

U S E R
GUIDE

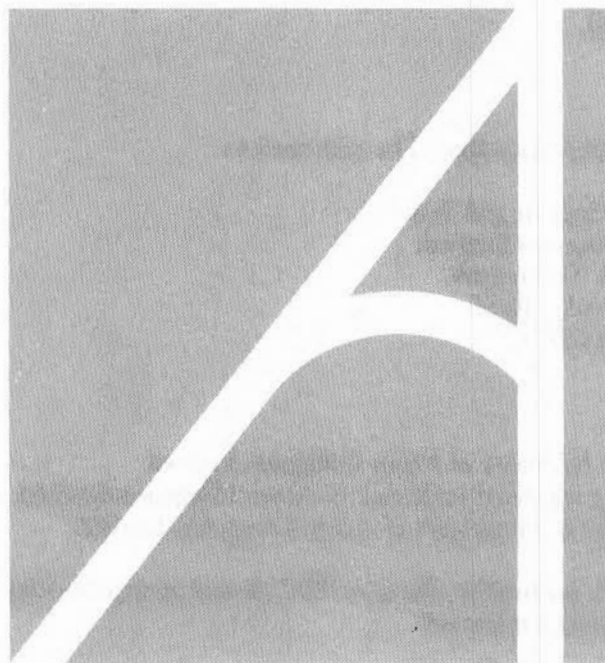


BRITISH BROADCASTING CORPORATION
MICROCOMPUTER SYSTEM



rchimedes

U S E R
G U I D E



BRITISH BROADCASTING CORPORATION
MICROCOMPUTER SYSTEM



Archimedes

Designed, and laser-typeset by Human-Computer Interface Limited, Cambridge.

Copyright © Acorn Computers Limited 1987

Neither the whole nor any part of the information contained in, or the product described in, this guide may be adapted or reproduced in any material form except with the prior written approval of Acorn Computers Limited (Acorn Computers).

All correspondence should be addressed to:

Customer Support and Training,
Acorn Computers Limited,
Cambridge Technopark,
645 Newmarket Road,
Cambridge CB5 8PB.

Acorn is a trademark of Acorn Computers Limited.
Econet is a registered trademark of Acorn Computers Limited.
Archimedes is a trademark of Acorn Computers Limited.

Within this publication, the term 'BBC' is used as an abbreviation for 'British Broadcasting Corporation'.

First published 1987
Issue 2 July 1987
Published by Acorn Computers Limited
Part number 0476,002

The product described in this guide and products for use with it are subject to continuous development and improvement. All information of a technical nature and particulars of the product and its use (including the information and particulars in this guide) are given by Acorn Computers Limited in good faith. However, it is acknowledged that there may be errors or omissions in this guide or in the products it describes. A list of details of any amendments or revisions to this guide can be obtained upon request from Acorn Computers. Acorn Computers welcomes comments and suggestions relating to the product and this guide.

All maintenance and service on the product must be carried out by Acorn Computers' authorised dealers. Acorn Computers can accept no liability whatsoever for any loss or damage caused by service, maintenance or repair by unauthorised personnel. This guide is intended only to assist the reader in the use of this product, and therefore Acorn Computers shall not be liable for any loss or damage whatsoever arising from the use of any information or particulars in, or any error or omission in, this guide, or any incorrect use of the product. Refer to the explicit instructions for installation supplied with upgrades.

WARNING: THIS COMPUTER MUST BE EARTHED

Important: The wires in the mains lead for the computer are coloured in accordance with the following code:

Green and yellow	Earth
Blue	Neutral
Brown	Live

For United Kingdom users

The moulded plug must be used with the fuse and fuse carrier firmly in place. The fuse carrier is of the same basic colour (though not necessarily the same shade of that colour) as the coloured insert in the base of the plug. Different manufacturers' plugs and fuse carriers are not interchangeable. In the event of loss of the fuse carrier, the moulded plug **MUST NOT** be used. Either replace the moulded plug with another conventional plug wired as described below, or obtain a replacement fuse carrier from an Acorn Computers' authorised dealer. In the event of the fuse blowing it should be replaced, after clearing any faults, with a 5-amp fuse that is ASTA approved to BS1362.

For all users

If the socket outlet available is not suitable for the plug supplied, either a different lead should be obtained or the plug should be cut off and the appropriate plug fitted and wired as noted below. The moulded plug which was cut off must be disposed of as it would be a potential shock hazard if it were to be plugged in with the cut off end of the mains cord exposed.

As the colours of the wires may not correspond with the coloured markings identifying the terminals in your plug, proceed as follows:

The wire which is coloured green and yellow must be connected to the terminal in the plug which is marked by one of the following: the letter E, the safety earth symbol, the colour green, or the colour green and yellow.

The wire which is coloured blue must be connected to the terminal which is marked with the letter N, or coloured black.

The wire which is coloured brown must be connected to the terminal which is marked with the letter L, or coloured red.

GUIDELINES FOR SAFE OPERATION

The equipment described in this guide is designed and manufactured to comply with International safety standards IEC65 (BS415) and IEC380 (BS5850), and is intended for use only as a desktop microcomputer. It should not be used for other purposes. It is most important that unpacking and installation is carried out in accordance with the instructions given in the Welcome Guide.

The equipment is robustly constructed but in the interests of continued safe and reliable operation, careful handling and the following guidelines should be observed.

- DO keep the machine within a room temperature of 5 to 35 degrees C (41 to 95 degrees Fahrenheit) and a relative humidity of 15% to 95% (non-condensing).
- DO avoid sudden extremes in temperature, exposure to direct sunlight, heat sources (such as an electric fan heater) and rain.
- DO make sure that the equipment is standing on a suitable horizontal flat surface, allowing enough space for air to circulate when the equipment is in use.
- DON'T spill liquids on the machine. If liquid does spill, turn the machine off immediately and take it to your dealer for assessment.
- DON'T drop the equipment or subject it to excessive bumping and jarring. This is particularly important if you have a hard disc installed.
- DO ensure that wires and cables are routed sensibly so that they cannot be snagged or tripped over. Don't tug or twist any wires or cables, or use them to hang or lift any of the units.
- DON'T poke objects through the ventilation openings in the computer casing, and don't let items such as necklaces or bracelets drop into the openings.
- DON'T exceed a maximum power consumption of 20 watts from the Podule backplane supply.
- DON'T balance any objects or stand other equipment not designed for the purpose, on top of this equipment.
- DO switch off and unplug the equipment and any accessories before opening any unit, to install an upgrade, for example. The main computer unit should normally be operated with the cover attached, but it can safely be switched on with the cover removed, provided that care is taken not to short circuit any connections or to allow any fingers or objects in the area of the fan or disc drives when these are running. Be especially careful with jewellery. Do not attempt to open any display or monitor unit, whether supplied with this equipment or not.
- DO make sure you have read and understood any installation instructions supplied with upgrade kits before attempting to fit them. If you have any doubts, contact your supplier.

CONTENTS

GUIDELINES FOR SAFE OPERATION	iv
INTRODUCTION	1
THE ARCHIMEDES USER GUIDE	1
CONVENTIONS USED IN THIS GUIDE	1
BASIC COMMAND MODE	3
BASIC PROGRAMS	7
ENTERING A PROGRAM	7
EDITING A PROGRAM	8
MORE ABOUT LINE NUMBERS	13
LISTING LONG PROGRAMS	15
COMMENTS AND MULTIPLE STATEMENTS	16
SAVING AND RECALLING PROGRAMS	18
VARIABLES AND EXPRESSIONS	21
WHAT IS A VARIABLE?	21
NUMERIC VARIABLES	22
STRING VARIABLES	28
ARRAYS	36
OUTPUTTING TEXT	45
PRINT FORMATTING	45
THE TEXT CURSOR	48
DEFINING YOUR OWN CHARACTERS	51
INPUTTING INFORMATION	53
INPUTTING DATA FROM THE KEYBOARD	53
INCLUDING DATA AS PART OF A PROGRAM	55
BASIC CONTROL STATEMENTS	59
IF ... THEN ... ELSE	59
IF ... THEN ... ELSE ... ENDIF	61
FOR ... NEXT	63
REPEAT ... UNTIL	67
WHILE ... ENDWHILE	68

CASE ... OF ... WHEN ... OTHERWISE ... ENDCASE	69
GOTO	71
GOSUB ... RETURN	72
ON ... GOTO/GOSUB	72
PROCEDURES AND FUNCTIONS	75
DEFINING AND CALLING PROCEDURES	75
PARAMETERS AND LOCAL VARIABLES	76
ON ... PROC	79
RECURSIVE PROCEDURES	80
FUNCTIONS	82
FUNCTION AND PROCEDURE LIBRARIES	83
FILING SYSTEMS	87
SELECTING THE ADFS	87
LEAVING THE ADFS	87
SELECTING THE ANFS	88
LEAVING THE ANFS	88
DRIVE NUMBERS	89
DISC NAMES	89
FORMATTING A DISC	90
BACKING UP DISCS	92
COPYING USING A SINGLE FLOPPY DISC DRIVE	92
COPYING USING MORE THAN ONE FLOPPY DISC DRIVE	93
DIRECTORIES	94
PATHNAMES	100
DELETING FILES AND DIRECTORIES	101
COPYING AND MOVING FILES	102
FILE DETAILS AND ATTRIBUTES	104
DATA FILES	109
COMMAND FILES	112
*CONFIGURE OPTIONS	113

SCREEN MODES	117
THE MODES AVAILABLE	117
TEXT SIZE	119
GRAPHICS RESOLUTION	119
COLOURS	120
GRAPHICS	129
THE GRAPHICS SCREEN	129
THE LINE COMMAND	129
RECTANGLE AND RECTANGLE FILL	130
CIRCLE AND CIRCLE FILL	130
ELLIPSE AND ELLIPSE FILL	131
GCOL	132
THE GRAPHICS CURSOR	134
USING PLOT TO PRODUCE OTHER SHAPES	135
PATTERN FILLS	148
FLOOD-FILLS	156
COPYING AND MOVING	157
PRINTING TEXT AT THE GRAPHICS CURSOR	158
WINDOWS	161
TEXT WINDOWS	161
GRAPHICS WINDOWS	163
SPRITES	165
THE SPRITE EDITOR	165
SPRITE * COMMANDS	169
PLOTTING SPRITES	171
DEFINING SPRITES FROM THE SCREEN	172
TELETEXT MODE	175
TEXT DISPLAYS	175
TELETEXT GRAPHICS	177

SOUND	181
THE SOUND STATEMENT	182
THE BEATS STATEMENT	184
THE BEAT STATEMENT	185
THE TEMPO STATEMENT	185
KEYBOARD, MOUSE AND FUNCTION KEYS	189
THE KEYBOARD	189
THE MOUSE	191
FUNCTION KEYS	193
INDIRECTION OPERATORS	195
ACCESSING MEMORY LOCATIONS IN GENERAL	195
RESERVING A BLOCK OF MEMORY	195
THE '?' INDIRECTION OPERATOR	195
THE '!' INDIRECTION OPERATOR	197
THE '!' INDIRECTION OPERATOR	197
THE '\$' INDIRECTION OPERATOR	198
BASES	199
HEXADECIMAL NUMBERS	199
BINARY NUMBERS AND BITS	199
PRINTERS	205
CONNECTING YOUR PRINTER	205
DEFINING THE PRINTER TYPE	206
SELECTING THE BAUD RATE	206
PRINTER IGNORE CHARACTERS	207
SENDING OUTPUT TO THE PRINTER	208
ERROR HANDLING AND DEBUGGING	209
GLOBAL ERROR HANDLING	209
LOCAL ERROR HANDLING	210
DEBUGGING	212

BASIC KEYWORDS	215
ABS	215
ACS	216
ADVAL	216
AND	217
APPEND	218
ASC	219
ASN	220
ATN	220
AUTO	221
BEAT	222
BEATS	222
BGET#	223
BPUT#	224
BY	225
CALL	226
CASE	236
CHAIN	237
CHR\$	238
CIRCLE	238
CLEAR	239
CLG	239
CLOSE#	240
CLS	241
COLOUR (COLOR)	241
COS	243
COUNT	244
DATA	245
DEF	246
DEG	247
DELETE	247
DIM	248
DIV	250
DRAW	251
EDIT	252
ELLIPSE	252

ELSE	253
END	254
ENDCASE	255
ENDIF	256
ENDPROC	256
ENDWHILE	257
EOF#	257
EOR	258
ERL	259
ERR	260
ERROR	260
EVAL	261
EXP	262
EXT#	262
FALSE	264
FILL	264
FN	265
FOR	266
GCOL	267
GET	269
GET\$	270
GET\$#	271
GOSUB	271
GOTO	273
HELP	274
HIMEM	274
IF	275
INKEY	277
INKEY\$	278
INPUT	279
INPUT LINE	280
INPUT#	280
INSTALL	281
INSTR(282
INT	283
LEFT\$(283
LEN	285

LET	286
LIBRARY	287
LINE	288
LINE INPUT	288
LIST	289
LISTO	290
LN	291
LOAD	292
LOCAL	293
LOCAL ERROR	294
LOG	295
LOMEM	295
LVAR	296
MID\$(297
MOD	298
MODE	299
MOUSE	300
MOVE	302
NEW	303
NEXT	304
NOT	305
OF	306
OFF	306
OLD	308
ON	308
OPENIN	311
OPENOUT	312
OPENUP	312
OR	313
ORIGIN	314
OSCLI	314
OTHERWISE	316
PAGE	316
PI	317
PLOT	318
POINT	318
POINT(319

POS	320
PRINT	321
PRINT#	324
PROC	325
PTR#	326
QUIT	327
RAD	327
READ	328
RECTANGLE	328
REM	330
RENUMBER	331
REPEAT	332
REPORT	332
REPORT\$	333
RESTORE	333
RESTORE ERROR	334
RETURN	334
RIGHT\$(335
RND	337
RUN	338
SAVE	338
SGN	339
SIN	340
SOUND	341
SPC	342
SQR	343
STEP	344
STEREO	344
STOP	345
STR\$	346
STRING\$(347
SUM	347
SWAP	348
SYS	349
TAB(351
TAN	351
TEMPO	352

THEN	353
TIME	354
TIMES\$	355
TINT	356
TO	357
TOP	357
TRACE	358
TRUE	359
UNTIL	359
USR	360
VAL	361
VDU	361
VOICES	363
VPOS	363
WAIT	364
WHEN	365
WHILE	366
WIDTH	367
VDU COMMANDS	369
VDU 0	371
VDU 1	371
VDU 2	371
VDU 3	371
VDU 4	371
VDU 5	371
VDU 6	371
VDU 7	372
VDU 8	372
VDU 9	372
VDU 10	372
VDU 11	372
VDU 12	372
VDU 13	372
VDU 14	373
VDU 15	373
VDU 16	373

VDU 17,n	373
VDU 18,k,c	373
VDU 19,l,p,r,g,b	373
VDU 20	374
VDU 21	374
VDU 22,n	374
VDU 23,p1,p2,p3,p4,p5,p6,p7,p8,p9	374
VDU 24,x1;y1;x2;y2;	381
VDU 25,k,x;y;	381
VDU 26	381
VDU 27	382
VDU 28,lx,by,rx,ty	382
VDU 29,x;y;	382
VDU 30	382
VDU 31,x,y	382
OPERATING SYSTEM COMMANDS	383
*AUDIO	383
*CHANNELVOICE	383
*CONFIGURE	384
*ECHO	390
*FX	390
*GO	390
*GOS	391
*HELP	391
*IF	391
*IGNORE	391
*KEY	391
*POINTER	392
*QSOUND	392
*SET	392
*SETEVAL	393
*SETMACRO	393
*SHADOW	393
*SHOW	393
*SOUND	393
*SPEAKER	394

