bit register you would expect the value to be the high byte of an address, which could therefore be any multiple of 256. In fact, not all the bits of PMBASE are used.

In the sunflower program, the higher resolution of 1 screen line to each data line was used. PMBASE was set up by finding the high byte of the address of

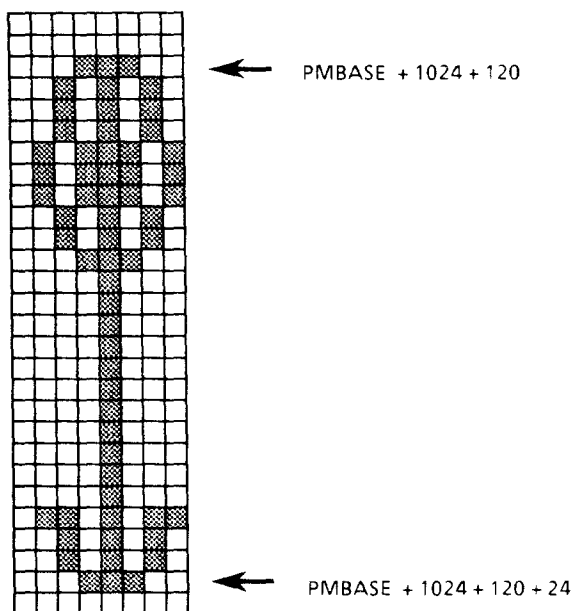
Player 3 data				PMBASE + 2048 (1024)
Player 2 data				PMBASE + 1792 (896)
Player 1 data				PMBASE + 1536 (768)
Player 0 data				PMBASE + 1280 (640)
Missile 3 data				PMBASE + 1024 (512)
Missile 2 data				
Missile 1 data				
Missile 0 data				
Not used for graphics				PMBASE + 768 (384)
				PMBASE

Player-Missile Data Area

the top of available memory, (stored in 742), finding the next lowest multiple of 2048 and subtracting a further 2048 to find a safe place for PMBASE.

Defining Players

The method of defining a player is similar to that used for defining characters (see Chapter 12). The symbol is drawn on a grid, eight pixels wide, and the values of the bits added up to give a data byte. The difference with players is that instead of using eight lines, you can use up to 256. The sunflower uses only 25 lines of the player to reduce the amount of typing needed.



The Sunflower data

Missiles

Missiles are defined in a similar manner to players, except that a missile is only two pixels wide instead of eight. This means that one register of the GTIA, the MGRAPH register, can be used to specify the graphics for all four missiles at once. The register is used like this:

- Bits 7 and 6 define the graphics for Missile 3
- Bits 5 and 4 define the graphics for Missile 2
- Bits 3 and 2 define the graphics for Missile 1
- Bits 1 and 0 define the graphics for Missile 0

Because there is only one register on the GTIA for all four missiles, only one data table is needed to specify the appearance of all four missiles.

Each missile usually takes the colour specified in the colour register for the player of the same number. The missiles use horizontal position registers in the same way as players.

Missile size is controlled by one register MSIZE (53260). The width of missile 0 is controlled by bits 0 and 1, that of missile 1 by bits 2 and 3, and so on. The values for the two bits are the same as those used in the player width registers.

It is possible to use the four missiles as a fifth player by careful use of the missile horizontal position registers. If bit 4 of the priority register GPRIOR is set to 1, the missiles will all appear the same colour - that indicated in display colour register number 3.

PRIORITIES

With a possible four players and four missiles moving around the screen, and passing over graphics displays, some rules are necessary for deciding what should be shown where two players overlap, or where a player or missile overlaps the other graphics. Another GTIA register is used to controls the relative priorities of players, missiles, and graphics of different colours. This register is called GPRIOR, and is at address 623. The four lowest bits of this register (bits 0 to 3) are used to select among four different orders of precedence for the colours. These are shown in the table overleaf.

Notice that the priorities are given in terms of colour registers. The players and missiles are not shown separately as the colour registers are shared, and it is the colour register used which determines the priority of a graphics item. Points to note are that colour register 4 is always used for the background colour of a graphics display, and

Bit 3 set	Bit 2 set	Bit 1 set	Bit 0 set
COL 0	COL 0	PCOL0	PCOL0
COL1	COL 1	PCOL1	PCOL1
PCOL 0	COL 2	COL0	PCOL2
PCOL1	COL 3/PCOL5	COL1	PCOL3
PCOL2	PCOL0	COL2	COL0
PCOL3	PCOL1	COL 3/PCOL5	COL1
COL 2	PCOL2	PCOL2	COL2
COL 3/PCOL5	PCOL3	PCOL3	COL 3/PCOL5
COL 4 (background)	COL 4 (background)	COL 4 (background)	COL 4 (background)

Player-Missile and Colour Priorities

colour register 3, which is not used in most graphics modes, may be used as the colour register for a fifth player if bit 4 of GPRIOR is set to 1.

If you try setting more than one of the four priority controlling bits of GPRIOR at the same time, you will get very strange results, as the order of precedence will not then be clearly defined.

Try running the example program below, and watch what happens as the priorities are changed. The program shows the four players: player 0 is pink, player 1 is white, player 2 is green and player 3 is light blue. The players move around each other and over graphics blocks coloured by three colour registers: colour register 0 is orange, colour register 1 is green and colour register 2 is dark blue.

```
10      REM PRIORITIES
```

```
90      REM SET UP PLAYERS
```

```
100 POKE 704, 56: POKE 705, 14:
    POKE 706, 198: POKE 707, 122
110 FOR GRAPH=53261 TO 53264: POKE
    GRAPH, 255: NEXT GRAPH
120 FOR SIZE=53256 TO 53259: POKE
    SIZE, 0: NEXT SIZE

138 REM
140 REM BEGIN DEMO
142 REM LOOP FOR EACH PRIORITY
144 REM
150 FOR PR=0 TO 3
155 GRAPHICS 5

159 REM SET PRIORITY
160 POKE 623, 2^PR
170 PRINT "PRIORITY BIT "; PR; "
    SET"

178 REM MOVE PLAYERS RIGHT
180 FOR HP=0 TO 3: POKE 53248+HP,
    60+HP*15: NEXT HP
200 FOR HP=0 TO 3
210 FOR X=60+HP*15 TO 170+HP*10
220 POKE 53248+HP, X
230 FOR D=1 TO 10: NEXT D
240 NEXT X
250 NEXT HP

290 REM DRAW RECTANGLES
300 FOR C=1 TO 3
310 COLOR C
320 X=(C-1)*20
330 Y=(C-1)*15
340 PLOTX+10, Y+10
350 DRAWTO X+10, Y
360 DRAWTO X, Y
370 POSITION X, Y+10
380 POKE765, C
390 XIO 18, #6, 0, 0, "S:"
```

```
400     NEXT C

490     REM MOVE PLAYERS LEFT
500     FOR HP=0 TO 3
510     FOR X=170+HP*10 TO 30 STEP -1
520     POKE 53248+HP, X
530     FOR D=1 TO 10: NEXT D
540     NEXT X
550     NEXT HP
600     NEXT PR: REM GO BACK FOR NEXT
          PRIORITY
```

One final point is that in the text mode, mode 0, and in text windows in other graphics modes, the text colour is controlled by colour register 2, but the luminance of the text is controlled by colour register 1. If a player passes over the text, the text colour changes to the colour of the player, but its luminance is unaltered. This means that the text may disappear if the luminances of the player and of the text are the same.

COLLISIONS

The final feature of player-missile graphics is that the GTIA chip can detect collisions between players, between a player and a missile, and between players or missiles and the graphics objects on the main display. These collisions, which occur when two graphics objects overlap, are automatically detected and recorded in various registers. The collision indication registers are shown in the table opposite. The contents of the registers have the following meanings:

Missile/Display and Player/Display collisions

Bit 0 set indicates a collision with a display item coloured by colour register 0.