*STATUS	394
*STEREO	394
*TEMPO	394
*TIME	394
*TUNING	395
*TV	395
*VOICES	395
*VOLUME	395
*UNSET	396
*FX COMMANDS	397
*FX 0	397
*FX 1	397
*FX 2	397
*FX 3	398
*FX 4	398
*FX 5	400
*FX 6	400
*FX 7	401
*FX 8	401
*FX 9	402
*FX 10	402
*FX 11	402
*FX 12	402
*FX 15	403
*FX 18	403
*FX 19	403
*FX 20	403
*FX 21	403
*FX 25	404
*FX 106	404
*FX 112	404
*FX 113	405
*FX 114	405
*FX 118	405
*FX 120	405
*FX 124	405

*FX 125	405
*FX 126	406
*FX 138	406
*FX 139	406
*FX 143	406
*FX 144	406
*FX 153	407
*FX 156	407
*FX 162	407
*FX 163	408
*FX 178	408
*FX 181	408
*FX 196	408
*FX 197	409
*FX 200	409
*FX 201	409
*FX 202	409
*FX 203	410
*FX 204	410
*FX 211	411
*FX 212	411
*FX 213	411
*FX 214	411
*FX 216	411
*FX 217	412
*FX 218	412
*FX 219	412
*FX 220	412
*FX 221	413
*FX 222	413
*FX 223	413
*FX 224	414
*FX 225	414
*FX 226	414
*FX 227	415
*FX 228	415
*FX 229	415

*FX 230	416
*FX 238	416
*FX247	416
*FX 254	417
*FX 255	417
THE BASIC SCREEN EDITOR	419
ENTERING THE EDITOR	419
THE EDIT SCREEN	420
SAVING AND LOADING PROGRAMS	422
SEEING OTHER PARTS OF YOUR PROGRAM	423
RENUMBERING THE PROGRAM	424
MARKING A LINE	426
LINE COMMANDS	426
SEARCHING AND REPLACING	429
SETTING VARIOUS OPTIONS	430
USER DEFINED KEYS	432
FULL USE OF WINDOWS	432
KEYBOARD SUMMARY	434
ERROR MESSAGES	438
APPENDIX A – MINIMUM ABBREVIATIONS	443
APPENDIX B – BASIC ERRORS	449
APPENDIX C – CHARACTER CODES	453
LATIN ALPHABET 1	453
LATIN ALPHABET 2	454
LATIN ALPHABET 3	455
LATIN ALPHABET 4	456
GREEK ALPHABET	457
BFONT CHARACTER CODES	458
APPENDIX D – TELETEXT CHARACTER CODES	461
TELETEXT ALPHANUMERIC CHARACTER CODES	461
TELETEXT GRAPHICS CHARACTER CODES	463

APPENDIX E - SCREEN MODES	465
APPENDIX F - INKEY VALUES	467
APPENDIX G - PLOT CODES	471
APPENDIX H - VDU COMMANDS	473
APPENDIX I - OPERATING SYSTEM COMMANDS	475
APPENDIX J - ADFS COMMANDS	477
APPENDIX K - *FX COMMANDS	479
APPENDIX L - PIN CONNECTIONS	481
VIDEO	481
SERIAL LINE	482
PRINTER	482
ECONET	483
INDEX	485

INTRODUCTION

THE ARCHIMEDES USER GUIDE

This guide describes the two levels of commands built into the Archimedes computer, and available for those wishing to program it themselves: the operating system commands, and the BBC BASIC programming language commands.

BBC BASIC is one of the most popular and widely-used programming languages. It consists of special keywords from which the programmer can create sequences of instructions, or programs, to be carried out by the computer. Such programs might perform calculations, create graphics on the screen, manipulate data, or carry out virtually any action involving the computer and the devices connected to it. Several examples of programs written in BBC BASIC are provided on the Archimedes Welcome Disc.

The BASIC language operates within an environment provided by the computer's operating system. The operating system is responsible for controlling the devices available to the computer, such as the keyboard, the screen, and the filing system. For example, it is the operating system which reads each key you press and displays the appropriate character on the screen. Commands can be passed to the operating system from within BASIC by prefixing them with an asterisk '*'.

The first chapters of this guide explain how to program in BASIC, and introduce many of the commands provided by the language. A complete alphabetical list of the BASIC keywords is given in the chapter: **BASIC KEYWORDS**. The last four chapters, and the appendices, list the features provided by the operating system, and the commands available to control them.

CONVENTIONS USED IN THIS GUIDE

The following conventions are applied throughout this guide:

- Specific keys to press are denoted as `Delete`, `Ctrl`, and so on.
- Text you type on the keyboard and text that is displayed on the screen appears as follows:

```
PRINT "Hello"
```

- After entering any text, press `[Enter]` to tell the computer that you have completed the line and that you want the computer to act upon it.
- Extra spaces are inserted into program listings to aid clarity.
- Program listings are indented to illustrate the structure of the programs.

If at any time you wish to interrupt a program the computer is executing you can do so safely by pressing `[Escape]`. Do not be afraid to experiment. Try modifying the programs listed in this book and writing new ones of your own.

BASIC COMMAND MODE

When you enter BASIC it is in command or interactive mode. This means that you can type commands and the computer responds straight away. For example, if you type

```
PRINT "Hello"
```

the computer displays the following on the screen:

```
Hello
```

PRINT is an example of a keyword which the computer recognises. It instructs the computer to display on the screen whatever follows the PRINT statement.

If you make a mistake, the computer may not be able to make sense of what you have typed. For example, if you type

```
PRINT "Hello
```

the computer responds with the message:

```
Missing "
```

This is an error message. It indicates that the computer cannot obey your command because it does not follow the rules of BASIC (in this case because it could not find a second quotation mark).

If PRINT is followed by any series of characters enclosed in quotation marks, then these characters are displayed on the screen exactly as you typed them. Thus:

```
PRINT "12 - 3"
```

produces the output:

```
12 - 3
```

PRINT, however, can also be used to give the result of a calculation. For example, typing

```
PRINT 12 - 3
```

produces the output:

```
9
```

In this case, because the sum was not enclosed in quotation marks, the computer performed the calculation and displayed the result.

Similarly, multiplication and division can be performed using the symbols '*' and '/'. For example:

```
PRINT 12 * 13  
PRINT 111 / 11
```

Some commands, although they have an effect on the computer, do not give evidence that anything has changed. If, for example, you type

```
LET FRED = 12
```

nothing obvious happens. Nevertheless, the computer now knows about the existence of a variable called FRED which has the value 12. A variable is a name which can have different values assigned to it. It is described in more detail later in this manual.

Now if you type

```
PRINT FRED / 3
```

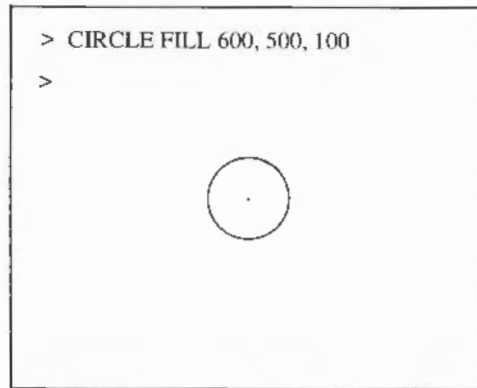
the computer responds by displaying the number 4.

The program below illustrates how you can give commands to produce some graphics on the screen:

```
MODE 1  
CIRCLE FILL 600,500,100
```

The `MODE` command sets up the computer to produce medium resolution graphics. It also clears the screen.

The `CIRCLE FILL` command tells the computer to draw a circle at a position 600 graphics units from the left of the screen and 500 units up from the bottom. This is near the centre of the screen because the screen is 1280 units across and 1024 units high. The third number tells the computer how big the circle should be, in this case giving a radius of 100 graphics units.



BASIC PROGRAMS

ENTERING A PROGRAM

A program is a list of instructions to be carried out by the computer. These instructions are stored in memory and are only executed when you tell the computer to do so. Each line of a program is numbered so that it can be referred to more easily. For example, type the following:

```
10 PRINT "Hello"
```

Note that nothing happens. Now type

```
RUN
```

The message is displayed on the screen. The number 10 at the start of the line is called the line number, and identifies the text after it as a program statement to be stored in memory, rather than as a command to be executed immediately.

You can type spaces either between the start of the line and the line number, or between the line number and the instruction without effecting the execution of the program.

```
10 PRINT "Hello"
```

and

```
10PRINT "Hello"
```

are equally valid.

One of the advantages of programs is that they can be executed repeatedly: typing RUN again here causes Hello to be displayed a second time – there is no need to type the complete PRINT "Hello" statement again.

The following is a simple program demonstrating the use of a variable and the INPUT statement:

```
10 PRINT "Can you give me a number ";
20 INPUT number
30 PRINT "The number you typed was ";number
```

Note that you must press at the end of each line.

The line numbers determine the order in which the computer executes these instructions. They can take any whole value between zero and 65279.

Now RUN this program. The computer obeys line 10 and displays the message:

```
Can you give me a number ?
```

The question mark is added automatically by the execution of line 20. The keyword INPUT instructs the computer to wait for you to type something, in this case a number. Type

```
6
```

Line 30 is now obeyed, and the following message is displayed:

```
The number you typed was 6
```

EDITING A PROGRAM

Replacing and adding lines

Once you have entered a program, you may wish to make changes to it. You can of course type in a whole new version of the program, but there are quicker methods available. To see the program which is currently stored in memory, type

```
LIST
```

Lines 10, 20 and 30 are listed on the screen.

To add extra lines to the program, type in the new line with an appropriate line number:

```
5 PRINT "Hello"
40 PRINT "Twice ";number " is ";2*number
```

and then:

```
LIST
```

Note that these two extra lines are added to the program in such a way that the line numbers are listed in numerical order:

```
5 PRINT "Hello"
10 PRINT "Can you give me a number ";
20 INPUT number
30 PRINT "The number you typed was ";number
40 PRINT "Twice ";number;" is ";2*number
```

To replace lines, enter the new line with the line number of the one which is to be replaced. For example:

```
40 PRINT number;" squared is ";number*number
```

Now when you type

```
LIST
```

the following is displayed:

```
5 PRINT "Hello"
10 PRINT "Can you give me a number ";
20 INPUT number
30 PRINT "The number you typed was ";number
40 PRINT number;" squared is "; number*number
```

Editing lines

If you wish to alter only part of a line, for example, to correct a single spelling mistake, you can do so using the cursor edit keys. These are the arrow keys to the right of the main keyboard.

Suppose you want to change the word `typed` to `entered` on line 40.

Begin by pressing the `[↑]` key twice. The original cursor position which was under line 30 becomes a white square and the cursor moves up to the start of line 30.

Press `[Copy]` a few times. The cursor moves along line 30, the white square moves along as well, and line 30 is copied underneath line 40. Keep on pressing `[Copy]` until the word `typed` is copied and then stop.

If you hold the key down, the repeat action allows you to move the cursor quickly across the screen. A quick press and release gives you precise control, moving one character position. The following is displayed on your screen:

```
5 PRINT "Hello"
10 PRINT "Can you give me a number ";
20 INPUT number
30 PRINT "The number you typed_was ";number
40 PRINT number;" squared is "; number*number
30 PRINT "The number you typed
```

Press `[Delete]` until the word `typed` is deleted from the new line 30. Note that the cursor on the old line 30 has not moved:

```
5 PRINT "Hello"
10 PRINT "Can you give me a number ";
20 INPUT number
30 PRINT "The number you typed_was ";number
40 PRINT number;" squared is "; number*number
30 PRINT "The number you
```

Type the word entered and press **Copy** to copy the rest of line 30 to your new version.

Press **Enter**. The white square disappears and the cursor moves to the start of a new line. Now type LIST to produce the following:

```
5 PRINT "Hello"
10 PRINT "Can you give me a number ";
20 INPUT number
30 PRINT "The number you entered was ";number
40 PRINT number;" squared is "; number*number
```

There are no restrictions on how much you move the cursor around when you are copying. You can use the right and left arrow keys to miss out parts of lines or to repeat them. You can also copy from lots of different lines on to your new line as you go.

Deleting lines

You can either delete lines one at a time, or delete a group of lines at once using the DELETE command.

To delete a single line, you just type the line number followed by **Enter**. To delete line number 5, for example, type

```
5
```

To check that line 5 is deleted, type

```
LIST
```

and the computer displays the following:

```
10 PRINT "Can you give me a number ";
20 INPUT number
30 PRINT "The number you typed was ";number
40 PRINT number;" squared is "; number*number
```

The DELETE command allows you to delete a number of consecutive lines in three different ways:

- *By deleting a block of lines.* To delete all line numbers between 10 and 30, type

```
DELETE 10,30
```

- *By deleting from the beginning of a program.* To delete all lines from the beginning of the program to line 30, type

```
DELETE 0,30
```

The number zero is the minimum line number that can be used in a program. Therefore, all lines from the start of the programs to line 30 are deleted.

- *By deleting from a line to the end of the program.* To delete all lines from line 20 to the end of the program, for example, type

```
DELETE 20,65279
```

The number 65279 is the maximum line number that can be used in a program, so in this case all lines from line 20 to the end of the program are deleted.

Deleting whole programs

Before you enter a new program, make sure no program currently exists in memory. If it does, the lines of the new program you enter will get mixed up with the lines of the existing program, and this could produce strange results!

To delete any existing program, you can use the DELETE command described above, but an easier method is to type

```
NEW
```

This tells the computer to forget about any existing program, and to be ready to accept a new one.

Although the `DELETE` and `LIST` commands combined with cursor editing are fine for making small changes to a BASIC program, you should note that the BASIC Editor is much more versatile. See the chapter: **THE BASIC SCREEN EDITOR** for details of using this program.

MORE ABOUT LINE NUMBERS

Renumbering programs

There may be occasions when you want to change the line numbers of a program without changing their order. The command to use is `RENUMBER`. This facility is particularly useful when you want to insert a large number of lines between two existing ones.

You can specify two numbers after typing the `RENUMBER` command. The first number tells the computer what you want the new first program line number to be. The second number tells the computer how much to add to each line number to get the next one. For example,

```
RENUMBER 100,20
```

makes the first line into line 100 and numbers the remaining lines 120, 140, 160, and so on.

If you leave out the second number in the `RENUMBER` command, the computer automatically increments the line numbers in steps of 10. So, for example, you might want to renumber the following program:

```
23 PRINT "This demonstrates"  
24 PRINT "the use of"  
48 PRINT "the very useful"  
67 PRINT "RENUMBER command"
```

Typing

```
RENUMBER 100
```


and then

LIST

produces the following display:

```
100 PRINT "This demonstrates"  
110 PRINT "the use of"  
120 PRINT "the very useful"  
130 PRINT "RENUMBER command"
```

Typing

RENUMBER

without including a number after the command, means that your program lines are renumbered 10, 20, 30, 40 and so on.