Bit 1 set indicates a collision with a display item

REGISTER	MEANING	LOCATION
M0DISP	Missile 0/Display	53248
M1DISP	Missile 1/Display	53249
M2DISP	Missile 2/Display	53250
M3DISP	Missile 3/Display	53251
P0DISP	Player 0/Display	53252
P1DISP	Player 1/Display	53253
P2DISP	Player 2/Display	53254
P3DISP	Player 3/Display	53255
M0PL	Missile 0/Player	53256
M1PL	Missile 1/Player	53257
M2PL	Missile 2/Player	53258
M3PL	Missile 3/Player	53259
P0PL	Player 0/Player	53260
P1PL	Player 1/Player	53261
P2PL	Player 2/Player	53262
P3PL	Player 3/Player	53263
HITCLR	Clear collisions	53278

Collision Registers

coloured by colour register 1.

Bit 2 set indicates a collision with a display item coloured by colour register 2.

Bit 3 set indicates a collision with a display item coloured by colour register 3.

Missile/Player and Player/Player collisions

Bit 0 set indicates a collision with Player 0.

Bit 1 set indicates a collision with Player 1.

Bit 2 set indicates a collision with Player 2.
Bit 3 set indicates a collision with Player 3.
(A player cannot collide with itself.)

Note that the addresses of these registers are the same as those of some of the control registers we have already described. This does not matter, as you will only be reading from the collision registers and writing to the control registers. The Atari hardware directs the signals to two different registers depending on whether data is being read or written.

It is not possible to clear the collision detection registers by writing zeros to them. The registers are cleared by **POKE**ing to a special address called **HITCLR**, at 53278. It does not matter what value you write this location, the effect is to clear all sixteen collision registers.

The final example programs demonstrate the detection of collisions between players, and between a player and an object on the display. The first program demonstrates the detection of collisions between two players.

```
10      REM PLAYER-PLAYER COLLISIONS
20      POKE 53261, 255: POKE 704, 87
30      POKE 53262, 255: POKE 705, 55
40      FOR X=0 TO 255
50      POKE53248, X: POKE 53249, 255-
        X
60      IF PEEK(53260)=2 THEN PRINT
        "{BELL}"
70      POKE 53278, 0
80      NEXT X
```

Note: The 'BELL' character is CONTROL-2

The second program shows the detection of collisions between a player and a coloured rectangle drawn on the mode 5 display.

```
10    REM PLAYER-DISPLAY COLLISIONS
20    GRAPHICS 5
30    POKE 53261, 255: POKE 704, 87
40    COLOR 1: PLOT 30, 30:
50    DRAWTO 30, 10: DRAWTO 20, 10:
60    POSITION 20, 30: POKE 765, 1:
70    XIO 18, #6, 0, 0, "S:"
80    IF PEEK(53252)=1 THEN PRINT
      "{BELL}"
90    NEXT X
```

Player-missile graphics are one of the most powerful features of the Atari, making the creation of impressive animated displays an easy task.

CHAPTER 16

PERMANENT STORAGE

The 130XE is able to store programs on cassette tapes or floppy disks, and to LOAD and RUN programs from tape, disk or plug-in cartridge. Tapes and disks may also be used to store data for future use.

PROGRAMS ON TAPE

There are three BASIC commands you can use to load programs from the cassette unit into the 130XE, and there are three corresponding commands to allow you to save programs.

CSAVE and CLOAD

To save a program using CSAVE, type:

CSAVE and press RETURN

The TV speaker will 'beep' twice, reminding you to position the tape and press the RECORD and PLAY buttons on the cassette unit. Pressing RETURN a second time will save the program on the tape. At the end of the process (which you can monitor by turning up the volume on the TV) the tape will stop and the READY prompt will be displayed. Your program, and any variables it created when RUN, will be saved on the tape.

To load a CSAVEd program back into the 130XE, type:

CLOAD and press **RETURN**
The TV speaker will beep once, indicating that you should position the tape and press the **PLAY** key on the cassette unit. When you press **RETURN** again the tape will begin to move and the first program the 130XE finds on the tape will be loaded.

The two other groups of loading and saving commands are less commonly used.

SAVE and LOAD

This pair of commands are more general in their application, and you must specify to which device you wish to save your program. For the cassette unit the syntax is:

SAVE "C:" - the C indicating the cassette unit.

Any program saved in this way must be reloaded using the **LOAD** command, the syntax of which is:

LOAD "C:"

To confuse things slightly, it is also possible to **CLOAD** a program which has been saved with the **SAVE** command, but it is not possible to **LOAD** a **CSAVE**d program.

LIST and ENTER

As we said earlier, these commands are less commonly used but can be very useful.

LIST will save a program to the cassette in a different form from that used by **SAVE** and **CSAVE**.

Both **SAVE** and **CSAVE** save programs in *tokenised* form, that is, any **BASIC** command

within the program are saved as single character *tokens*; for example, **SETCOLOR** would be saved as a one byte number, not as the characters **S E T C O L O R**. This has the advantage of saving both time and space on the tape.

LIST however, saves the program in exactly the form it appears when you **LIST** it to the screen, character by character. The syntax of **LIST** when used to save programs is:

```
LIST "C:"
```

You can use the **LIST** command to save sections of programs on tape by specifying the line numbers of the section to be saved in the command:

```
LIST "C:" 10,100
```

will save all the program lines between 10 and 100. This means that you can keep a tape of often used subroutines on tape for incorporation into your programs as you write them.

Programs saved with **LIST** can only be loaded back into the 130XE with the **ENTER** statement, the syntax being:

```
ENTER "C:"
```

AUTO RUN PROGRAMS

There is another command, **RUN "C:"**, which allows you to automatically load and run a program in one step. **RUN "C:"** will only load and run programs which have been **CSAVED**.

You can use this facility to chain programs, that is to have one program load and execute another

from the tape. If you use this technique, bear in mind that the variables of the first program will be destroyed when the second is loaded.

PROGRAMS ON DISK

There are two types of command for loading and saving programs on disks.

LOAD and SAVE

LOAD and **SAVE** operate on programs in tokenised form, much as their counterparts used with the cassette unit. Their syntax is:

```
SAVE "D: PROG.BAS"
```

```
LOAD "D: PROG.BAS"
```

where the program is given the name **PROG** and is a **BASIC** program.

It is possible to automatically load and run a program from disk with the **RUN** command:

```
RUN "D:PROG.BAS"
```

will load and run the **BASIC** program **PROG**.

LIST and ENTER

LIST and **ENTER** deal with programs in **ASCII** form, and can be used to save and load sections of program by specifying line numbers, as with cassettes.

```
LIST "D: PROG.BAS"
```

ENTER "D: PROG.BAS"

By adding line numbers at the end of these commands, sections of program can be merged with the program already in memory.

PROGRAMS ON CARTRIDGES

Programs on cartridges are the easiest of all to use. First, you must *switch off* the 130XE, or it may be damaged. Then plug the cartridge into the socket on the top of the machine and switch on again. Follow the instructions enclosed with the cartridge to start the program.

FILE HANDLING

As well as loading and saving programs, the Atari 130XE can use the cassette unit to store and reload data generated by programs. The data is stored in *files*, and can be in numerical or string form. To create a file, the command **OPEN** is used:

OPEN #C, A, F, D

The parameter *C* is the *channel number* - a reference number used to identify a channel through which communication to a peripheral device is made.

A is the *direction* in which the data is to travel on the channel, for the cassette unit, a value of 4 indicates that data is to be read from the cassette, while a value of 8 indicates writing data to the cassette.

The F parameter indicates *file type* and for our purposes this is 0.

Finally, D specifies the *device type* where "C:" indicates the cassette unit.

Having opened a channel, data is sent to the specified device using either the **PRINT#** command or the **PUT** command.

The syntax of the **PRINT#** command is:

```
PRINT #C;"DATA"
```

```
PRINT #C;"123456"
```

The **PRINT#** command prints a string of ASCII characters to the tape, via the channel specified by the parameter C.

The **PUT** command sends a single item of data to the cassette and takes the form:

```
PUT #C,X
```

where X is the numeric data and C specifies the channel.

Reading data from the tape involves the two commands **INPUT#** and **GET**.

The **INPUT** command is used to retrieve numeric or string data, depending upon the variable given in the command. Its syntax is:

```
INPUT #C,A
```

```
INPUT #C,A$
```

GET is used to read single bytes of data from the tape, its format is:

```
GET #1,X
```

Closing Files

After use, a file must be closed, with the command

```
CLOSE #C
```

where C is the channel number.

The following short program shows the use of OPEN, PRINT#, INPUT# and CLOSE.

```
5      DIM A$(40)
10     OPEN#1, 8, 0, "C:"
20     PRINT#1, "THIS IS A TEST FILE"
30     CLOSE#1
40     STOP
100    OPEN#1, 4, 0, "C:"
110    INPUT#1, A$
120    CLOSE#1
130    PRINT A$
```

Put a blank tape into the tape unit and wind it forwards past the transparent leader tape. RUN the program, and the TV speaker will sound twice as happens when saving programs. Start the tape, and the file will be created and the data stored. The program will stop with the message "STOPPED AT LINE 40".

Rewind the tape and type GOTO 100. The TV speaker sounds once. When you set the tape going the data file will be opened and read, and the data printed on the screen.

This technique may be used to store any amount of data onto tape.

Commas and semi-colons may be used with **PRINT#** as with **PRINT**. Without punctuation, **PRINT#** puts a carriage return character after each item of data written to the tape, and these carriage returns are used by the **INPUT#** statement to distinguish between adjacent items of data. Using commas or semi-colons suppresses the carriage returns with the result that the **INPUT#** statement will not read the data properly. This is where **GET#** is useful, reading single bytes from the file without regard for carriage returns or any other markers. If you need to store single byte numbers, then writing them to the tape with **PUT#**; and reading them with **GET#** makes very efficient use of the tape, as all the space is taken up by data with no separating characters. This method is used in the Character Generator program in Chapter 12 to save character sets onto tape.

OTHER PERIPHERAL COMMANDS

STATUS, is a system variable which provides information about tape files. The variable has three possible meanings, which are shown in the table:

STATUS	MEANING
1	OK
3	CLOSE TO END OF FILE
8	ERROR CODES

CHAPTER 17

ADVANCED TECHNIQUES

The previous chapters cover all you need to write many programs for your 130XE. This chapter describes some further BASIC commands which you will find useful as you write more ambitious programs, and rounds off with some general advice on how to write good programs.

THE POP COMMAND

This strange looking command is used in connection with BASIC subroutines.

Suppose you have a program with subroutines nested one within the other, one of which checks which keys are being pressed. You may wish to respond to one of these keys - ESC perhaps - by abandoning the current routine and returning to the start of the program. This could mean **RETURN**ing through several levels of subroutine, which would be tedious to program, requiring **IF ... THEN RETURN** instructions at each level. The 130XE allows a simpler way of doing this, using the **POP** command

Each time a **GOSUB** command is executed, the 130XE stores the location of the instruction after the **GOSUB** on a *stack* in the reserved area of memory. At any time the stack contains **RETURN** addresses for all **GOSUB**s for which **RETURN** commands have not yet been performed. The **POP** command removes the most recently added **RETURN** address from the stack, so that the next **RETURN** sends the computer back to the **GOSUB**

before that most recently added. Try this example program.

```
10      GOSUB 100
20      PRINT "MAIN PROGRAM"
30      END
100     GOSUB 200
110     PRINT "SUBROUTINE 100"
120     RETURN
200     GOSUB 300
210     PRINT "SUBROUTINE 200"
220     RETURN
300     RETURN
```

If you run this program, the 130XE will follow the **GOSUBs** to line 300 and then **RETURN** progressively through each subroutine, printing the message. If you now change line 300 to read:

```
300     POP:POP:RETURN
```

and run the program again, the program will only print 'MAIN PROGRAM' and stop.

ERROR HANDLING

The 130XE has a command which allows you to take corrective action should an error occur as a program is running - **TRAP**.

The syntax is:

```
TRAP N
```

where N is the line number to which the program is directed when an error is encountered. **TRAP** must be executed before the error occurs, so you should put it at the start of any program in which you wish to use it.

In use, if an error is encountered, the normal mechanism whereby program execution is stopped and an error message displayed is suspended. Instead, program execution continues at the line specified by the **TRAP** command. At this line number you must include code to handle any possible errors. You can determine what type of error has occurred with the statement:

```
ERR = PEEK(195)
```

since location 195 contains the code number of the error (see Appendix 2). The line number at which the error occurred can be calculated with a statement like:

```
LINE = PEEK(186)+256*PEEK(187)
```

The following program uses the **TRAP** command to good effect by using it to display full error messages rather than error codes which must be looked up in a table.

```
1      GOSUB 32000:TRAP 32500
32000  DIM ERROR$(400):DIM
      MESSAGE$(20)
32020  RESTORE 32100
32030  FOR ITEM =2 TO 21
32040  MESSAGE$="
      ":REM 20 SPACES
32050  READ MESSAGE$
32060  ERROR$((ITEM-2)*20+1)=MESSAGE$
32070  NEXT ITEM
32080  RESTORE
32090  RETURN
32100  DATA MEMORY INSUFFICIENT,
      ILLEGAL VALUE,TOO MANY
      VARIABLES,STRING LENGTH,OUT OF
      DATA
```

```
32110 DATA LINE NO TOO LARGE, INPUT
      TYPE ERROR,DIMENSIONING
      ERROR,STACK OVERFLOW,
      OVER/UNDER FLOW
32120 DATALINE NOT FOUND,NEXT
      WITHOUT FOR,LINE TOO LONG,
      GOSUB/FOR DELETED,RETURN
      WITHOUT GOSUB
32130 DATA GARBAGE,INVALID STRING,
      LOAD PROG TOO LONG, DEVICE
      NUMBER,LOAD FILE ERROR
32500 LINE=PEEK(186)+PEEK(187)*256
32510 CODE=PEEK(195)
32520 PRINT"{BELL}ERROR ";CODE;" IN
      LINE ";LINE
32530 P=(CODE-2)*20+1
32540 PRINT ERROR$(P,P+19)
32550 END
```

If you save this program on tape using the LIST command it can be reloaded to form a temporary part of the program you are developing without disturbance.

MACHINE CODE SUBROUTINES

The 130XE's microprocessor does not understand BASIC programs without some assistance; it understands a much cruder set of instructions called *machine code* or *machine language*. BASIC programs can be run only because the 130XE has some machine code programs permanently stored in read-only memory (ROM) which interpret the BASIC. Writing programs in machine code is more difficult than using BASIC, but can be very worthwhile as machine code programs run very much faster. Machine code is beyond the scope of this book, but there are several books available

about the 130XE's microprocessor (the 6502) as it is very common in personal computers.

You can enter machine code routines from BASIC using the command **USR**, giving the start address in memory of the machine code routine. For example:

```
USR (69316)
```

will perform the same function as pressing the **RESET** key, and:

```
USR (69319)
```

will call the power-on setting up routines which clear the memory and print the start-up message. **BE CAREFUL** using this - it will destroy any program you have in the 130XE at the time.

The **USR** command may also contain one or more arguments, separated by commas, after the address of the machine code routine. When the command is executed, the address of the current BASIC instruction is placed on the stack, followed by each of the arguments, followed by the number of arguments included in the command. The machine code routine can then use the arguments and return them to the stack, for use by BASIC.