Automatic numbering of programs

You do not have to type line numbers at the beginning of each new program line. The computer does it automatically when given the AUTO command.

For example, type

AUTO

The computer displays the number 10 on the line below. If you type the first program line and press , the number 20 appears on the next line, and so on.

To switch off this automatic line numbering, press .

You can start a program at a line other than line 10 by following the AUTO command with the first line number you wish to use. Thus,

AUTO 250

generates lines which are numbered 250, 260, 270, and so on.

You can also specify the number of spare lines between each of your program lines by adding a second number, separated from the first by a comma. Thus,

```
AUTO 250,15
```

starts at line number 250 and subsequently increases the line numbers in steps of 15, generating lines numbered 250, 265, 280, and so on.

LISTING LONG PROGRAMS

Listing sections of programs

The `LIST` command, used above to display the current program on the screen, can be used to look at part of a program. This is particularly useful if the program is very big and you want to concentrate on one part of it.

To look at one particular line, for example, type

```
LIST 40
```

To look at a number of consecutive lines, for example, type

```
LIST 20, 40
```

To see from the beginning of the program up to a particular line, for example, type

```
LIST ,30
```

To display from a particular line to the end of the program, for example, type

```
LIST 20,
```

Halting listings

If you list more of a program than can fit on the screen all at once, the beginning of the listing disappears off the top of the screen before you have time to read it. There are three ways of getting round this problem.

- Pressing the **Scroll Lock** halts the listing; pressing it again allows the listing to continue. This enables you to step through chunks of the listing.
- Holding down **Ctrl Shift** together after typing LIST, halts the displayed listing on the screen. To continue the listing, take your finger off either **Ctrl** or **Shift**.
- Putting the computer into paged mode. This is the most reliable method. To enter this mode press **Ctrl N**, then type LIST. The listing stops as soon as the whole screen is filled. To display the next screenful of listing, press **Shift**. This method ensures that you will not miss any of the listing. To cancel the effect of **Ctrl N**, type **Ctrl O** when the listing is finished.

In addition to the methods described for halting listings, you can also slow the listing down by pressing **Ctrl**. This makes the screen halt for 1/25 of a second between each new line. Thus it takes a second to scroll one screenful in a 25-line text mode.

COMMENTS AND MULTIPLE STATEMENTS

Comments

When writing programs, especially long or complex ones, it is a good idea to insert comments to remind you what each part of the program is doing. This is done by using the REM keyword which is short for 'remark'. REM tells the computer to ignore the rest of the line when it executes the program. For example, to add comments to the following program:

```
10 PRINT "Can you give me a number ";
20 INPUT number
30 PRINT "The number you typed was ";number
40 PRINT number;" squared is "; number*number
```

type

```

5 REM Read in a value and assign it to number
25 REM Now print out the number given.
35 REM And its square

```

and then

LIST

to display the complete program:

```

5 REM Read in a value and assign it to number
10 PRINT "Can you give me a number ";
20 INPUT number
25 REM Now print out the number given.
30 PRINT "The number you typed was ";number
35 REM And its square
40 PRINT number;" squared is "; number*number

```

You may like to add further REM statements to underline comments or leave space above them to make them clearer:

```

5 REM Read in a value and assign it to number
6 REM -----
10 PRINT "Can you give me a number ";
20 INPUT number
24 REM
25 REM Now print out the number given
26 REM -----
30 PRINT "The number you typed was ";number
34 REM
35 REM And its square
36 REM -----
40 PRINT number;" squared is "; number*number

```

Multiple statements

A line of BASIC can contain up to 238 characters and can be spread over several lines on the screen. In all the programs given so far, each line of BASIC contains a single statement. Several statements, however, may be placed on one line separated by colons. For example:

```
10 PRINT "Can you give me a number ";":INPUT number
30 PRINT "The number you typed was ";number: REM print out the number
40 PRINT number;" squared is "; number*number: REM and its square
```

Note that REM statements can only be placed at the end of a line since the whole of the rest of the line is ignored. If you alter the program so that line 30 reads as follows:

```
30 REM print out the number: PRINT "the number you typed was ";number
```

you will prevent the PRINT statement being executed.

SAVING AND RECALLING PROGRAMS

You can save a copy of the current program on a floppy disc at any time. This allows you to recall (load) it at a later date, without having to retype all the instructions.

Saving a program

Before you can save a program onto a floppy disc, you must make sure the disc is formatted. Formatting prepares a disc to receive data. For information on how to format a floppy disc, see the chapter: **THE ADVANCED DISC FILING SYSTEM.**

To save a program, in this case a program called `prog1`, insert a formatted floppy disc into the drive and type

```
SAVE "prog1"
```

The program with the name `prog1` is now saved onto the floppy disc.

The name you use when saving a program can contain up to 12 characters. At this stage, you should confine your names to numbers and upper- and lower-case letters. Other characters may be used but some have special meanings. A full explanation of these is given in the chapter: **THE ADVANCED DISC FILING SYSTEM**.

After using `SAVE`, your program remains in memory and is unaltered in any way. You can still edit, `LIST`, `RUN`, and so on.

Another capability of the `REM` statement is that it allows you to give the program name for use by the `SAVE` command. The filename must be preceded by a '>' character. Thus, if the first line of the program is

```
10 REM >deskTop
```

all you need to do is type the `SAVE` command on its own, and the name `deskTop` will be used to save the program.

Loading a program

To load a program which you have previously saved, in this case `prog1` type

```
LOAD "prog1"
```

The `LOAD` operation replaces the current program with the one from the disc. You can check this by listing the program currently in memory.

In addition to loading a program, you can add a program to the end of the current one using the `APPEND` command. The appended program is renumbered to ensure that its line numbers start after those of the initial program.

The statements `INSTALL` and `LIBRARY` may be used to add libraries of procedures and functions to the current program (see the chapter: **PROCEDURES AND FUNCTIONS** for details).

Other disc operations

BASIC programs are not the only type of information that can be stored on disc. You can also, for example, store text from a word processor, data for use by a program, or screen displays. In general, information saved on disc is called a file.

You can perform many operations on the files on a disc, including:

Copying individual files from one disc to another.

Copying all the files from one disc to another.

Renaming files.

Deleting files.

Protecting files so they cannot be deleted.

Grouping files together into 'directories'.

These operations are described in the chapter: **THE FILING SYSTEMS.**

VARIABLES AND EXPRESSIONS

WHAT IS A VARIABLE?

A variable has a name and a value associated with it. The name, for example FRED or number or a single letter such as x, allows it to be identified and its value to be accessed. This value can be changed and retrieved as many times as required.

There are three different types of variables used to store different types of information. These are:

- Integer variables which can only store whole numbers
- Floating point variables, which can store either whole numbers or fractions
- String variables which store characters.

Each type is distinguished by the last character of the variable name. A name by itself, like Fred, signifies a floating point variable, Fred% an integer variable and Fred\$ a string variable.

The rules for naming variables are as follows:

- They can contain digits and unaccented upper- and lower-case letters
- Two additional 'letters' are '_' and '\$'
- They must not start with a digit
- They must not start with a BASIC keyword.

All the following names are allowed:

```
x
xpos
XPOS
Xpos
x_position
```


greatest_x_position
position_of_X
XPOS1

Note that upper- and lower-case letters are regarded by the computer as being different, so that XPOS, xpos and Xpos are three separate variables.

The following names are not allowed:

1pos	It does not begin with a letter.
TOTAL_x	It begins with TO which is a BASIC keyword.
x-pos	It contains a minus sign.
X Position	It contains a space.
X.pos	It contains a punctuation mark.

It is very easy to be caught out by the rule which says that the variables must not start with a BASIC keyword. The best way to avoid this problem is to use lower-case variable names since BASIC keywords are all in upper-case. This has the added advantage of making the program easier to read.

The values of the current variables are displayed at any time by typing LVAR at the BASIC prompt and then pressing .

NUMERIC VARIABLES

Floating point numbers and integers

Floating point variables can represent both whole numbers (integers) and decimal fractions, but integer variables can only store whole numbers. For example, the assignments

```
LET number = 4/3  
LET number% = 4/3
```

leave the variables with the following values:

```
number      is 1.33333333
number%     is 1
```

In the case of the integer variable, the decimal fraction part has been lost.

The advantages, however, of using integer variables are:

- They are processed more quickly by the computer
- They occupy less memory
- They are accurate (decimal numbers are only accurate to 9 figures).

The range and accuracy of floating point and integer variables is summarised below:

	Integers	Floating point numbers
Range	-2147483648 to 2147483647	-1.7×10^{38} to 1.7×10^{38}
Accuracy	absolute	9 significant figures
Stored in	4 bytes	5 bytes

The \wedge symbol is used here to describe the range of floating point numbers. In BASIC, the \wedge operator means 'raised to the power'. Thus `PRINT 2^4` will print two raised to the power of four: sixteen. So, the number 1.7×10^{38} means 1.7 times ten raised to the power of 38: 1 with 38 zeros after it.

Another way of denoting powers of ten is to use 'E' notation. The number 1.7×10^{38} may be written 1.7E38 in 'E' notation. Similarly, 1 234 567 may be written 1.234567E6, as the 'E6' part means 'times ten to the sixth', which is a million. BASIC uses 'E' notation when accepting floating point numbers, and may be made to print numbers in this way.

Assigning values to variables

The value assigned to a numeric (floating point or integer) variable can be specified as:

- a single number
- the current value of another variable
- an expression
- the result of a function.

For example:

```
LET base      = 3
LET height    = 4
LET area      = (base * height)/2
LET hypot     = SQR(base*base + height*height)
```

$(base * height) / 2$ is a mathematical expression consisting of the variables `base` and `height`, and arithmetic operations to be performed on them.

`SQR` is a function which returns the square root of a number, in this case the expression $(base*base + height*height)$.

The above assignments leave the variables with the following values:

```
base   is 3
height is 4
area   is 6
hypot  is 5
```

Note that giving a new value to `base` or `height` does not automatically update `area` or `hypot`. Once the expression is evaluated using the values of `base` and `height` current at that time, it is forgotten. In other words, `area` and `hypot` only know what value they contain, not how it was obtained.

The use of LET is optional. For example,

```
LET x = x+1
```

is equivalent to:

```
x = x+1
```

Using LET, however, makes it easier initially to understand what is happening. On its own $x = x+1$ looks like an unbalanced equation. Using LET makes it clear that the '=' is not being used in its usual sense but as shorthand for 'become equal': LET $x = x+1$ can be read as 'let x become equal to its old value with one added to it'.

An alternative way of expressing this is to use:

```
x += 1
```

This means 'let x become equal to itself with one added to it'. Similarly,

```
x -= 3
```

means 'let x become equal to itself with three subtracted from it'.

Special integer variables

The 26 integer variables A% to Z% are treated differently from the others. They are called resident integer variables because they are not cleared when the program is RUN, or when NEW or BREAK is used. This means that they can be used to pass values from one program to another.

A special integer pseudo-variable is TIME. TIME is an elapsed time clock which is incremented every hundredth of a second while the computer is switched on. It is used to find out how long something takes by putting the following statements around a program:

```
T% = TIME
.....
PRINT TIME - T%
```

TIME may be assigned a starting value just like any other variable. So, for example, the statement above could be replaced by:

```
TIME = 0
.....
PRINT TIME
```

Arithmetic operators

The full list of arithmetic operators is given in the table on the following page. Each operator is assigned a priority. When an expression is being evaluated, this priority determines the order in which the operators are executed. Priority one operators are acted upon first, and priority seven last.

Priority	Operator	
1	-	Unary minus
	+	Unary plus
	NOT	Logical NOT
	FN	Functions
	()	Brackets
	? ! \$	Indirection operators
2	^	Raise to the power
3	*	Multiplication
	/	Division
	DIV	Integer division
	MOD	Integer remainder
4	+	Addition
	-	Subtraction
5	=	Equal to
	<>	Not equal to
	<	Less than
	>	Greater than
	<=	Less than or equal to
	>=	Greater than or equal to
	<<	Shift left
	>>	Arithmetic shift right
	>>>	Logical shift right
6	AND	Logical and bitwise AND
7	OR	Logical and bitwise OR
	EOR	Logical and bitwise Exclusive OR

For example, $12+3*4^2$ is evaluated as $12+(3*(4^2))$ and produces the result 60.

Operators with the same priority are executed left to right, as they appear in the expression. Thus, $22 \text{ MOD } 3 / 7$ is evaluated as $(22 \text{ MOD } 3) / 7$.

Note that the shift operators are entered by typing two '>' or '<' symbols, and should not be confused with the «« and »» characters in the ISO Latin1 alphabet.

STRING VARIABLES

Assigning values to string variables

String variables may be used to store strings of characters, constituting words and phrases. Each string can be up to 255 characters long. The following gives some examples of strings:

```
day$ = "Monday"
Date$ = "29th February"
space$ = " "
ADDRESS$ = "10 Downing Street, London"
Age$ = "21"
```

Note that the variable `Age$` is assigned a string containing the two characters '2' and '1', and not the number 21. So, if you type

```
Real_Age$ = 21 * 2
```

the result will not be "42" because BASIC cannot do arithmetic with strings. Instead, the error message:

```
Type mismatch: number needed
```

appears on the screen, indicating that only a string expression can be assigned to a string variable. A type mismatch error can also be caused by an attempt to multiply strings, as in:

```
total$ = "12"*"32"
```

You should note that the 'null' string "" is valid. This is a string containing zero characters. In comparisons, it is less than any other string (except, of course, another null string).

In order to obtain a double quotation character, ", in a string, you use two of them adjacent to each other. For example, to print the string A"here, you would use:

```
PRINT "A""here"
```

Joining strings together

Two strings may be joined together, or more correctly speaking concatenated. The '+' operator is used to indicate this:

```
10 Road$ = "Downing Street"
20 City$ = "London"
30 PRINT Road$ + " " + City$
```

Typing RUN produces the following:

```
Downing Street London
```

The '+=' operator can also be used, and as the following program shows, produces the same output as '+':

```
10 Address$ = "Downing Street"
20 Address$ += " "
30 Address$ += "London"
40 PRINT Address$
```

Note, however, that the '-=' operator is meaningless when applied to strings and produces an error message.

Splitting strings

As well as joining two strings together, the computer can split a string into smaller sequences of characters. Three functions are provided for doing this.

- LEFT\$ (A\$, n) which gives the first (left hand end) 'n' characters of a string.
- RIGHT\$ (A\$, n) which gives the last (right hand end) 'n' characters of a string.
- MID\$ (A\$, n, m) which gives 'm' characters from the middle, beginning at the nth character.

For example,

```
PRINT LEFT$("HELLO",2) , RIGHT$("THERE",2) , MID$("GORDON",3,2)
```

gives

```
HE RE RD
```

and

```
10 title$ = "Moonlight Sonata"
20 left_of_string$ = LEFT$(title$,4)
30 right_of_string$ = RIGHT$(title$,6)
40 middle_of_string$ = MID$(title$,5,9)
50 PRINT left_of_string$
60 PRINT right_of_string$
70 PRINT middle_of_string$
```

produces the following when RUN:

```
Moon
Sonata
light Son
```

Each of these functions has a convenient shorthand form:

- LEFT\$ (A\$) gives all but the last character of the string
- RIGHT\$ (A\$) gives the last character of the string

– `MID$(A$, n)` gives all the characters from the `n`th to the last

For example:

```
10 PRINT LEFT$("Hello")
20 PRINT RIGHT$("Hello")
30 PRINT MID$("Hello", 3)
```

produces the following:

```
Hello
o
llo
```

Replacing part of a string

`LEFT$, RIGHT$` and `MID$` may be used to replace part of a string. In each case the number of new characters equals the number of characters being replaced, and the string stays the same length. The number of characters being changed can be determined by the length of the replacement string. Thus:

```
10 A$ = "Hello there."
20 MID$(A$, 7) = "Susan"
30 PRINT A$
40 LEFT$(A$) = "Howdy"
50 PRINT A$
60 RIGHT$(A$) = "!"
70 PRINT A$
```

produces:

```
Hello Susan.
Howdy Susan.
Howdy Susan!
```

Alternatively, you can give the maximum number of characters to be replaced. Then, if the length of the replacement string is less than the given value, all of it is

used. Otherwise only the first designated number of characters have an effect. For example,

```
10 A$ = "ABCDEFGHJIJ"
20 RIGHT$(A$, 3) = "KL"
30 PRINT A$
40 LEFT$(A$, 4) = "MNOPQR"
50 PRINT A$
60 MID$(A$, 4, 3) = "STUVW"
70 PRINT A$
```

produces:

```
ABCDEFGHKL
MNOPEFGHKL
MNSTUGHKL
```

There are also BASIC keywords to produce a long string consisting of multiple copies of a shorter string, to find the length of a string, and to determine whether one string is contained within the other. These keywords are:

- STRING\$(n, A\$), which returns a string consisting of 'n' copies of A\$.
- LEN(A\$), which gives the length of string A\$.
- INSTR(A\$, B\$), which looks for the string B\$ within the string A\$ and returns the position of the first place where it is found.

For example,

```
PRINT STRING$(20, "+-")
```

produces the output:

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

```
PRINT LEN("PAUL")
```

prints the number '4'

and

```
A$ = "Great Britain"
PRINT LEN(A$)
```

produces the result 13. Note that the space is treated like any other character.

```
A$ = "Great Britain"
PRINT INSTR(A$, "it")
```

prints '9' because the string `it` is contained in `Great Britain` at the ninth position.

If the substring in the `INSTR` function is not present in the first string, then '0' is returned. Note also that you can start the search for the substring at any position, not just from the start of the substring. This is done by specifying a third parameter, so that for example,

```
PRINT INSTR("'ello 'ello'", "'ello", 2)
```

will print '7', since the first occurrence of the substring will be skipped.

How characters are represented

Every character and symbol which can be reproduced on the screen is represented within the computer by a number in the range zero to 255. The system used to assign numbers to characters and symbols is known as ASCII which stands for American Standard Code for Information Interchange. It is wise to follow such a standard so that different computers can all understand the same numerical alphabet.

BASIC provides a pair of functions for converting characters to their ASCII number-codes and back again. These are:

- `ASC("A")`, which gives the ASCII code of the first character of a string.

- CHR\$(n), which gives the character string whose ASCII code is 'n'.

Converting between strings and numbers

There are three keywords which convert between strings and numbers:

- VAL(A\$), which converts a string of digits A\$ into a number.
- STR\$(n), which converts the number 'n' into a string.
- EVAL(A\$), which evaluates the string A\$ as though it were a BASIC expression.

VAL returns the value of string, up to the first non-numeric character.

For example:

```
PRINT VAL("10to10")
```

prints the value 10, since all the characters after the 't' are ignored. The string may, however, begin with a '+' or '-'. Thus,

```
number = VAL("-5")
```

assigns the value -5 to number. If, however, the string does not start with a number or a plus or minus sign, VAL returns '0'.

EVAL, however, considers the whole string as an expression, allowing operators and variable names to occur within it. Variables must be assigned values beforehand.

```
10 radius = 5
20 circumference = EVAL("2*PI*radius")
30 PRINT circumference
```

When this program is run the value printed is 78.5398163, which is the value of PI (3.14159265) multiplied by 5 squared.

STR\$ performs the opposite conversion to the above two functions. It takes the number given and returns a string containing the digits in the number. For example,

```
10 A = 45
20 B = 30.5
30 A$ = STR$(A)
40 B$ = STR$(B)
50 PRINT A + B
60 PRINT A$ + B$
```

produces the following when it is RUN:

```
      75.5
4530.5
```

STR\$~ gives the hexadecimal version. Thus,

```
10 A = 45
20 A$ = STR$~(A)
30 PRINT A$
```

produces:

```
2D
```

For an explanation of hexadecimal numbers, see the chapter: BASES.

ARRAYS

Arrays are groups of variables. Each array has a name which applies to all variables in the group. The individual members, known as the elements of the array, are identified by a subscript. This is a whole number (zero or greater) indicating the element's position within the array. For example, $A(0)$ is the first element in the array named 'A', and $A(1)$ is the second element.

The DIM statement informs the computer as to how many elements you wish to use in the array. For example,

```
DIM A(9)
```

allocates space in the computer's memory for ten elements, each called 'A', but each having a different subscript, zero to nine. The DIM statement also assigns the value zero to each of these elements, which may then be individually assigned values, just like any other variables. For example:

```
A(1) = 0.56  
A(2) = A(1) + 4
```

The example shown above is of a one-dimensional array: it may be thought of as a line of variables, numbered from 0 to 9 in a sequence. More dimensions may be used. Two dimensional arrays in which the individual variables are identified by two subscripts can be thought of as the printing on a TV screen. Each character printed on the screen is at a particular position from the left, and a particular position from the top.

A two dimensional array may be defined as follows:

```
DIM B(2,2)
```

This allocates space for nine elements, each called 'B', and each identified by two subscripts as shown in the following table:

B(0,0)	B(0,1)	B(0,2)
B(1,0)	B(1,1)	B(1,2)
B(2,0)	B(2,1)	B(2,2)

Arrays may have as many dimensions as you like, and may hold floating point numbers, integers, or strings. For example,

```
DIM string$(1,3,2)
```

allocates space for 24 string variables, each of them containing up to 255 characters.

The subscript need not be specified as a number – a variable can be used instead. For example:

```
10 DIM A(9)
20 X = 6
30 A(X) = 3
40 A(A(X)) = 1
```

This gives A(6) the value 3, and A(3) the value 1.

Any arithmetic expression may be used as a subscript. Since subscripts can only be whole numbers, any expression giving a floating point result has the number truncated to its integer value (the part before the decimal point).

When using arrays, remember that if you DIM the array using a particular number of subscripts, each element of the array must be referenced with the same number of subscripts:

```
10 DIM name$(2,2,2)
20 name$(0) = "FRED"
```

produces an error. Line 20 should be replaced by:

```
20 name$(0,0,0) = "FRED"
```


In addition, the numbers used as subscripts must not be too big or less than zero:

```
10 DIM position(9,4)
20 position(-1,5) = 1
```

If you now type RUN, an error message is displayed because the first subscript must be between zero and nine and the second between zero and four.

When you DIM a string array, the elements are initialised, just as they are for numeric arrays. Each element in the array is set to the null string, "". No space is allocated for the characters of each string element until they are assigned a value.

The operators += and -= are particularly useful with arrays, as they remove the need to evaluate the subscript expressions twice. For example, suppose you had the assignment:

```
a(100*(SINRADangle+1))=a(100*(SINRADangle+1))+increment
```

The expression `100*(SINRADangle+1)` must be calculated twice, which could be quite time-consuming. On the other hand, if you used

```
a(100*(SINRADangle+1)) += increment
```

the complex subscript expression would only be used once, saving time.

Finding the size of an array

Functions are available to find the number of dimensions of an array, and the size of each dimension. To find the number of dimensions of an array type

```
PRINT DIM(A())
```

To find the number of elements of the *n*th dimension, type

```
PRINT DIM(A(),n)
```

For example,

```
10 DIM A(4,2,7)
20 n = DIM(A())
30 PRINT n
40 PRINT DIM(A(),n)
```

produces:

```
3
7
```

These functions are useful mainly in procedures and functions which take array parameters. See the chapter: **PROCEDURES AND FUNCTIONS** for more details.

Operating on whole arrays

As described above, every element of an array is given the value zero when the array is DIMmed. It is possible to set every element in an array to any given value using a single assignment as follows:

```
10 DIM A(10), B(10)
20 n% = 2
30 A() = (3*n%)
40 B() = A()
```

Line 10 dimensions two arrays of the same size. Line 30 sets all of the elements of A() to $3 \times n\%$, ie 6. Then line 40 sets all of the elements of B() from the corresponding elements in A().

In addition, all the elements in an array can be increased, decreased, multiplied or divided by a given amount:

```
10 DIM A(2,2), B(2,2)
20 A(0,0) = 4
30 A(1,1) = 5
```

```

40 A(2,2) = 6
50 n% = 2: m% = 3
60 A() = A() + (n%*n%)
70 A() = A() - m%
80 B() = A() * 6
90 B() = B() / n%

```

When you RUN this program, the elements of the arrays 'A' and 'B' are assigned the following values:

Array	Value	Array	Value	Array	Value
A(0,0)	5	A(0,1)	1	A(0,2)	1
A(1,0)	1	A(1,1)	6	A(1,2)	1
A(2,0)	1	A(2,1)	1	A(2,2)	7
B(0,0)	15	B(0,1)	3	B(0,2)	3
B(1,0)	3	B(1,1)	18	B(1,2)	3
B(2,0)	3	B(2,1)	3	B(2,2)	21

Note that in line 60 the brackets around $n\%*n\%$ are necessary. The amount being added, subtracted, and so on may be either a constant, a variable, a function result or an expression, provided that it is enclosed in brackets.

It is also possible to add, subtract, multiply or divide two arrays, provided that they are of the same size. In the result, every element is obtained by performing the specified operation on the two elements in the corresponding positions in the operands.

For example, for two arrays which have been dimmed $A(1,1)$ and $B(1,1)$, the instruction

```
A() = A() + B()
```

is equivalent to the following four instructions:

```

A(0,0) = A(0,0) + B(0,0)
A(0,1) = A(0,1) + B(0,1)
A(1,0) = A(1,0) + B(1,0)
A(1,1) = A(1,1) + B(1,1)

```

BASIC will perform proper matrix multiplication on pairs of two-dimensional arrays. The first index of the array is interpreted as the row and the second as the column. For example:

```

10 i=2:j=3:k=4
20 DIM A(i,j),B(j,k),C(i,k)
30 :
40 REM Set up the array contents...
50 :
60 C() = A().B()

```

Note that the first dimension of the array must be identical to the first dimension of the second array.

Also, the matrix multiplication operation can multiply a vector (a one-dimensional array) by a two dimensional matrix to yield a vector. There are two possible cases:

```
row().matrix()
```

This gives a row vector as the result. The number of elements is equal to the number of columns in the matrix.

```
matrix().column()
```

This gives a column vector as the result. The number of elements is equal to the number of rows in the matrix.

The first index of an array is interpreted as the row and the second as the column.

For example:

```

10 i = 2: j = 3
20 DIM row(i), column(j)
30 DIM matrix(i,j)
40:
50 REM lines to set up the arrays
200 column() = matrix().column()
220 PROCprint(column())
260 row() = row().matrix()
270 PROCprint(row())

```

Array operations

Arithmetic operations on arrays are not quite as general as those on simple numbers. Although you can say $a = b * b + c$, you cannot use the equivalent $a() = b() * b() + c()$. Instead, you would have to split it into two assignments:

```

a() = b() * b()
a() = a() + c()

```

Also, the only place these array operations may appear is on the right hand side of an assignment to another array. You cannot, for example, say `PRINT a() * 2`.

The table below gives a complete list of array operations.

<code><array> = <array></code>	Copy all elements
<code><array> = -<array></code>	Copy all elements, negating
<code><array> = <array> + <array></code>	Add corresponding elements
<code><array> = <array> - <array></code>	Subtract corresponding elements
<code><array> = <array> * <array></code>	Multiply corresponding elements
<code><array> = <array> / <array></code>	Divide corresponding elements
<code><array> = <factor></code>	Set all elements
<code><array> = <array> + <factor></code>	Increment all elements
<code><array> = <factor> + <array></code>	
<code><array> += <expression></code>	

`<array> = <array> - <factor>` Decrement all elements
`<array> = <factor> - <array>`
`<array> -= <expression>`

`<array> = <array> * <factor>` Multiply all elements
`<array> = <factor> * <array>`

`<array> = <array> / <factor>` Divide all elements
`<array> = <factor> / <array>`

`<array> = <array> . <array>` Matrix multiplication

`<array>` means any array variable. All of the operations on two arrays require arrays of exactly the same size and type (real and integer arrays are treated as different types for this purpose). Only the assignment and concatenational operations are available on string arrays.

`<factor>` means a simple expression, such as 1, LENA\$, or "HELLO". If you want to use an expression using binary operators, it must be enclosed in brackets: (a+b).

The arrays used in these operations may all be the same, or all be different, or somewhere in between. For example, you are allowed to use:

```

a() = b() + c()
a() = a() + c()
a() = a() + a()
  
```

The matrix multiplication operator works on two arrays which must be compatible in size. This means that in the assignment

```

a() = b() . c()
  
```

the following DIMs must have been used:

```
DIM b(i, j)
DIM c(j, k)
DIM a(i, k)
```

In addition, the following would be permitted:

```
DIM b(i, j)
DIM c(j)
DIM a(i)
```

or

```
DIM b(j)
DIM c(j, k)
DIM a(j)
```

OUTPUTTING TEXT

PRINT FORMATTING

The PRINT statement provides a number of ways of formatting the printed output.

Using print separators

The items in a PRINT statement can be separated by a variety of different punctuation characters. Each of these characters affects the way in which the text is formatted:

- Items separated by semicolons are printed one after the other, with no spaces.
- Items separated by commas are tabulated into columns.
- Items separated by apostrophes are printed on separate lines.

The following program demonstrates this:

```
10 PRINT "Hello " "Hello ", "Hello" "What's going on here then?"
```

Typing RUN produces the following output:

```
Hello Hello      Hello
What's going on here then?
```

Printing numbers

Numbers are printed in the same way as text:

```
10 A% = 4
20 PRINT 4;" ";A%
```


Typing RUN produces:

```
4 4
```

Numbers are normally printed (displayed) as decimal values unless they are preceded by a '~', in which case they are given in hexadecimal notation (hexadecimal numbers are discussed in the chapter: **BASES**):

```
10 PRINT 10
20 PRINT &10
30 PRINT ~10
40 PRINT ~&10
```

produces:

```
10
16
A
10
```

Defining fields

The columns controlled by commas are called fields. By default a field is ten characters wide. Each string which is printed following a comma starts at the left-hand side of the next field. In other words using commas is a convenient method of left-justifying text. Numbers, on the other hand, are displayed to the right of the next field, so that the units of integers, or the least significant decimal places of floating point numbers, line up. Thus,

```
10 FOR N% = 1 TO 5
20 A$ = LEFT$( "Hello", N%)
30 B% = N%*10^(N%-1)
40 PRINT A$, A$, A$, A$'B%, B%, B%, B%
50 NEXT N%
```

produces the following when RUN:

```

H          H          H          H
          1          1          1          1
He         He         He         He
          20         20         20         20
Hel        Hel        Hel        Hel
          300        300        300        300
Hell       Hell       Hell       Hell
          4000       4000       4000       4000
Hello      Hello      Hello      Hello
          50000      50000      50000      50000

```

Using @% to alter output

Problems may occur when printing out floating point numbers. For example:

```
PRINT 6,9,7/3,57
```

produces:

```

        6          92.33333333          57

```

The nine and the decimal equivalent of $7/3$ run into each other.