Machine code subroutines may be stored as string variables, and another command, **ADR**, used to determine the start address in memory of the string; hence the start address of the machine code subroutine. The syntax of **ADR** is:

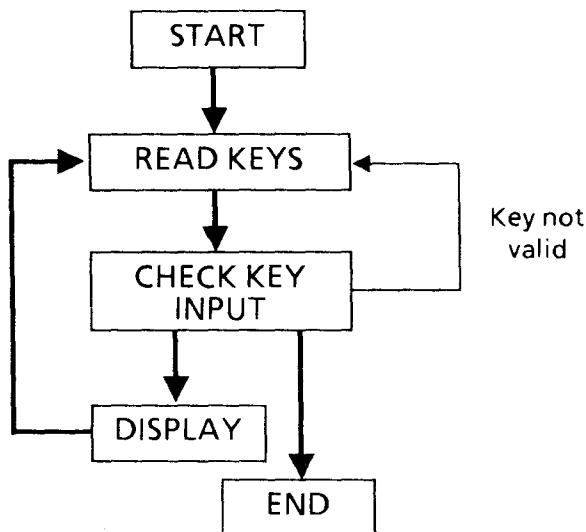
```
X = ADR(A$)
```

DESIGNING GOOD PROGRAMS

It is easy to write simple BASIC programs, and after some practice you will find it fairly easy to write quite complicated routines. However, unless you plan your programs carefully, a long program can get very messy and it can be very difficult to find all the mistakes you are bound to make.

To write complex programs successfully you must design them carefully, following a few simple rules. The phrase *structured programming* is often used to describe these rules ; all this means is that programs designed in accordance with these rules have a clear and logical structure, and the flow through the program is easy to follow. You may be put off by hearing people say that BASIC is not a suitable language for structured programming. Don't let this discourage you. While BASIC may lack the elegance of some other languages, it is still possible to use it to write good programs by following the simple guidelines set out here:

- 1 Always plan out the program on paper. Decide what you want the program to do, identifying the various tasks it will have to perform, and divide the program into sections which correspond to the tasks. For example, you might have one section of program to get data from the keyboard, another to sort the data, a third to display results, and so on.
- 2 It is often useful to draw sketches of the way the tasks fit together.
- 3 When you have got the overall structure of the program sorted out, work out in more detail



*Sketch programs before writing the
BASIC*

what each section should be doing, and how it should be done.

- 4 Write the BASIC routines for each section of the program - writing them on paper and **not** typing straight into the 130XE. Try to write each section of the program so that it is as independent as possible of the other sections, and arrange the routines with a clear sequential flow from beginning to end: avoid jumping around with **GOTO**. Use a different set of line numbers for each section, beginning each at a round number of so many thousands or tens of thousands.
- 5 Now type the program into the 130XE.
- 6 The program will not work! At this stage you can sort out all the errors easily. If you have designed the program well, with different

sections doing different tasks, it will be much easier to find the more subtle errors, as you will be able to isolate the faulty bit of the program without difficulty.

If you follow these guidelines, you will save a lot of time in getting your programs to work, and you will have more time for actually using them.

APPENDIX 1

BASIC COMMANDS

ABS (N)	Returns the absolute value of a number (removing the minus sign). <i>Chapter 8</i>
ADR (C\$)	Returns the address of a string in memory. <i>Chapter 17</i>
AND	Logical operator. Returns the value TRUE if both operands are true (1). $A = B \text{ AND } C$ A is TRUE if B and C are both TRUE. <i>Chapter 9</i>
ASC (C\$)	Function. Returns the ASCII code of a character, or of the first character of a string. <i>Chapter 7</i>
ATN (N)	Function. Gives the angle, in radians or degrees, whose arctangent is N. <i>Chapter 8</i>
BYE	Causes exit from BASIC to Self-Test mode.
CHR\$ (N)	Converts ASCII codes to characters in string form. <i>Chapter 7</i>

CLOSE (N)	Closes a channel, N, to a peripheral device (disk, printer, etc) <i>Chapter 15</i>
CLOAD	Loads a program from the cassette unit. <i>Chapter 16</i>
CLOG	Returns base 10 logarithm of a number. <i>Chapter 8</i>
CLR	Clears variables, arrays, etc, from memory, and makes the memory available to BASIC programs. <i>Chapter 16</i>
COLOR	Specifies colour register to be used in subsequent PLOT commands. In modes 0 to 2 selects character and colour register for PLOT. <i>Chapter 12</i>
COM	Identical to DIM.
CSAVE	Saves a program on the cassette unit. <i>Chapter 16</i>
CONT	Restarts program after pressing the BREAK key or executing a STOP command. <i>Chapter 5</i>

COS (N)	Gives cosine of angle, which may be given in radians or degrees. <i>Chapter 8</i>
DATA	Marks a list of string or numeric data written into a program. The items must be separated by commas. <i>Chapter 6</i>
DEG	All subsequent calculations to be performed in degrees. <i>Chapter 8</i>
DIM A(L,M) DIM A\$(L)	Dimensions arrays and strings and allocates memory space for them. <i>Chapter 4</i>
DOS	Displays DOS menu if disk in use, otherwise has same effect as BYE.
DRAWTO X,Y	Draws a line from last cursor position to specified position. <i>Chapter 13</i>
ENTER	Loads a program which has been listed to cassette or disk. <i>Chapter 16</i>
END	Ends program. Program may be restarted using CONT. <i>Chapter 5</i>
EXP (N)	Function returning exponential of N (e^N). Acts as natural antilog function. <i>Chapter 8</i>

- FOR** Begins loop. For example:
- FOR N = A TO B STEP C
- All lines as far as NEXT command are repeated with value of N increased each time from A to B in steps of C. STEP may be omitted, in which case the variable is increased by 1.
Chapter 5
- FRE (0)** Returns the amount of memory at present unused by BASIC.
Chapter 10.
- GET # (N)** Reads a single character from the specified file or device and assigns it to a variable. Device may be keyboard.
Chapter 15
- GOSUB** Program branches to a sub-routine at the specified line, returning to instruction after GOSUB when RETURN is encountered. Line number may be specified by a variable.
Chapter 5
- GOTO** Program branches to the specified line. The line number may be specified by a variable.
Chapter 5

- GRAPHICS M** Selects graphics mode M. M+16 removes text window, M+32 sets mode without clearing screen.
Chapters 12 and 13
- IF (condition)
THEN (action)** If the condition is true the action after THEN is carried out, otherwise the program continues at the next line.
Chapters 5 and 9
- INPUT N
INPUT "data";N** Prompts the operator for an input and assigns it to a variable. The variable may be a number or string: the input data must correspond.
Chapter 4
- INPUT #(N)** Retrieves data from the specified file number. Data is in the form of strings up to 80 characters in length, delimited by CHR\$(13) or , or ; or : .
Chapter 15
- INT (N)** Returns integer component of real number. For example:

 INT(3.75)

returns 3.
Chapter 8
- LEN(C\$)** Gives the number of characters in the string.
Chapter 7

LET	Optional. May be used when assigning values to variables: <pre>LET P = 5 and P = 5</pre> have the same effect.
LIST	Lists the specified lines of the program on to the screen or to a peripheral device. <i>Chapter 4</i>
LOAD	Reads program file from peripheral device. <i>Chapter 15</i>
LOCATE X,Y,Z	Returns the value of the data at screen location X,Y in variable Z. <i>Chapter 13</i>
LOG (N)	Returns the natural log - arithm of a number. <i>Chapter 8</i>
LPRINT	Outputs data to the printer.
NEW	Clears a program and its variables from memory. <i>Chapter 4</i>
NEXT N	Marks end of loop begun by FOR. The variable must be specified. <i>Chapter 5</i>
NOT	Logical operator. Reverses truth of expression. <i>Chapter 9</i>