To prevent this, you can alter the field width or limit the number of decimal places printed (or both) by using the integer variable @%. To see the effect of altering the value of @%, type

```
@% = &20408
```

then

```
PRINT 6,9,7/3,57
```

and the following is produced:

```

6.0000 9.0000 2.3333 57.0000

```

The assignment of the variable @% is made up of a number of parts:

- & indicates that a hexadecimal number follows.
- The first number indicates the format of the print field, two means the computer prints a fixed number of decimal places.
- Zero and four indicate the number of decimal places required.
- Zero and eight give the field width.

The format: the first figure after the '&' symbol, can take three values:

- Zero is the default configuration; the computer uses the number of decimal places it requires up to a maximum of ten.
- One gives numbers in exponent form: a number between one and 9.99999999 followed by 'e' and then a power of ten.
- Two gives a number to a fixed number of decimal places giving up to a maximum of ten significant figures.

See PRINT in the chapter: SYNTAX AND USAGE OF BASIC KEYWORDS for more details on @%.

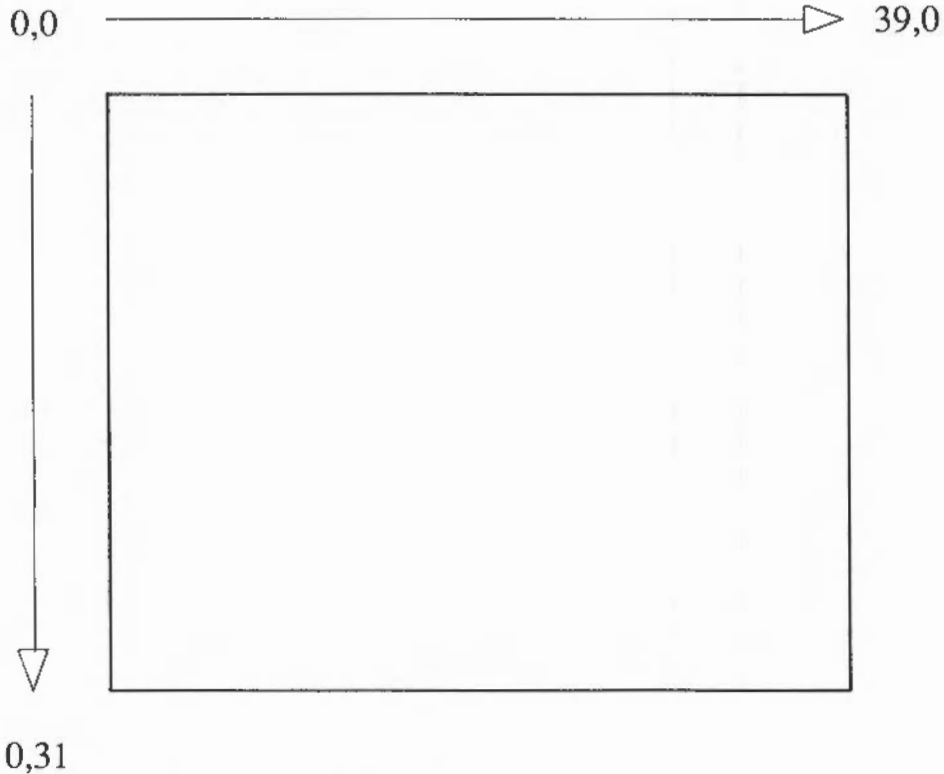
THE TEXT CURSOR

Text cursor coordinates

When text is entered at the keyboard or displayed using the PRINT statement, the position it appears at on the screen depends on the location of the text cursor. As each character is printed, this cursor moves on to the next character position.

Initially, the text cursor is at the top left-hand corner of the screen, which is position (0, 0). The number of possible positions for the cursor depends on the screen mode. For example, in MODE 1 which has 40 characters across the screen

and 32 rows, the coordinates it can have vary as follows:



Altering the position of the text cursor

You can use `TAB` with one parameter to control the position of the text cursor. For example:

```
PRINT TAB(x) "Hello"
```

It works as follows. If the current value of `COUNT` (which holds the number of characters printed since the last newline) is greater than the required tab column (ie 'x'), a newline is printed. This moves the cursor to the start of the next line, and

resets COUNT to zero. Then 'x' spaces are printed, moving the cursor to the required column.

Note that it is possible to tab to column 60 in a 40 column mode; the cursor will simply move to column 20 of the line below the current one. Using TAB with one parameter to position the cursor on the line will also work, for example, when characters are sent to the printer, as it is just printing spaces to achieve the desired tabulation.

On the other hand, TAB with two arguments works in a completely different way: it uses the Archimedes operating system to position the cursor at a specified position on the screen – this is relative to the screen 'home' position, which is normally the top left.

If you try to position the cursor on, say, column 60 in a 40 column mode, the command will be ignored. Furthermore, this kind of tabbing does not affect any characters being sent to the printer.

The VDU statement

In addition to TAB, there are other methods of altering the position of the cursor. If, for example, you type

```
10 PRINT "A";  
20 VDU 8  
30 PRINT "B"
```

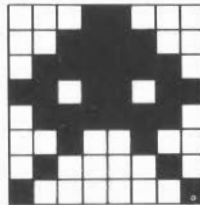
PRINT "A"; prints an 'A' at the current cursor position and moves the cursor one place to the right. VDU 8 moves the cursor back one position so that it is underneath the 'A'. Hence, PRINT "B" prints a 'B' at the same position as the 'A', and so rubs it out.

VDU 8	Moves the cursor back one space
VDU 9	Moves the cursor forwards one space
VDU 10	Moves the cursor down one line
VDU 11	Moves the cursor up one line
VDU 12	Clears the screen and puts the cursor at the top left
VDU 13	Moves the cursor to the beginning of the line

For detail of these and other effects available with VDU see the chapter: **VDU COMMANDS**.

DEFINING YOUR OWN CHARACTERS

Each character is made up of a pattern of dots on an eight by eight grid. All normal letters, numbers and so on are pre-defined in this way. It is possible, however, to define your own characters with ASCII values in the range 32 to 255. To do this, use the VDU 23 command, followed by the code of the character you wish to define and then eight integers, each representing one row of the character, from top to bottom. The bit pattern of each integer defines the sequence of dots and spaces: one gives a dot and zero gives a space.



To set up character 128 to be the shape shown above, use the following:

```
VDU 23,128,24,60,126,219,126,36,66,129
```

Then, to display this character, type

```
PRINT CHR$(128)
```


INPUTTING INFORMATION

INPUTTING DATA FROM THE KEYBOARD

INPUT

The INPUT statement allows a program to request information from the user. The following program gives an example:

```
10 PRINT "Give me a number and I'll double it";
20 INPUT X
30 PRINT "Twice ";X " is ";X*2
```

When you RUN this program, the INPUT command on line 20 displays a question mark on the screen and waits for you to enter data. The number you type is assigned to the variable 'X'. If you do not type anything or type letters or symbols instead, 'X' is assigned the value 0.

INPUT may also be used with string and integer variables:

```
10 PRINT "What is your name ";
20 INPUT A$
30 PRINT "Hello ";A$
```

Line 10 in each of the above two programs is used to print a message on the screen indicating the type of response required. The INPUT statement allows text prompts to be included, so the program above could be written more neatly as:

```
10 INPUT "What is your name ",A$
20 PRINT "Hello ";A$
```

The comma in line 10 tells the computer to print a question mark when it wants input from the keyboard. If you leave out the comma, the question mark is not printed. A semi-colon may be used, with exactly the same effect as the comma.

When the program is being executed, the INPUT statement requires you to press if you wish to send what you have typed to the computer. Until you press , you can delete all or part of what you have typed by pressing Delete .

When you are inputting a string, the computer ignores any leading spaces and anything after a comma, unless you put the whole string inside quotation marks.

To input a whole line of text, including commas and leading spaces, `INPUT LINE` may be used:

```
10 INPUT A$
20 INPUT LINE B$
30 PRINT A$
40 PRINT B$
```

RUN the above program and, in response to each of the question marks, type

Hello, how are you?

This produces the following output:

```
Hello
Hello, how are you?
```

Several inputs may be requested at one time:

```
10 INPUT A,B,C$
```

You may enter the data individually, pressing after each one. In this case you are prompted with a question mark until you enter the number required. Alternatively, you can give all the inputs on one line, separated by commas.


`GET` and `GET$`

`GET$` may be used to read a single key press:

```

10 PRINT "Press a key"
20 A$ = GET$
30 PRINT "The key you pressed was ";A$

```

In this example the program waits at line 20 until you press a key. As soon as you do so, the character that key represents is placed in A\$. You do not have to press  and so do not get the chance to change your mind.

GET is similar to GET\$ but returns the ASCII code of the key pressed, instead of the character.

INKEY and INKEY\$

INKEY\$ is similar to GET\$, except that it does not wait indefinitely for a key to be pressed. You give it a time limit and it waits for that length of time only. For example:

```

10 PRINT "You have 2 seconds to press a key"
20 A$ = INKEY$(200)

```

The number following the INKEY\$ is the number of hundredths of a second it waits. If a key is pressed in time, A\$ holds the character which was typed. Otherwise, A\$ is empty.

INKEY is used in a similar manner to INKEY\$: it waits for a given time for a key to be pressed, and then returns the ASCII code for the key pressed, or -1 if no key is pressed within this time.

INCLUDING DATA AS PART OF A PROGRAM

Predefined data may be included within a program and saved as part of it. When the program is RUN, individual items of data are read and assigned to variables as follows:

```

10 FOR I% = 1 TO 4
20 READ age%, dog$
30 PRINT "Name: ";dog$ " Age: ";age%

```

```
40 NEXT I%
50 DATA 9, "Laddie", 3, "Watson"
60 DATA 1
70 DATA "Mungo", 3, "Honey"
```

You may use as many DATA statements as you like, but you must make sure that the type of each item of data matches the type of the variable into which it is being read. Each DATA statement can be followed by one or more items of data separated by commas.

You can usually leave out the quotation marks around strings, but they *are* needed if you want to include spaces or commas in the string.

For example,

```
10 DATA Hello, my name is
20 DATA Rose
30 READ A$, B$
40 PRINT A$; B$
```

produces:

```
Hello my name is
Rose
```

To obtain the sentence Hello, my name is Rose, change the program as follows:

```
10 DATA "Hello, my name is"
20 DATA " Rose"
30 READ A$, B$
40 PRINT A$; B$
```

A DATA statement must appear as the first statement on a line, otherwise it will not be found. If the computer reaches a DATA statement while executing a program, it ignores it and goes on to the next line.

When it attempts to READ the first item of data, it scans through the lines of the program from the top until it finds the first DATA statement and uses the first item of data on this line. The next READ uses the second item and so on until the DATA statement has no more items left, at which point the next DATA statement is searched for and used.

If there is insufficient data, the computer produces an error message, such as:

```
Out of data at line 20
```

This indicates that it has tried to READ an item of data, but that all items have already been read.

In general, if you use line numbers anywhere in a program (and there should be very few situations where you have to), they should be simple numbers in the range 0 to 65279, not expressions. Otherwise, if the program is renumbered, it will stop working since BASIC does not know how to change the RESTORE expression in the right way.

BASIC CONTROL STATEMENTS

Normally, lines in a BASIC program are executed by the computer in sequence, one after the other. The language includes two types of structure which alter this sequence:

Conditional structures allow statements to be executed only if certain conditions are met.

Loop structures allow statements to be executed repeatedly, either for a fixed number of times, or until a certain condition is met.

In all cases, the code is easier to read if it is clear which statements are in the loop and which are conditional on certain factors. This clarity can be achieved by use of the LISTO command before listing the programs, to indent the conditional and loop structures in the listing. All programs included in this chapter are listed as if the command:

```
LISTO 3
```

had been typed beforehand; this gives a space after the line number and indents structures.

IF ... THEN ... ELSE

The IF statement may be used to enable the computer to make a choice about whether or not to execute a statement or group of statements. For example:

```
10 PRINT "What is 2 * 4"  
20 INPUT ans%  
30 IF ans% = 8 THEN PRINT "Well done" ELSE PRINT "No - you're wrong"
```

Line 30 contains a conditional expression. In the example shown the expression is TRUE (ie has a non-zero value) when ans% is equal to eight, and is FALSE (ie has a zero value) otherwise.

Three kinds of operators may be used in conditional expressions:

- numerical operators
- string operators
- logical operators.

The following table lists the operators and their meaning:

Numerical operators

Operators	Meaning
A = B	TRUE when A is equal to B
A < B	TRUE when A is less than B
A > B	TRUE when A is greater than B
A <= B	TRUE when A is less than or equal to B
A >= B	TRUE when A is greater than or equal to B
A <> B	TRUE when A is not equal to B

String operators

Operators	Meaning
A\$ = B\$	TRUE when A\$ and B\$ are the same
A\$ <> B\$	TRUE when A\$ and B\$ are different
A\$ < B\$	String comparisons; see below:
A\$ > B\$	
A\$ <= B\$	
A\$ >= B\$	

Corresponding characters of each string are examined until either they are different, or the end of a string is reached. If the strings are the same length, the

strings are said to be equal; otherwise, the shorter string is less than the longer one.

In the case where the two corresponding characters differ, the relationship between the strings is the same as that between the ASCII codes of the mismatched characters. For example, "HI" < "Hi", because the ASCII code of upper case I is less than that of lower case i.

Logical operators

Operators	Meaning
NOT A	TRUE when A is false
A AND B	TRUE if both A and B are true
A OR B	TRUE if either A or B or both are true
A EOR B	TRUE if either A or B but not both are true

If the result of the conditional statement `ans% = 8` is TRUE, the computer executes only the statement after the THEN. Otherwise, it executes only the statement after the ELSE.

Several statements can be included after an IF statement; they will all only be executed if the condition is true. For example:

```
IF ans% = 8 THEN PRINT "Well done" : answerscorrect% += 1
```

IF ... THEN ... ELSE ... ENDIF

A block structured IF ... THEN ... ELSE ... ENDIF statement is available. It executes a series of statements, which may be split over several lines, conditionally on the result of the IF expression.

```
10 n% = RND(10)
20 m% = RND(10)
```



```

30 PRINT "What is ";n% " * "m%;
40 INPUT ans%
50 IF ans% = n% *|Z m%|Z THEN
60   PRINT "Well done"
70 ELSE PRINT "No - you're wrong"
80   PRINT n%;" * ";m% " = ";n%*m%
90 ENDIF
100 RUN

```

The ENDIF on line 90 terminates the statement. It indicates that execution of the following statements is not dependent on the outcome of the conditional expression on line 50, so these statements are executed as normal. Without the ENDIF the computer has no way of knowing whether or not the statements on lines 80 and 100 belong to the ELSE part.

There are certain rules which must be obeyed when using IF ... THEN ... ELSE ... ENDIF constructions:

- The first line must take the form:

```
IF <conditional expression> THEN
```

THEN being the last item on the line.