NOTE	Finds location of next byte to be accessed from disk.
ON N GOSUB	Program branches to Nth subroutine in list. If N is larger than number of items in list no branch occurs. <i>Chapter 5</i>
ON N GOTO	Program branches to Nth destination in list. If N larger than number of items in list no branch occurs. <i>Chapter 5</i>
OPEN #N,O,0,D	Opens a channel N to a peripheral device specified by D. <i>Chapter 16</i>
OR	Logical operator. Returns value TRUE if either or both operands are true. Thus: $A = B \text{ OR } C$ A is true if B or C is true. <i>Chapter 9</i>
PEEK (LOC)	Gives the contents of memory location LOC. <i>Chapter 15</i>
PLOT X,Y	Puts point or character on screen at location X,Y. <i>Chapter 13</i>

POINT	Selects position on disk at which next byte is to be written or read.
POKE LOC,N	Puts value of N into memory location LOC. N must have value 0 - 255. <i>Chapter 15</i>
POP	Removes one address from the stack. <i>Chapter 17</i>
POSITION X,Y	Positions the cursor at X,Y for subsequent PRINT operation. <i>Chapter 12</i>
PRINT	Puts data, numbers or characters on the screen. <i>Chapter 3</i>
PRINT # N	Writes data to the specified file. <i>Chapter 15</i>
PUT # D	Outputs a single byte to specified device. <i>Chapter 16</i>
RAD	All subsequent calculations to be in radians. <i>Chapter 8</i>
READ	Copies items from DATA statements into variables.

Chapter 6

REM Allows remarks to be inserted in programs as an aid to clarity. Remarks are ignored when program is run.
Chapter 4

RESTORE Returns READ pointer to first DATA item. May be followed by a line number to indicate the DATA item required.
Chapter 6

RETURN Marks end of subroutine. Program returns to the instruction after the GOSUB instruction which called the subroutine.
Chapter 5

RND (N) Returns a random number between 0 and 1, N is a dummy variable.
Chapter 8

RUN Begins execution of BASIC program. All variables are cleared.
Chapter 4

SAVE	Stores the program currently in memory onto tape or disk. Device number specifies tape(C) or disk(D). <i>Chapter 15</i>
SETCOLOR R,C,L	Sets colour register R to colour C at luminance L. <i>Chapter 13</i>
SGN (N)	Returns 1, 0 or -1 according to whether number is positive, zero or negative. <i>Chapter 8</i>
SIN	Function. Returns the sine of a number in radians. <i>Chapter 8</i>
SQR	Returns the square root of the argument. <i>Chapter 8</i>
STATUS	Returns status of the specified device as an error code. <i>Chapter 8</i>
STICK(N)	Returns position of specified joystick. <i>Chapter 13</i>
STOP	Stops execution of program. A message 'STOPPED AT LINE xxx' is displayed. Program may be restarted using CONT. <i>Chapter 5</i>

STRIG(N)	Returns 0 if fire button of specified joystick is pressed, 1 if not. <i>Chapter 13</i>
STR\$(N)	Converts numbers to strings of numeric characters. <i>Chapter 7</i>
THEN	See IF. <i>Chapter 5</i>
TO	See FOR. <i>Chapter 5</i>
TRAP N	Causes jump to specified line number when an error occurs. <i>Chapter 17</i>
USR (ADDR,[X,Y...])	Calls machine code routine at address specified by ADDR. The optional arguments values X, Y etc. are placed on the 6502 hardware stack, together with a count of the arguments and the address of the next BASIC instruction. <i>Chapter 17</i>
VAL (C\$)	Converts string of number characters to number. <i>Chapter 7</i>
XIO T,C,p1,p2,D	Mainly used for filling blocks of colour on screen. <i>Chapter 13</i>

APPENDIX 2

BASIC ERROR MESSAGES

If the 130XE encounters a command which it is unable to execute, or a number it cannot handle, an error message will be displayed.

If the error occurred while running a program, the program will stop, and a message of the form

ERROR * AT LINE XX**

will be displayed, where *** represents the error code, and XX is the program line at which the error has occurred. The program is retained in the computer, as are the values assigned to all variables at the time of the error.

In immediate mode, the error message takes the form

ERROR - **

and is embedded in the line which is in error.

The following descriptions explain the error codes, and the possible reasons for them:

2 OUT OF MEMORY

Either the program is too large, or you have used too many variables, too many FOR ... NEXT loops,

too many GOSUBs or allocated too much memory space with a DIM command.

3 VALUE ERROR

A number outside the allowable range was encountered, or a negative value has been used as an argument for a function requiring a positive argument.

4 TOO MANY VARIABLES

More than 128 variable names have been used.

5 STRING LENGTH ERROR

A program tried to manipulate string data outside the range for which the string was dimensioned, or zero was used as a character position reference in a string handling command.

6 OUT OF DATA

There were insufficient data items for the READ command.

7 ILLEGAL LINE NUMBER

A negative line number, or one greater than 32767 has been referenced.

8 INPUT ERROR

An attempt to input a string value into a numeric variable was made.

9 ARRAY OR STRING DIMENSION

Either an undimensioned string or array has been referenced, or an attempt has been made to redimension a string or array.

10 ARGUMENT STACK OVERFLOW

An expression is too large or the program contains too many GOSUBs.

11 FLOATING POINT OVER/UNDER FLOW

A number less than $1E-99$ or greater than $1E 98$ has been encountered, or an attempt has been made to divide by zero.

12 LINE NOT FOUND

A non existent line number has been referenced.

13 NO MATCHING FOR

A NEXT statement with no corresponding FOR was encountered.

14 LINE TOO LONG

The line entered is greater than 140 characters.

15 GOSUB OR FOR DELETED

The line containing a GOSUB or FOR statement has been deleted.

16 RETURN ERROR

There is no corresponding GOSUB statement for a RETURN.

17 GARBAGE ERROR

Random error.

18 INVALID STRING CHARACTER

Either an invalid character is present in a string, or the argument of a VAL command is non-numeric.

19 LOAD PROGRAM TOO LONG

The program currently being loaded is too large for the available memory.

20 DEVICE NUMBER ERROR

A device number less than 0 or greater than 7 was used.

21 LOAD FILE ERROR

The LOAD command was used to load a program which was CSAVED or LISTed.

128 BREAK ABORT

The BREAK key was pressed during an I/O operation.

129 CHANNEL OPEN

An attempt has been made to use a channel which is already open.

130 NON - EXISTENT DEVICE

The device has not been specified in an I/O operation.

131 IOCB WRITE ONLY

An attempt has been made to read from a file opened for a write operation.

132 INVALID COMMAND

An illegal command has been used in an I/O operation.

133 DEVICE/FILE NOT OPEN

An attempt has been made to reference an un-OPENed channel.

134 ILLEGAL IOCB NUMBER

A number less than 0 or greater than 7 was used to access a channel.

135 IOCB READ ONLY ERROR

An attempt has been made to write to a device or file which was opened only for read operations.

136 END OF FILE

The end of file marker has been reached.

137 TRUNCATED RECORD

The INPUT command has been used to access a file created with the PUT command, or the record is larger than allowed.

138 DEVICE TIMEOUT

The specified device did not respond within the time allotted by the 130XE for communication between the computer and peripherals.

139 DEVICE NAK

No response from peripheral device.

140 SERIAL FRAME ERROR

Indicates faulty computer or peripheral.

141 CURSOR OUT OF RANGE

The cursor is outside the allowable range for a graphics mode.

142 SERIAL BUS RUNOUT

Fault in serial bus between computer and peripherals.

143 CHECKSUM ERROR

Data being transferred from peripheral is being corrupted.

144 DEVICE DONE ERROR

An attempt has been made to write to a write-protected disk.

145 BAD SCREEN MODE ERROR

An error in the screen handler has occurred.

146 FUNCTION NOT IMPLEMENTED

An attempt has been made to use a device in a manner which is not possible.

147 INSUFFICIENT SCREEN RAM

Not enough memory for the graphics mode selected.

160 DRIVE NUMBER ERROR

An incorrect number has been used to specify the disk drive.

161 TOO MANY OPEN FILES

The limit imposed on the number of disk files open at one time has been reached.

162 DISK FULL

All sectors of the disk are used.

163 UNRECOVERABLE SYSTEM I/O ERROR

An error is present in the DOS or on the disk.

164 FILE NUMBER MISMATCH

Incorrect use of the POINT command - the sector specified was not included in the open file.

165 FILE NAME ERROR

Illegal filename.

166 POINT DATA LENGTH ERROR

Incorrect use of the POINT command - the specified byte number did not exist within the specified sector.

167 FILE LOCKED

An attempt was made to access a locked file.

168 INVALID DEVICE COMMAND

An illegal device command was used.

169 DIRECTORY FULL

No more than 64 files can exist on one disk.

170 FILE NOT FOUND

An attempt has been made to access a file which is not on the disk.

171 POINT INVALID

Incorrect use of POINT command - an attempt has been made to use the POINT command with an incorrectly opened file.

172 ILLEGAL APPEND

The wrong DOS has been used to append a file.

173 BAD FORMAT

Bad sectors have been found on the disk while formatting - discard the disk.

APPENDIX 3

SPEEDING UP PROGRAMS

There are several things you can do to increase the running speed of your BASIC programs. Unfortunately, this is usually at the expense of clarity, and you may find it useful to keep a 'slow', but easy-to-follow version of your program should you wish to amend it at a later date.

- 1 Remove all unnecessary spaces and REMs from the program. A small speed increase will result, because BASIC will not have to skip over redundant spaces to find executable commands.
- 2 Always use variables instead of constants. The 130XE can handle variables much more rapidly than numbers. This is especially important in FOR ... NEXT loops.
- 3 Use as many statements per line separated by ':' as possible.
- 4 Re-use the same variables whenever possible.
- 5 Use the zero elements of arrays when possible.
- 6 Assign often used variables early on in the program. The 130XE stores all variables in a table. The first declared variables are the first in the table and are found more quickly.

- 7 Put all subroutines near the start of the program. The computer searches through the whole program for a subroutine each time a **GOSUB** command is executed, and subroutines having low line numbers will be found and executed more quickly than those at the end of the program.
- 8 Programs involving large amounts of calculation but not requiring the screen display may be speeded up by turning off the screen, with a line such as :

```
100 A=PEEK(559):POKE 559,0
```

The screen can be turned back on to display the results of the calculations with a statement:

```
200 POKE 559,A
```

APPENDIX 4

NUMBERING SYSTEMS

Computers store and operate upon numbers in a different way from humans – they use a numbering system known as *binary notation*.

Binary notation is a means of representing quantities with groups of 1s and 0s. We are more used to a system called decimal notation, in which quantities can be represented by combinations of up to ten symbols (the numbers 0 to 9).

Computers use the binary system because they are able to recognise and differentiate between only two states – ON and OFF. These two states can conveniently be represented by 1 (ON) and 0 (OFF).

A single 1 or 0 is called a **BI**nary digiT, or **BIT**, and computers store data in the form of groups of eight of these bits, known as *BYTES*.

In the computer memory, one memory location is able to store one Byte of data (eight bits). A collection of 1024 of these bytes is called a *KILOBYTE*, or k for short. We can get an idea of the data storage capacity of a computer from the number of k of memory it has (48k, for example, is $48 * 1024 * 8$ bits).

We have described a byte as a collection of eight bits, like this :

11111111

This is an 8-bit binary number, which represents 255 in decimal notation. To see how this is so, we must first examine the decimal number and see what it means.

H	T	U
2	5	5

means:

$$2 \times 100 + 5 \times 10 + 5 \times 1$$

In other words, each digit is worth 10 times the one to its right.

Binary notation uses this same 'place value' principle, except each bit in a binary number is worth *double* that to its right. We can assign values to the eight bits in the same way as the 'hundreds, tens and units' assigned to the digits of a decimal number.

128	64	32	16	8	4	2	1
1	1	1	1	1	1	1	1

By adding up, we can see why this number represents 255:

1	*	128
1	*	64
1	*	32
1	*	16
1	*	8
1	*	4
1	*	2
+1	*	1
<hr/>		
		255

You'll notice that 255 is the biggest number we can represent with an 8-bit binary number. Hence this

is the largest number we can store in a single memory location.

As a further example, let's take the decimal number 170. To find its binary representation, we continuously divide by two, and the remainder becomes a bit in the binary number.

170/2	=	85	remainder	0
85/2	=	42	"	1
42/2	=	21	"	0
21/2	=	10	"	1
10/2	=	5	"	0
5/2	=	2	"	1
2/2	=	1	"	0
1/2	=	0	"	1

giving us the binary number

10101010 (reading upwards)

ADDITION OF BINARY NUMBERS

Binary numbers can be added together in the same way as decimal numbers. An example will make this clear.

To perform the sum:

$$\begin{array}{r} 105 \\ + 19 \\ \hline 124 \end{array}$$

we add up the digits in each column to form that digit of the answer. If the result of this addition is greater than 9, we generate a carry into the next column. This principle also applies to binary numbers. Let's perform the same calculation with the binary forms of 105 and 19:

$$\begin{array}{r}
 01101001 \\
 +00010011 \\
 \hline
 01111100
 \end{array}$$

In the case of binary addition we generate a carry when adding 1 to 1 (in columns 1 and 2 in our example).

NEGATIVE BINARY NUMBERS

You might have wondered how the computer can recognise negative numbers, since it can only tell the difference between on and off. This is achieved by a method known as 'two's complement' notation, which uses one bit of the binary number (the most significant bit, often labelled bit 7) as a sign bit.

To subtract two numbers in binary, we form the two's complement of the number to be subtracted, then add it to the other number.

As an example we'll subtract 50 from 100 :

$$\begin{array}{r}
 100 \qquad \qquad 01100100 \\
 -50 \qquad \qquad 00110010 \\
 \hline
 50 \qquad \qquad 00110010
 \end{array}$$

(the answer we want).

To perform the sum in binary, first we find the two's complement of 50, by changing all the 0s to 1s and all the 1s to 0s, then adding 1.

$$\begin{array}{r}
 50 \text{ in binary is:} \qquad \qquad 00110010 \\
 \text{Change 1s to 0s:} \qquad \qquad 11001101 \\
 \text{Adding 1 gives} \\
 \text{its two's complement:} \qquad \underline{11001110}
 \end{array}$$

Now add this to the binary for 100:

$$\begin{array}{r}
 01100100 \\
 11001110 \\
 \hline
 1 \quad 00110010
 \end{array}$$

The result is binary 50 – the method worked.

Notice that a carry was generated, indicating that the answer is positive. A consequence of using bit 7 as a sign bit is that the range of numbers we can represent with an 8-bit number is restricted to

-128 to +127

HEXADECIMAL

Manipulating numbers in binary is a lot easier for a computer than it is for a human, and one way in which binary numbers can be made more digestible is by representing them in hexadecimal notation, or *hex*.

Hex is a system of counting in base 16, using the symbols 0 to 9 and A to F as follows

DECIMAL 0 1...9 10 11...15 16 17

HEX 0 1...9 A B... F 10 11

Thus FF (hex) represents 255 (decimal) and 11111111(binary).

You will frequently encounter references to hex, usually as memory addresses, because it is so convenient. (Which of these is easiest to recognise? 1111111111111111 or 65535 or #FFFF).