- The ELSE need not be present, but if it is, it must be the first thing on line (excluding spaces).
- The ENDIF statement must be the first thing on a line (excluding spaces).

IF ... THEN ... ELSE ... ENDIF statements may be nested: one may occur inside another. For example:

```

10 DIM A%(10)
20 count% = 0
30 PRINT "Give me an integer between 0 and 9 ";
40 INPUT number%
50 IF number% >= 0 AND number% <= 9 THEN

```

```

60  IF A%(number%) = 0 THEN
70    PRINT "Thank you"
80    A%(number%) = 1 : count% = count% + 1
90    ELSE PRINT "You've already had that number"
100  ENDIF
110 ELSE PRINT number% " is not between 0 and 9 !"
120 ENDIF
130 IF count% < 10 GOTO 30

```

FOR ... NEXT

The FOR and NEXT statements are used to specify the number of times a block of a program is executed. These statements are placed so that they surround the block to be repeated:

```

10 FOR N% = 1 TO 6
20  PRINT N%
30 NEXT N%

```

Type RUN and the following is produced:

```

1
2
3
4
5
6

```

The variable N% is called the control variable. It is used to control the number of times the block of code is executed. The control variable can be started at any number you choose, and you may alter the step size: the amount by which it changes each time round the loop.

```

10 FOR N% = -5 TO 5 STEP 2
20  PRINT N%
30 NEXT N%

```

This program produces:

```
-5  
-3  
-1  
1  
3  
5
```

The step size can be negative so that the control variable is decreased each time. It does not have to be an integer value. You can also use a decimal step size, although this is not generally advisable. The reason is that numbers such as 0.1 are not exactly representable in the binary format used by the computer. This means that when the step is added to the looping variable several times, small errors may accumulate. You can see this by typing the program:

```
10 FOR i=0 TO 100 STEP 0.1  
20 PRINT i  
30 NEXT i
```

The looping variable *i* never quite reaches 100.

FOR ... NEXT loops may be nested. For example,

```
10 FOR N = 3.0 TO -1.0 STEP -2.0  
20 FOR M = 2.5 TO 2.9 STEP 0.2  
30 PRINT N,M  
40 NEXT M  
50 NEXT N
```

produces:

3	2.5
3	2.7
3	2.9
1	2.5
1	2.7
1	2.9
-1	2.5
-1	2.7
-1	2.9

You do not need to specify the control variable to which NEXT refers. The following program produces the same results as the one above:

```

10 FOR N = 3.0 TO -1.0 STEP -2.0
20   FOR M = 2.5 TO 2.9 STEP 0.2
30     PRINT N,M
40   NEXT
50 NEXT

```

The computer assumes that NEXT applies to the most recent FOR.

If you put variable names after NEXT but mix them up as shown,

```

10 FOR N = 3.0 TO -1.0 STEP -2.0
20   FOR M = 2.5 TO 2.9 STEP 0.2
30     PRINT N,M
40     NEXT N
50     NEXT M

```

the output produced is:

3.0	2.5
1.0	2.5
-1.0	2.5

Not in a FOR loop at line 50

Loops must be nested totally within each other: they must not cross. In the above example, the 'N' and 'M' loops are incorrectly nested. The computer tries to RUN the program, but when line 50 is reached, it gives an error message indicating that it cannot match the FOR statements with the NEXT statements.

The loop is ended when the control variable is:

- equal to or greater than the terminating value when a positive step size is used.
- equal to or less than the terminating value when a negative step size is used.

The loop is performed in the following sequence:

- Assign the initial value to the control variable.
- Execute the block of code.
- Increment the control variable by the step size.
- Test against terminating value, and if it is to be performed again, go back to 2.

One of the consequences of the way in which the loop is performed is that the block of code is always executed at least once. Thus,

```
10 FOR N = 6 TO 0
20 PRINT N
30 NEXT
```

produces:

6

FOR ... NEXT loops are very versatile, since the initial and terminating values and the step size can be assigned any arithmetic expression containing variables or functions. For example:

```

10 REM Draw a sine curve
20 MODE 0 : MOVE 0,512
30 PRINT "Please give me a step size "
40 INPUT step
50 FOR angle = -2*PI TO 2*PI STEP step
60   DRAW 100*angle, 100*SIN(angle)+512
70 NEXT
80 END

```

REPEAT ... UNTIL

The REPEAT ... UNTIL loop repeats a block of code until a given condition is fulfilled. For example:

```

10 REM Input a number in a given range
20 REPEAT
30   PRINT "Please give me a number between 0 and 9 "
40   INPUT N
50 UNTIL (N >= 0) AND (N <= 9)
60 PRINT "Thank You"

```

If the result of the conditional expression following the UNTIL is TRUE, then the loop is ended and the statement following the UNTIL is executed. If, however, the result of the expression is FALSE, the block of code after the REPEAT is executed again and the conditional expression is re-evaluated.

REPEAT ... UNTIL loops may be nested in the same way as FOR ... NEXT loops. They are also similar to FOR loops in that the body of the loop is always done once, since no test is done until the end of the loop is reached.

```

10 REM Repeat questions until answered right first time
20 REPEAT
30   tries% = 0
40   REPEAT
50     PRINT "What is 20 * 23 + 14 * 11 ";
60     INPUT ans%
70     tries% += 1

```

```

80  UNTIL ans% = 20 * 23 + 14 * 11
90  REPEAT
100  PRINT "What is 12 + 23 * 14 + 6 / 3 ";
110  INPUT ans%
120  tries% += 1
130  UNTIL ans% = 12 + 23 * 14 + 6 / 3
140 UNTIL tries% = 2;

```

WHILE ... ENDWHILE

The WHILE ... ENDWHILE loop repeats a block of code while a given condition holds true. For example:

```

10 X = 0
20 WHILE X < 100
30  PRINT X
40  X += X + RND(5)
50 ENDWHILE

```

The WHILE ... ENDWHILE loop has a conditional expression at the start of it. If this expression returns TRUE, the block of statements following the WHILE, down to the ENDWHILE statement, is executed. This is repeated until the expression returns FALSE, in which case execution jumps to the statement following the ENDWHILE.

WHILE ... ENDWHILE is very similar to REPEAT ... UNTIL except that the conditional expression is evaluated at the beginning of the loop and the loop is executed again only if the result is TRUE. The following program demonstrates the fact that REPEAT ... UNTIL loops are always executed at least once, whereas the WHILE ... ENDWHILE loops need not be executed at all.

```

10 REPEAT
20  PRINT "Repeat"
30 UNTIL TRUE
40
50 WHILE FALSE
60  PRINT "While"

```

```

70 ENDWHILE
80
90 PRINT "All done"

```

This program produces the following output:

```

Repeat
All done

```

CASE ... OF ... WHEN ... OTHERWISE ... ENDCASE

The IF ... THEN ... ELSE ... ENDIF construct is useful if you wish to make a choice between two alternatives. The CASE statement can be used when there are many alternatives to be acted upon in different ways.

The following program is a keyboard controlled sketch pad. The statements after the WHENs alter the values of X% and Y%, and then DRAW a line.

```

10 REM Draw a line depending on the L,R,U,D keys
20 MODE 0
30 MOVE 640,512
40 X% = 640: Y% = 512
50 REPEAT
60 CASE GET$ OF
70   WHEN "L","l": X% -= 40: DRAW X%,Y% : REM go left
80   WHEN "R","r": X% += 40: DRAW X%,Y% : REM go right
90   WHEN "D","d": Y% -= 40: DRAW X%,Y% : REM go down
100  WHEN "U","u": Y% += 40: DRAW X%,Y% : REM go up
110 ENDCASE
120 UNTIL FALSE : REM go on forever ...

```

This program reads in the character of the next key pressed and checks it against each of the strings following the WHEN statements. If it matches one of these values, the statements following WHEN are executed. The computer then finds the ENDCASE and continues executing from there.

If you press a key which is not recognised by any of the four WHEN statements, the program goes round again and waits for another key to be pressed. You can include another line to warn you that you pressed the wrong key. For example:

```
105  OTHERWISE VDU 7 : REM Make a short noise
```

The OTHERWISE statement is used if none of the WHENs finds a matching key. The VDU 7 makes a short bell sound to warn you that you have pressed the wrong key.

The following rules apply to CASE statements:

- CASE must be followed by an expression and then OF. This statement must be at the end of the line.
- Each WHEN must start at the beginning of a line. It may be followed by one or more values, separated by commas.
- The statements dependent on a WHEN may follow it on the same line after a colon ':', or be spread over several lines following it.
- OTHERWISE is optional. If present it must be at the beginning of a line.
- An ENDCASE statement must be present. Like WHEN and OTHERWISE, it must be the first non-space item on a line.

Whenever the result of the expression matches one of the values listed after a WHEN, all the statements following this WHEN down to the next WHEN or OTHERWISE or ENDCASE are executed. The computer then finds the statement following the ENDCASE. This means that if the result matches a value in more than one list, only the statements following the first one are executed: the others are ignored. Since OTHERWISE matches any value, having WHEN statements following an OTHERWISE is pointless since they can never be reached.

The following gives another demonstration of its use:

```
10 REM Guess a number  
20 X% = RND(100)
```

```

30 Still_guessing% = TRUE
40 tries% = 0
50 WHILE Still_guessing%
60   INPUT "What is your guess ",guess%
70   CASE guess% OF
80     WHEN X%:
90       PRINT "Well done, you've guessed it after ";tries% " attempts"
100      Still_guessing% = FALSE
110      WHEN X%-1,X%+1: PRINT "Very close": tries% = tries% + 1
120      OTHERWISE
130        IF guess%<X% THEN PRINT "Too low" ELSE PRINT "Too high"
140        tries% = tries% + 1
150      ENDCASE
160 ENDWHILE

```

Like all the other BASIC structures, CASE statements may be nested.

GOTO

The GOTO instruction may be used to specify a line number from which the computer is to continue executing the program. For example:

```

10 PRINT "Hello"
20 GOTO 10

```

Whenever the computer executes line 20 it is sent back to line 10 once again. Left on its own, this program never ends. To stop it, press **Escape**

GOTO instructions send the control of the program either forwards or backwards. The specified line number may be given as an expression. For example:

```

10 start% = 100
20 GOTO (start%+10)
30 PRINT "This line should not be executed"
100 REM start of the action
110 PRINT "Hello"
120 END

```

Using a variable, however, as the destination for a GOTO is not recommended because while RENUMBER changes the line numbers, it does not alter GOTO destinations that are given as anything other than a constant. If you must use an expression, it is best to put in inside brackets, since BASIC may get confused if the expression starts with a number.

If you wish to make your programs easy to read, especially for other people, use as few GOTOs as possible because they make a program very difficult to follow. It is far better to use one of the loop constructs like REPEAT ... UNTIL which have been described above.

GOSUB ... RETURN

GOSUB stands for 'go to subroutine' and is another variation of GOTO. Instead of continuing indefinitely from the line number which is jumped to, the lines are executed until a RETURN statement is reached. The computer then jumps back to the instruction which comes after the GOSUB. For example,

```
10 GOSUB 100
20 PRINT "This is printed after the first GOSUB returns"
30 GOSUB 100
40 PRINT "This is printed after the second GOSUB returns"
50 END
100 PRINT "This is printed in the GOSUB"
110 RETURN
```

produces:

```
This is printed in the GOSUB
This is printed after the first GOSUB returns
This is printed in the GOSUB
This is printed after the second GOSUB returns
```

Like GOTO, GOSUB should be used sparingly. Better methods of providing blocks of code, which once executed then return control back to the point from which they were called are described in the chapter: PROCEDURES AND FUNCTIONS.

ON ... GOTO/GOSUB

The ON ... GOTO statement is used to choose one of a number of different lines depending on the value of a given expression. For example:

```

10 PRINT "Input a number between 1 and 4"
20 INPUT N%
30 ON N% GOTO 60, 100, 80, 120
60 PRINT "Your number is 1"
70 GOTO 999
80 PRINT "Your number is 3"
90 GOTO 999
100 PRINT "Your number is 2"
110 GOTO 999
120 PRINT "Your number is 4"
999 END

```

The computer checks the value of N% which is input, then jumps to the N%th line number in the list. If N% is three, the computer starts executing at line 80 and so on. If N% is less than one or greater than four, the error message ON range at line 30 is displayed.

ELSE is used to catch all other values. It is followed by a statement which is executed if the value of the expression has no corresponding line number. For example, line 30 above could be replaced by:

```

30 ON N% GOTO 60,100,80,120 ELSE PRINT "Number out of range"
40 GOTO 999

```

Now, when the program is run, if N% is not between 1 and 4 the message Number out of range is displayed and the program ends normally.

ON ... GOSUB acts in exactly the same way:

```
10 PRINT "Input a number between 1 and 4"
20 INPUT N%
30 ON N% GOSUB 60, 100, 80, 120
40 END
60 PRINT "Your number is 1"
70 RETURN
80 PRINT "Your number is 3"
90 RETURN
100 PRINT "Your number is 2"
110 RETURN
120 PRINT "Your number is 4"
130 RETURN
```

Note, however, that when writing new programs, it is better to use CASE structures rather than the ON ... GOTO ... GOSUB constructs.

PROCEDURES AND FUNCTIONS

Procedures and functions provide a way of structuring a program by grouping statements together and referring to them by a single name. The statements can be executed from elsewhere in the program simply by specifying the procedure or function name. In addition, a function returns a value.

DEFINING AND CALLING PROCEDURES

Procedure names begin with the keyword `PROC`, followed by a name. The following shows how a procedure may be defined and called:

```
10 MODE 12
20 PRINT TAB(0,10)"Countdown commencing ";
30 FOR N% = 30 TO 1 STEP -1
40   PRINT TAB(22,10) " " TAB(22,10);N%;
50   PROCwait_1_second
60 NEXT
70 PRINT TAB(0,10) "BLAST OFF";STRING$(14," ")
80 END
90
100 DEF PROCwait_1_second
110 TIME = 0
120 REPEAT
130 UNTIL TIME >= 100
140 ENDPROC
```

The important points about procedures are:

- The procedure definition must start with DEF PROC followed by the procedure name.
- The procedure definition must end with the keyword ENDPROC.
- Procedures are called by the keyword PROC followed by the procedure name.
- Procedure names obey the same rules as variable names, except that they are allowed to start with a digit. Procedure names can also include or start with reserved words such as PROCTO.
- The main body of the program must be separated from the procedure definitions by an END statement.

Procedures enable you to split up a large amount of code into smaller distinct sections which are easy to manage. The main body of a program can then consist almost entirely of procedure calls, so that it can remain short and easy to follow (since it should be obvious from the procedure names what each call is doing).

PARAMETERS AND LOCAL VARIABLES

Consider the following program:

```

10 REM Draw boxes centred on the screen
20 MODE 12
30 FOR N% = 1 TO 10
40 PRINT "What size do you want the next box to be ";
50 INPUT size
60 IF size < 1024 THEN PROCbox(size) ELSE PRINT "Too large"
70 NEXT
80 END
100 DEF PROCbox(edge)
110 RECTANGLE 640-edge/2, 512-edge/2, edge, edge
120 ENDPROC

```

The procedure PROCbox draws a box around the centre of the screen. The size of this box is determined by the value of the variable edge. This variable has the

current value of `size` assigned to it each time the procedure is called from line 60. The values being passed to the procedure are known as actual parameters. The variable `edge` used within the procedure is known as a formal parameter.