APPENDIX 5

GRAPHICS MODES

MODE	TYPE	COLS.	ROWS SPLIT	ROWS FULL	COLOURS	MEM REQ. SPLIT	MEM REQ FULL
0	TEXT	40	-	24	1	-	992
1	TEXT	20	20	24	5	674	672
2	TEXT	20	10	12	5	424	420
3	GR	40	20	24	4	434	432
4	GR	80	40	48	2	694	696
5	GR	80	40	48	4	1174	1176
6	GR	160	80	96	2	2174	2184
7	GR	160	80	96	4	4190	4200
8	GR	320	160	192	1	8112	8138
9	GR	80	-	192	1	-	8138
10	GR	80	-	192	9	-	8138
11	GR	80	-	192	16	-	8138
12	TEXT	40	20	24	5	1154	1152
13	TEXT	40	10	12	5	664	660
14	GR	160	160	192	2	4270	4296
15	GR	160	160	192	4	8112	8138




























APPENDIX 6

COLOUR CODES

COLOUR	CODE
GREY	0
GOLD	1
ORANGE	2
RED/ORANGE	3
PINK	4
PURPLE	5
BLUE/PURPLE	6
BLUE	7
BLUE	8
LIGHT BLUE	9
TURQUOISE	10
GREEN/BLUE	11
GREEN	12
YELLOW/GREEN	13
ORANGE/GREEN	14
LIGHT ORANGE	15

APPENDIX 7

CHARACTER CODES

SET 1	SET 2	0	1	2	3
SPACE		32	0	160	128
!		33	1	161	129
"		34	2	162	130
#		35	3	163	131
\$		36	4	164	132
%		37	5	165	133
&		38	6	166	134
'		39	7	167	135
(	40	8	168	136
)		41	9	169	137
*		42	10	170	138
+		43	11	171	139
,		44	12	172	140
-		45	13	173	141
.		46	14	174	142
/		47	15	175	143
0		48	16	176	144
1		49	17	177	145
2		50	18	178	146
3		51	19	179	147
4		52	20	180	148
5		53	21	181	149
6		54	22	182	150
7		55	23	183	151
8		56	24	184	152
9		57	25	185	153
:		58	26	186	154
£	£	59	27	187	155
<	↑	60	28	188	156
=	↓	61	29	189	157

SET 1	SET 2	0	1	2	3
>	←	62	30	190	158
?	→	63	31	191	159
@	◆	64	96	192	224
A	a	65	97	193	225
B	b	66	98	194	226
C	c	67	99	195	227
D	d	68	100	196	228
E	e	69	101	197	229
F	f	70	102	198	230
G	g	71	103	199	231
H	h	72	104	200	232
I	i	73	105	201	233
J	j	74	106	202	234
K	k	75	107	203	235
L	l	76	108	204	236
M	m	77	109	205	237
N	n	78	110	206	238
O	o	79	111	207	239
P	p	80	112	208	240
Q	q	81	113	209	241
R	r	82	114	210	242
S	s	83	115	211	243
T	t	84	116	212	244
U	u	85	117	213	245
V	v	86	118	214	246
W	w	87	119	215	247
X	x	88	120	216	248
Y	y	89	121	217	249
Z	z	90	122	218	250
[♠	91	123	219	251
\		92	124	220	252
]	↖	93	125	221	253
^	◀	94	126	222	254
	▶	95	127	223	255

APPENDIX 8

DISPLAY LIST INSTRUCTIONS

In order to create borders, ANTIC has a set of instructions which create up to eight scan lines in either screen or border colour:

ANTIC CODE	NUMBER OF BLANK LINES
0	1
16	2
32	3
48	4
64	5
80	6
96	7
112	8

Another group of instructions select graphics modes:

ANTIC CODE	BASIC MODE
2	0
6	1
7	2
8	3
9	4
10	5
11	6

13	7
15	8
15	9
15	10
15	11
4	12
5	13
12	14
14	15
3	-

Antic can be instructed to jump, in which case it treats the next two bytes in the display list as the address to which it should jump:

JUMP TO ADDRESS	1
JUMP AT VERT BLANK	65

Most instructions can be modified as follows:

add 16	HORIZONTAL SCROLL
add 32	VERTICAL SCROLL
add 64	LOAD MEMORY SCAN (REFRESHES SCREEN MEMORY POINTERS)

APPENDIX 9

USEFUL MEMORY LOCATIONS

LOCATION	DESCRIPTION
16	POKE with 64 and POKE 53774 with 64 to disable BREAK key.
18, 19, 20	Real time clock.
65	Poke with 0 to hear sound from cassette during loading. Poke with 1 to stop sound.
82	Left margin of mode 0 text display.
83	Right margin of mode 0 display.
84	Current cursor row.
85, 86	Current cursor column.
87	Current display mode.
88, 89	Start of screen memory.
93	Contains a copy of the character underneath the cursor.
94, 95	Current cursor address.
128, 129	Start of BASIC memory.
656	Cursor row in text window.
657, 658	Cursor column in text window.
144, 145	Top of BASIC memory.
186, 187	Line number of error found by TRAP .
195	Error number of error found by TRAP .
212, 213	Value returned by USR function.
251	0 = Radians mode. 6 = Degrees mode.
559	POKE with 0 to disable display. See Chapters 14, 15.
694	0 = Keyboard gives normal characters. Non-zero = Keyboard gives inverse video characters.

702	0 = No shift locks. 64 = CAPS lock on. 128 = CONTROL lock on.
703	24 = Text window off. 4 = Text window on.
704-707	Player-Missile colour registers 0 to 3.
708-712	Graphics colour registers 0 to 4.
741, 742	Top of free RAM.
743, 744	Start of free RAM.
752	0 = Cursor on. 1 = Cursor off.
755	Controls appearance of cursor and characters. Setting bit 0 makes the cursor opaque. Setting bit 1 makes the cursor visible. Setting bit 3 inverts the displayed characters.
756	224 = Standard character set. 226 = Alternate character set.
763	ASCII code of last key pressed.
764	Internal code of last key pressed.
765	Holds colour data for XIO command.
53279	Bit 0 is zero if START is pressed. Bit 1 is zero if SELECT is pressed. Bit 2 is zero if OPTION is pressed. POKE 53279, 8 retracts loudspeaker cone. POKE 53279, 0 extends loudspeaker cone.
54018	POKE with 52 to start cassette motor. POKE with 60 to stop cassette.
58484	X = USR(58484) gives RESET.
58487	X = USR(58487) gives switch-on reset.

APPENDIX 10

MUSIC FREQUENCIES

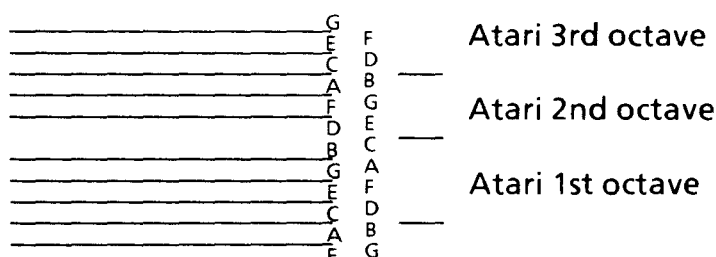
NOTE	PERIOD	NOTE	PERIOD
C	255	F#,Gb	89
C#,Db	239	G	81
D	227	G#,Ab	76
D#,Eb	211	A	72
E	203	A#,Bb	68
F	191	B	64
F#,Gb	179	C	63
G	171	C#,Db	57
G#,Ab	159	D	53
A	151	D#,Eb	50
A#,Bb	143	E	47
B	135	F	45
C	127	F#,Gb	42
C#,Db	119	G	40
D	113	G#,Ab	37
D#,Eb	105	A	35
E	101	A#,Bb	33
F	95	B	31

APPENDIX 11

MUSICAL NOTATION

This appendix will not teach you all about music, but it contains the basic information you need to translate sheet music into XL programs.

Music is written by positioning symbols which represent the length of notes on a framework (called a staff) representing the pitch.





The lengths of notes are indicated by the note shape:

A Semibreve		is twice as long as
a Minim		which is twice as
a Crotchet		long as
a Quaver		which is twice as
		long as

a Semiquaver




Tails on notes may go up  or down . The feathers on quavers and shorter notes may be joined where they appear in groups:



A dot after a note means that it is made half as long again as a normal note:



The mark  is a tie which means the notes are joined together.