A procedure can be defined with more than one parameter. However, it must always be called with the correct number of parameters. These parameters may be integers, floating point numbers, strings or arrays. If a string variable is used as a formal parameter, it must have either a string or a string variable passed to it. Floating point and integer parameters may be passed to one another and interchanged freely, but remember that the fractional part of a floating point variable is lost if it is assigned to an integer variable.

The formal parameters of a procedure are local to that procedure. This means that assigning a value to any variable within the procedure does not affect any variable elsewhere in the program which has the same name. In the following program, the procedure `PROCsquare` has a parameter `S%` which is automatically local. It also contains a variable, `J%`, which is declared as being `LOCAL`.

```
10 FOR I% = 1 TO 10
20   PROCsquare(I%)
30   PROCcube(I%)
40 NEXT
50 END
60
100 DEF PROCsquare(S%)
110 LOCAL J%
120 J% = S% ^ 2
130 PRINT S% " squared equals "J%;
140 ENDPROC
150
200 DEF PROCcube(I%)
210 I% = I% ^ 3
220 PRINT " and cubed equals ";I%
230 ENDPROC
```


In the case of PROCcube, the actual parameter passed and the formal parameter referred to within it are both called I%. This means that there are two versions of the variable, one inside the procedure and another outside it. Adding the line

```
35 PRINT I%
```

to the program above prints out the numbers 1 to 10, showing that the assignment to I% within PROCcube does not affect the value of I% in the main body of the program.

It is good practice to declare all variables used in a procedure as LOCAL, since this removes the risk that the procedure will alter variables used elsewhere in the program.

When declaring a local array, the LOCAL statement must be followed by a DIM statement to dimension the local array. For example, consider the following function which, when passed two vectors of the same size, returns their scalar product:

```
100 DEF FNscalar_product(A(),B())
110 REM ** Both arrays must have a dimension of 1 **
120 IF DIM(A()) <> 1 OR DIM(B()) <> 1 THEN
130 PRINT "Vectors required"
140 =0
150 ENDIF
160 REM ** Both arrays must be the same size **
170 IF DIM(A(),1) <> DIM(B(),1) THEN
180 PRINT "Vectors must be of same size"
190 =0
200 ENDIF
210 REM ** Create a temporary array of the same size **
220 LOCAL C()
230 DIM C(DIM(A(),1))
240 REM ** Multiply the corresponding elements and place in C() **
250 C() = A()*B()
260 REM ** Finally sum all the elements of C() **
270 =SUM(C())
```

This example uses a function instead of a procedure. The two structures are very similar, but they are used in slightly different circumstances. PROCs are used wherever a statement can be executed. FNs are used in expressions, wherever a built-in function might be used. Whereas procedures end with an `ENDPROC` statement, functions return using `=<expression>`. The expression is returned as the result of the function call. Note that `SUM` is a built-in function.

Value-result parameter passing

The simple parameter passing scheme described above is known as value parameter passing because the value of the actual parameter is copied into the formal parameter, which is then used within the procedure. The result of any modification to the formal parameter is not communicated back to the actual parameter. Thus the formal parameter is entirely local.

BASIC provides a second method of parameter passing known as value-result. This is just like the simple value mechanism in that the actual parameter's value is copied into the formal parameter for use inside the procedure. The difference is, however, that when the procedure returns the final value of the formal parameter is copied back into the actual parameter. Thus, a result can be passed back.

A statement specifying that you wish to pass a result back for a particular parameter should be preceded by the keyword `RETURN`. For example:

```
100 DEF PROCorderedswap (RETURN A, RETURN B)
110 IF A > B SWAP A, B
120 ENDPROC
```

ON ... PROC

`ON ... PROC` is similar to `ON ... GOTO` which is described in the chapter: **BASIC CONTROL STATEMENTS**. It evaluates the expression given after the `ON` keyword. If the value `N%` is given, it then calls the procedure designated by `N%` on the list. For example:

```
10 REPEAT
20 INPUT "Enter a number ", num
```

```

30 PRINT "Type 1 to double it"
40 PRINT "Type 2 to square it"
50 INPUT action
60 ON action PROCdouble(num), PROCsquare(num)
70 UNTIL FALSE
100 DEF PROCdouble(num)
110 PRINT "Your number doubled is ";num*2
120 ENDPROC
200 DEF PROCsquare(num)
210 PRINT "Your number squared is ";num*num
220 ENDPROC

```

Note, however, that in most circumstances, the CASE statement provides a more powerful and structured way of performing these actions.

RECURSIVE PROCEDURES

A procedure may contain calls to other procedures and may even contain a call to itself. A procedure which does call itself from within its own definition is called a recursive procedure:

```

10 PRINT "Please input a string : "
20 INPUT A$
30 PROCremove_spaces(A$)
40 END
100 DEF PROCremove_spaces(A$)
110 LOCAL pos_space%
120 PRINT A$
130 pos_space%=INSTR(A$, " ")
140 IF pos_space%=0 THEN ENDPROC
150 A$=LEFT$(A$, pos_space%-1)+RIGHT$(A$, pos_space%+1)
160 PROCremove_spaces(A$)
170 ENDPROC

```

In the example above, PROCremove_spaces is passed a string as a parameter. If the string contains no spaces, the procedure ends. If a space is found within the string, the space is removed and the procedure is called again with the new string

as an argument to remove any further spaces. For example, inputting the string
The quick brown fox causes the following to be displayed:

```
The quick brown fox
Thequick brown fox
Thequickbrown fox
Thequickbrownfox
```

Recursive procedures often provide a very clear solution to a problem. There are
two reasons, however, which suggest that they may not be the best way to solve a
problem:

- Some operations are more naturally expressed as a loop, that is, using
FOR ... NEXT, REPEAT ... UNTIL, or WHILE ... ENDWHILE.
- Recursive procedures often use more of the computer's memory than the
corresponding loop.

As an example, the following two programs both print "Good morning !" backwards.
The first one uses a WHILE ... ENDWHILE loop. The second uses a recursive technique
to achieve the same result.

First example:

```
10 PROCreverseprint("Good morning !")
20 END
100 DEF PROCreverseprint(A$)
120 WHILE LEN(A$) > 0
130   PRINT RIGHT$(A$);
140   A$=LEFT$(A$)
150 ENDWHILE
160 ENDPROC
```

Second example:

```
10 PROCreverseprint ("Good morning !")
20 END
100 DEF PROCreverseprint (A$)
110 IF LEN(A$) > 0 THEN
120   PRINT RIGHT$(A$);
130   PROCreverseprint (LEFT$(A$))
140 ENDIF
160 ENDPROC
```

FUNCTIONS

Functions are similar to procedures, but differ in that they return a result. BASIC provides many functions of its own, like the trigonometric functions SIN, COS, TAN and RND. If you give RND a parameter with an integer value greater than 1, it returns a random value between 1 and the number given inclusive. For example,

```
X = RND(10)
```

produces random numbers between 1 and 10.

You may define functions of your own using the keyword DEF followed by FN and the name of your function. The function definition ends when a statement beginning with an '=' sign is encountered. This assigns the expression on the right of the sign to the function result. This result may be assigned to a variable in the normal way.

Functions obey the same rules with regards to naming conventions, the use of parameters and local variables. Procedures also follow these rules.

The following is an example of how a function may be defined and used:

```
10 FOR N% = 1 TO 10
20 PRINT "A sphere of radius ";N%;" has a volume ";FNvolume(N%)
30 NEXT
40 END
100 DEF FNvolume(radius%)
110 = 4/3*PI*radius%^3
```

FUNCTION AND PROCEDURE LIBRARIES

Libraries provide a convenient way of adding frequently-used procedures and functions to a BASIC program.

The libraries are kept in memory, and if a reference is made to a procedure or function which is not defined in your program, a search of each library in turn is made until a definition is found. If the routine is found in a library, it is executed exactly as though it were part of the program.

The advantages of using libraries are:

- They standardise certain routines between programs.
- They reduce the time required to write and test a program. (The library routines only need to be written and tested once, not each time a new program is developed.)
- They make programs shorter and more modular.

Loading a library into memory

There are two methods of loading a library into memory: `INSTALL` and `LIBRARY`.

`INSTALL` loads the library at the top of memory, then lowers `HIMEM` and BASIC's stack down by an appropriate amount. Any number of libraries can be installed, provided there is enough memory for them. Since `INSTALL` affects the stack, it cannot be used whenever there is anything on the stack. This means that it

cannot be used inside procedures or loops. Installed libraries are only removed when you exit from BASIC.

LIBRARY reserves a sufficient area of memory for the library just above the main BASIC program and loads the library. Any library loaded in this way remains only until the heap is cleared. This occurs, for example, when the CLEAR or NEW commands are given, or when a program is RUN.

For example:

```
10 MODE 1
20 REM Print out a story
30 REM Load output library
40 LIBRARY "Printout"
50 REM Read and print the heading
60 READ A$
70 PROCcentre(A$)
80 REM Print out each sentence in turn
90 REPEAT
100  READ sentence$
110  REM if sentence$ = "0" then have reached the end
120  IF sentence$ = "0" END
130  REM otherwise print it out
140  PROCprettyprint(sentence$)
150 UNTIL FALSE
200 DATA A story
210 DATA This,program,is,using,two,procedures:
220 DATA 'centre',and,'prettyprint',from,a,library
230 DATA called,'Printout'.
240 DATA The,library,is,loaded,each,time,
245 DATA the,program,is,run.
250 DATA The,procedure,'centre',places,a,string,in,the
260 DATA centre,of,the,screen.
270 DATA The,procedure,'prettyprint',prints,out,
280 DATA a,word,at,the,current,text,cursor,
290 DATA position,unless,it,would,be,spilt,over,
300 DATA a,line,in,which,case,it,starts,the,word,
```

```
305 DATA on,the,next,line,down.
310 DATA 0
```

The library Printout could be as follows:

```
10 REM Printout-Text output library-see PROCPrintouthelp
20 REM *****
30 DEF PROCPrintouthelp
40 REM Print out details of the library routines
50 PRINT "PROCcentre(a$)"
60 PRINT "Place a string in the centre";
70 PRINT "PRINT "of a 40 character line""
80 PRINT "PROCprettyprint(a$)"
90 PRINT "Print out a word at the current";
100 PRINT "text cursor position, starting";
110 PRINT "a new 40 character line if required";
120 PRINT "to avoid splitting it over two lines";
130 ENDPROC
140 REM *****
200 REM Place a string in the centre
210 REM of a 40 character line
220 DEF PROCcentre(a$)
230 LOCAL start%
240 start% = (40 - LEN(a$))/2
250 PRINT TAB(start%);a$
260 ENDPROC
270 REM *****
300 REM Print out a word at the current
310 REM text cursor position, starting
320 REM a new 40 character line if required
330 REM to avoid splitting it over two lines
340 DEF PROCprettyprint(a$)
350 LOCAL end%
360 end% = POS + LEN(a$)
370 IF end% < 40 PRINT a$;" " ; : ENDPROC
380 PRINT 'a$;" " ;
```



```
390 ENDPROC
400 REM *****
```

Building your own libraries

There are certain rules which should be obeyed when writing library procedures and functions:

- Line number references are not allowed.

Libraries must not use `GOTO`, `RESTORE`, etc. Any reference to a line number is to be taken as referring to the current program, not to the line numbers with which the library is constructed.

- Only local variables should be used.

It is advisable that library routines only use local variables, so that they are totally independent of any program which may call them.

- Each library should have a heading.

It is recommended that a library's first line contains the full name of the library and details of a procedure which prints out information on each of the routines in the library.

This last rule applies because `BASIC` contains a command, `LVAR`, listing the first line of all libraries which are currently loaded. As a result, it is important that the first line of each library contains all the essential information about itself.

The Advanced Disc Filing System (ADFS) provides facilities for saving, organising, and accessing data held on floppy and hard discs. Another filing system, The Advanced Network Filing System (ANFS) is supplied for use with Econet file servers. The Advanced Disc Filing System and the Advanced Network Filing System are very similar in operation and most of the details in this chapter, which concentrate on the Advanced Disc Filing System, apply equally to the Advanced Network Filing System. Any important differences between the two systems are mentioned where appropriate.

SELECTING THE ADFS

You may select the Advanced Disc Filing System at any stage by giving the command:

```
*ADFS
```

If you are using a different filing system, such as the Advanced Net Filing System, when you issue the command, the filing system you are using is closed down and the Advanced Disc Filing System is selected.

If you are already using the Advanced Disc Filing System when you issue this command, the system is reselected.

LEAVING THE ADFS

You may leave the Advanced Disc Filing System by issuing the command to enter a different one. For example, to select the Advanced Net Filing System, type

```
*NET
```

When using a hard disc system, end each session by typing

```
*BYE
```

This moves the heads of the hard drive to a transit position. It prevents the heads or disc surface from being damaged if the hard drive is moved or accidentally knocked.

SELECTING THE ANFS

You may select the Advanced Network Filing System at any stage by typing

*NET

If you are using a different filing system, such as the Advanced Disc Filing System, when you issue this command, the filing system you are using is closed down and the Advanced Network Filing System is selected.

If you are already using the Advanced Network Filing System when you type *NET, the system is reselected.

If you intend to use the network file server to load and save files, you need to introduce yourself to the file server by typing

*I AM <Name>

or the equivalent

*LOGON <Name>

where <Name> is the user name that has been issued to you by the manager of the network system.

LEAVING THE ANFS

You may leave the Advanced Network Filing System by issuing the command to enter a different filing system. For example, to select the Advanced Disc Filing System, type

*ADFS

When you have finished using the network file server, end each session by typing

*BYE

This finishes your session on the file server and closes down all your open files and directories.

DRIVE NUMBERS

Many operations performed by the Advanced Disc Filing System require you to identify the drive containing the disc you wish to use. Disc drives are identified by numbers. The Archimedes disc drive numbers are as follows:

Drives 0 - 3	Floppy disc drives
Drives 4 - 7	Hard disc drives

It is because the software can support a maximum of four floppy disc drives and four hard disc drives that this scheme allows each drive to be uniquely identified.

The single floppy disc drive supplied with your computer is numbered '0' while additional second, third and fourth floppy disc drives are numbered consecutively '1', '2' and '3'. A single hard disc drive is numbered '4' while additional second, third and fourth hard disc drives are numbered consecutively '5', '6' and '7'.

You tell the computer which drive you wish to use by including the drive number when you issue a command. Examples are given in the discussions below.

DISC NAMES

When using the ANFS, you will rarely have to refer to the name of the disc you are using; most operations will use the default, which is the name of the disc on which your user directory resides.

When using the ADFS, you can refer to a disc either by the drive number in which it resides, or by the name of the disc itself. In the latter case, you should ensure that the disc is actually in one of the drives or you will get a `Disc not present` error.

To use the disc name when referring to a file, you prefix the name with a colon and separate it from the filename using a period. For example, to load a BASIC program called 'test' from a disc called 'Petel', you would use:

```
LOAD ":Petel.test"
```

FORMATTING A DISC

Formatting discs is a fundamental function performed by the ADFS. Formatting lays down the magnetic tracks on the disc and divides each of these tracks into sectors. Each sector is given header information which prepares it for receiving data. When using the Advanced Network Filing System, you should not need to format discs for yourself as this will be done by the manager of the network. Details of formatting and backing up discs for the file server are given in the relevant file server manager's guide.