Volume is indicated by markings below the stave.

ff Very loud


f Loud

mf Moderately loud

mp Moderately soft

p Soft

pp Very soft

 Get louder

 Get softer

Speed is indicated by markings which may be above or below the stave. Examples are:

Presto	which means Fast
Allegro	Quite fast
Allegro moderato	Moderately fast
Moderato	Medium pace
Andante	Slow
Largo	Very slow

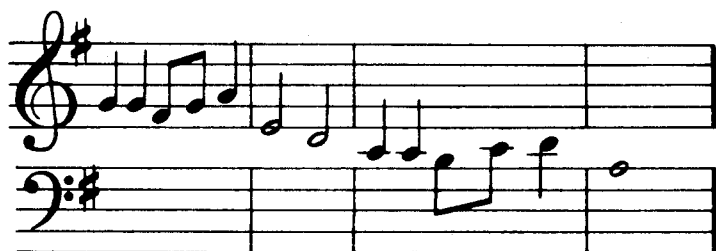
Unfortunately, there are many other Italian words and phrases which may be used. The best thing to do is to adjust your XL program until the speed sounds right, and not worry too much about what is written on the music.

Two other markings which may appear next to notes are # and b. # (sharp) means that the note should be raised by one semitone and b (flat) means that the note is lowered one semitone.

All other markings which may appear on sheet music (and there are many of them) can be ignored.

LAND OF HOPE AND GLORY

Here is the beginning of 'Land of Hope and Glory', one of the examples used in Chapter 11. The notes



shown in this music correspond to the notes

represented by the first item of every group of two in the **DATA** statement in the program. The length of the note, indicated by its shape, corresponds to the second number of each group.

Note the 'sharp' symbols (#) on the lines of the stave corresponding to the note F. These mean that all F's throughout the music are sharpened.

APPENDIX 12

MEMORY MAP

Operating system in ROM	65535
Not Used	55296
ANTIC registers	54784
Input/Output registers	54272
POKEY registers	54016
Not Used	53760
GTIA registers	53504
BASIC ROM	53248
Cartridge Slot or RAM	40960
Screen Memory	32768

Display List	

BASIC Program and Variables	
Free RAM	1792
Operating system and BASIC Storage	1536
	0

NOTE

Locations 16784 to 32767 are not used in the 600XL.

INDEX

ABS	64
ADR	200
Alternative char. set	114
AND	71
Animation	123, 172
Arrays	24
Arithmetic	12, 21
Assembly Language	199
ASC	54
ASCII code	53
ATN	65
BASIC	1
Commands	204
Binary numbers	225
Bit	225
Boolean algebra	71
Branching	34
BYE	204
Byte	225
Cassette unit	4
Cartridges	192
Channels	192
Character sets	115
CHR\$	53
Clock	38
CLOAD	188
CLOG	67
CLOSE	194
CLR	76
Collision detection	184
Colour	105
COLOR	111, 137
COM	205
CONT	41
COS	65
CSAVE	188
Cursor	8
Cursor keys	9
DATA	45
DEG	65

DELETE key	11
DIM	24, 52
DRAWTO	138
Editing	11
END	41
ENTER	188
Error messages	215
EXP	66
Files	192
Filter	102
FOR	31
FRE	76
Functions	63
GET	47
GET#	193
GOSUB	38
GOTO	34
GRAPHICS	108
Graphics Symbols	15
Hexadecimal numbers	225
Hierarchy of operations	13
IF	34
Immediate mode	8
INPUT	27
INPUT#	193
INSERT key	11
INT	64
Joysticks	147
Jumps	34
Keyboard	8
Keywords, BASIC	204
Languages	1
LEN	52
LET	209
LIST	19
LOAD	188
Loading programs	188
LOCATE	141
LOG	67
Loops	31

Lower case	14	ROM	74
LPRINT	209	RUN	19
Machine code	199	SAVE	189
Memory map	243	Saving programs	188
Memory use	74	Scrolling	10, 158
Microprocessor	199	SETCOLOR	105
Monitor	7	SGN	65
Music	77	SIN	65
		SOUND	79
Nested loops	32	SQR	63
NEW	20	STATUS	195
NEXT	31	STEP	31
NOT	73	STICK	147
Numbering systems	225	STOP	41
		Stopping programs	41
ON ... GOSUB	42	STRIG	147
ON ... GOTO	41	String variables	23, 52
OPEN	192	STR\$	54
Operating System	243	Structured programming	201
OR	71	Subroutines	39
PADDLE	147	TAB key	10
Pauses	38	Television	4
PEEK	74	THEN	34
Pixel	115	TO	31
PLOT	111, 138	TRAP	197
POKE	74		
POP	196	Upper case	14
POSITION	49	User defined characters	115
PRINT	8, 18, 48	USR	200
PRINT #	193		
Program	18	VAL	55
PTRIG	148	Variables	21
PUT	193		
		XIO	139
RAD	65		
RAM	74		
Random numbers	67		
READ	45		
Real variables	23		
Register	75		
REM	28		
Reserved words	204		
Reset key	11		
RESTORE	46		
RETURN	39		
Reverse video	14		
RND	67		

Also available from Century Communications

ATARI

Handbooks

0225 9	ATARI XL HANDBOOK	Peter Lupton and Frazer Robinson	£5.95
0513 4	MICRO ENQUIRER: THE ATARI	Ben Woolley and Chris Bidmead (c)	£13.95
0508 8	MICRO ENQUIRER: THE ATARI	Ben Woolley and Chris Bidmead (p)	£8.95

Software and Applications (Books)

0500 2	ATARI XL GRAPHICS HANDBOOK	Peter Lupton and Frazer Robinson	£7.95
0501 0	BEST OF PCW: SOFTWARE FOR THE ATARI	Editors of Personal Computer World	£5.95

SOFTWARE (Cassettes/Book packs and cassettes)

0520 7	BEST OF PCW: SOFTWARE FOR THE ATARI XL	Editors of Personal Computer World	£11.95
--------	--	------------------------------------	--------

GENERAL

Applications and Languages

*0507 X	ASSEMBLER ROUTINES FOR THE 6502	David Barrow	£7.95
*0506 1	ASSEMBLER ROUTINES FOR THE Z80	David Barrow	£7.95
0946 97006 8	COMPUTER ART & GRAPHICS	Axel Brück (c)	£14.95
0115 5	COMPUTER GAMESMANSHIP	David Levy	£7.95
*0561 4	EPSON PRINTER USER'S HANDBOOK	Weber Systems	£9.95
*0650 5	HACKER'S HANDBOOK	Hugo Cornwall	£3.95
0220 8	LOGO PROGRAMMING	Anne Moller	£6.95
*0579 7	PROGRAMMING THE REAL WORLD	Marcus Watney	£6.95
0665 3	TALKING TO THE WORLD	John Newgas (c)	£8.95
*0558 4	TALKING TO THE WORLD	John Newgas (p)	£5.95
*0550 9	TEACH YOURSELF ASSEMBLER: 6502	Paul Overaa	£7.95
0549 5	TEACH YOURSELF ASSEMBLER: Z80	Paul Overaa	£7.95

The Atari 130XE Handbook offers its readers the opportunity of harnessing the power of one of the newest and most exciting microcomputers with its advanced sound and graphics facilities.

Peter Lupton and Frazer Robinson are both professional technical authors and their clear step-by-step introduction opens the Atari 130XE to the beginner by starting with first principles and developing to explain the more advanced features and capabilities of the micro.

The wealth of hints and tips, exciting programs and applications makes the book essential reading for even the experienced programmer. There are lengthy sections on sound and graphic capabilities which are particularly impressive features of the machine.

The Atari 130XE is compatible with the older Atari 400 and 800 micros and the XL range. Although written for the XE series, the *Atari 130XE Handbook* is suitable for all these Atari users.

ISBN 0-7126-9705-5



£7.95

ISBN 0 7126 9705 5

CENTURY COMMUNICATIONS LTD

A division of Century Hutchinson Ltd