Before you format a disc, note the following points:

- You can format blank discs or reformat previously formatted discs using the same procedures. However, formatting a disc destroys any information currently stored on the disc. Be very careful about formatting any disc already containing programs. Always make sure you no longer wish to keep them or that you have copies of these programs on other discs.
- The disc you are formatting should not be write-protected.
- You can format a disc to hold either 640 or 800 Kbytes of information. Discs formatted to hold 640 Kbytes are compatible with the BBC Master-Series Microcomputers.

To format a floppy disc in drive 0 (to hold 800K of data):

- Place the disc in the disc drive.
- Type the following:

```
*ADFS
```

Press . The cursor drops down a line next to a new screen prompt.

- Type the following:

```
*FORMAT 0 D
```

Press . The following prompt is displayed:

```
Are you sure (Y/N) ?
```

- Press 'N' if you change your mind about formatting this disc. The screen prompt appears.
- Press 'Y' to format the disc. The following message appears on the screen:

```
Formatting xx
```

The xx begins with zero and increases until 79 is reached. 79 is significant because the disc has 80 tracks. When 79 appears, the following message is displayed:

```
Verifying
```

The Archimedes takes a few seconds to verify that the disc is formatted correctly. If there is a fault in the disc, an error message is displayed and verification stops. If no fault appears, the following message is displayed:

```
Formatted 800k
```

The screen prompt appears and the format is done.

If you wish to format a disc to hold 640 Kbytes of information, follow the steps above with one exception. At step 3, instead of typing *FORMAT 0 D, type *FORMAT 0 L.

If no errors occur during verification, the message,

```
Formatted 640k
```

is displayed before the screen prompt appears. The disc is now formatted to hold 640 Kbytes of data.

If a verification error occurs, repeat the process. If this is still unsuccessful, it probably means that your disc is faulty. The safest thing to do is to destroy it and start again with a different disc.

BACKING UP DISCS

A very important facility of the ADFS is the opportunity it provides for making backup copies of the information on floppy discs.

Floppy discs, like all other equipment, can develop faults. If this happens then it could mean that you are unable to load one or more programs from that disc, resulting in lost information and time. It is a good idea, therefore, to keep more than one copy of important programs.

You can copy information from one floppy disc to another, from one hard disc to another, from a floppy disc to a hard disc, or from a hard disc to several floppy discs.

- *Note:* when you make a backup copy of a disc, copying all the information from the first disc to the second, any data already on the second disc is deleted and replaced by an exact copy of the information on the first disc.

COPYING USING A SINGLE FLOPPY DISC DRIVE

To copy all the programs from one floppy disc to another floppy disc using a single disc drive, take the following steps:

At the '>' prompt, type

```
*BACKUP 0 0
```

The following message is displayed:

```
Are you sure (Y/N) ?
```

Press 'Y' to proceed with the backup, any other key to abort.

```
Insert source disc in drive 0 then press SPACE bar
```

Carry out this request. You will then see:

```
Insert destination disc in drive 0 then press SPACE bar
```

This cycle repeats until the whole disc has been backed up. You can make the process quicker by adding the letter Q onto the end of the command. This tells the backup program to use as much memory as possible to perform the backup, so it requires fewer disc swaps. It is important, however, to realise that this option will overwrite any BASIC program you may have loaded.

COPYING USING MORE THAN ONE FLOPPY DISC DRIVE

To copy programs from one floppy disc drive to another floppy disc drive, follow the instructions below (your source drive is 0 and your destination drive is 1):

- At the '>' prompt, type

```
*BACKUP 0 1
```

The following message is displayed:

```
Insert source disc
```

- Place the disc containing the programs to be copied into drive 0.

The following message is displayed:

```
Insert destination disc
```


- Place the disc onto which you wish to copy the programs into drive 1. All data on the disc in drive 0 is copied onto drive 1. When the backup is finished, the '>' prompt is displayed.

DIRECTORIES

Every time you save data or write and save a program, a file is produced. In order to make it possible to organise files so that you can find them easily, the Advanced Disc Filing System and the Advanced Network Filing System group them together into units called directories.

When a floppy disc is formatted, one directory is automatically created. This is known as the root directory and is identified by the '\$' symbol. When the ADFS is first selected it normally reads the contents of the root directory into memory. If you save a file without changing the directory, the file is saved in the root directory.

The Advanced Network Filing System also uses a root directory identified by the '\$' symbol. When, however, you use the Advanced Network Filing System to access the file server, all users are provided with their own directory in which to store files. This is known as their user root directory and is identified by a name which is the same as their user name. If you save a file using the Advanced Network Filing System without changing directory, the file is saved in your user root directory.

To work with a file you must give it a filename. A filename can contain up to ten characters. These characters can be either letters or numbers. Other characters may be used. This is not recommended, however, as the ADFS attaches a special meaning to several characters.

There are two special symbols, called wildcards, which you can use to refer to a group of files or directories in one command. These symbols are the '#' (hash sign) and the '*' (asterisk).

- The '#' is used to denote any single character. Thus,

AB#

can refer to ABC and ABZ , but not AB or ABCD.

- The '*' is used to denote any sequence of characters. Thus,

dir1.W*

refers to all the files and directories in the directory dir1 which begin with 'W', including a file or directory called 'W'.

You can create new directories and subdirectories. The rules that apply to naming files also apply to naming directories and subdirectories. Directories and subdirectories may, however, be referred to by special symbols; the '\$' symbol, for example, defines the root directory. The special symbols are:

- \$ The root directory.
- & The user root directory. This is a directory you designate as your own root directory using the *URD command.
- @ The current directory. This is the directory you are currently in.
- ^ The parent directory. This is the directory that is directly above the current directory in the hierarchy.
- \ The previous directory. Using this symbol refers you to the directory you were previously in.
- % The library. This is the directory that contains utility programs used frequently. Library is discussed in the chapter: **FILING SYSTEMS** in the Reference guide.

These symbols can be used in place of directory names. Examples are included in the discussions below.

The '^' character in pathnames may not be valid on some Econet systems. (It depends on the fileserver software.) This applies also to the *UP command.

Note also that strictly speaking '^' should be interpreted as 'the directory above where we are so far in this pathname'. So, assuming we have a directory called `dir` in the current directory, the pathname `@.dir.^fred` just refers to `@.fred`: we went down a level, then up again. Similarly, `@.^.^` refers to a directory two levels above the current one.

As mentioned above, the directory with which you are actively working is called the current directory. How to create a new directory is discussed below.

- *Note:* the following discussion on saving files and creating directories assumes you are in BASIC, have typed the *ADFS command and are beginning with the root directory as your current directory.

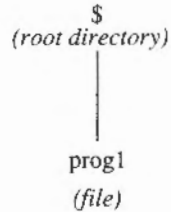
The number of files and subdirectories a directory can contain depends on how the disc you are using is formatted. If you format the disc to hold 800Kbytes of information its directories can contain up to 77 files and subdirectories each. If you format the disc to hold 640Kbytes of information its directories can contain up to 47 files and subdirectories each.

Saving a file

To save a BASIC program file with the name `prog1` type

```
SAVE "prog1"
```

On disc, a file structure is created that can be shown in a diagram as follows:

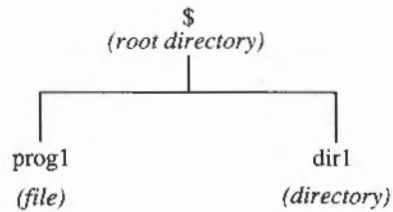


Creating a directory

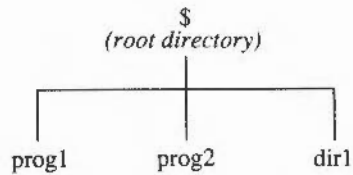
To create a directory with the name `dir1`, type

```
*CDIR dir1
```

The resulting file structure can now be shown in a diagram as follows:



If you save a new file, say a file named `prog2`, the file structure is as follows:



Creating a new directory does not alter the current directory. As a result, the root directory is still the current directory, so the file `prog2` is saved there.

Changing directories

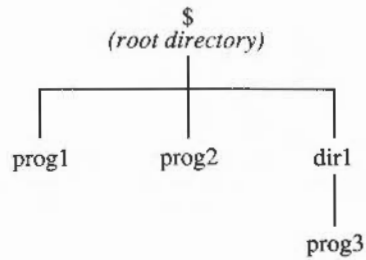
To change directories, making `dir1` your current directory, type

```
*DIR dir1
```

Now when a file is saved, it is saved in `dir1`. For example, to save a file named `prog3` type

```
SAVE "prog3"
```

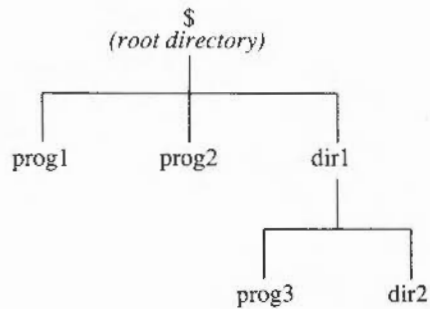
The resulting file structure is as follows:



Similarly, if you create a new directory (in the following example a directory named `dir2`) this also is created within `dir1`:

```
*CDIR dir2
```

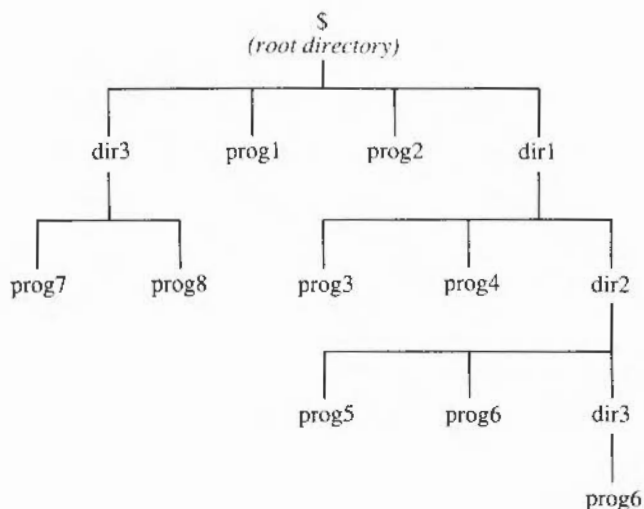
The resulting file structure appears as follows:



PATHNAMES

Files and directories within the current directory can be referenced using a single name. It is possible, however, to reference any file at any time from any position within the file structure by giving a pathname. A pathname contains a sequence of directory names separated by a '.' character which shows the file or directory's location within the hierarchical structure.

The file structure shown below is used in the examples throughout the remainder of this chapter.



To load the file `prog5` from the current directory, `dir1`, type either

```
LOAD "$.dir1.dir2.prog5"
```

or

```
LOAD "dir2.prog5"
```

Both of these are pathnames. The pathname in the first example starts from the root directory. The pathname in the second example starts from the current directory.

Notice that in the diagram above there are two files with the name `prog6`. Their pathnames, however, are:

```
$.dir1.dir2.prog6
```

and

```
$.dir1.dir2.dir3.prog6
```

Since their pathnames are different, each can be uniquely identified.

The other directory symbols can be used to load a file from anywhere in the hierarchy. For example, to load `prog8` when the current directory is `dir1`, type

```
LOAD "^ .dir3.prog8"
```

DELETING FILES AND DIRECTORIES

Files can be deleted using the `*DELETE` command.

When deleting a file, you can either specify the full pathname, or you can make the directory containing the file you wish to delete the current directory and just specify the filename.

For example, to delete `prog5` when `dir2` is the current directory, type

```
*DELETE prog5
```

To delete `prog5` when the root directory is the current directory, type

```
*DELETE $.dir1.dir2.prog5
```


When you create a file using a command such as `SAVE`, it is not locked by default: to delete it you do not have to unlock it using an `*ACCESS` command. On the other hand, directories are locked by default, and have to be unlocked (and empty) before they can be deleted.

A directory can also only be deleted if it is empty; each of the files and subdirectories within it must therefore be deleted first. For example, to unlock and delete directory `dir2` when the root directory is the current directory, type

<code>*DELETE \$.dir1.dir2.dir3.prog6</code>	This deletes <code>prog6</code> in <code>dir3</code> .
<code>*ACCESS \$.dir1.dir2.dir3</code>	This unlocks <code>dir3</code>
<code>*DELETE \$.dir1.dir2.dir3</code>	This deletes <code>dir3</code>
<code>*DELETE \$.dir1.dir2.prog5</code>	This deletes <code>prog5</code>
<code>*DELETE \$.dir1.dir2.prog6</code>	This deletes <code>prog6</code>
<code>*ACCESS \$.dir1.dir2</code>	This unlocks <code>dir2</code>
<code>*DELETE \$.dir1.dir2</code>	This deletes <code>dir2</code>

COPYING AND MOVING FILES

Using the `*COPY` command

When you are using the `COPY` command note the following:

- Wildcards can be used to refer to groups of files or directories in one command.
- Files may be copied between hard and floppy discs by including the appropriate drive number.
- Files are added to the catalogue for the specified directory and therefore `Dir full` or `Disc full` error messages could occur.

To copy one or more files from one directory to another the `*COPY` command may be used. If you type

```
*COPY $.dir1.dir2.prog# $.dir1.prog#
```

the files `prog5` and `prog6` in directory `$.dir1.dir2` are copied to the directory `$.dir1`.

You can also use the directory symbols to copy a file from one directory to another. For example, to copy the file `$.dir1.prog3` to the directory `$.dir1.dir2` type

```
*DIR $.dir1.dir2
```

This makes `dir2` the current directory. Now type

```
*COPY ^.prog3 @.prog3
```

This copies the file `prog3` in `dir1` to the current directory, `dir2`.

Using the *RENAME command

Before using the *RENAME command, note the following:

- If you do not specify a new directory when giving the pathname, the computer assumes you are moving a file or directory within the current directory.
- You cannot use the *RENAME command to move a locked file or directory.
- You cannot use the *RENAME command to move a file or directory to a named file or directory which already exists.
- If you are moving a directory, all the files and subordinate directories within it remain unchanged but are accessible only by their new pathname.

You can move files using the *RENAME command if a different pathname is given as the second parameter. For example,

```
*RENAME prog5 $.prog8
```

moves `$.dir1.dir2.prog5` to directory `'$'` and changes its name to `prog8`.

FILE DETAILS AND ATTRIBUTES

The *CAT command: displaying files and directories

You can display the files and subdirectories in any given directory. For example, if you type

```
*CAT $.dir1.dir2
```

a list of the files and directories in `dir2` is displayed on the screen.

If no pathname is given, the files and directories in the current directory are displayed.

The information given in a screen display listing files and directories includes one or more lines of general information, followed by the filenames listed in columns in alphabetical order. Next to each filename is a sequence of characters. For example a listing of the files and directories in `dir2` includes:

```
DeskTop      Disc "No Name" :0   Option 03 (Exec) URD"Unset"  
Dir. DeskTop Lib. "Unset"
```

```
dir3  DL      prog5  WR  prog6  WR
```

The letters following the filenames show what attributes, or access restrictions, the files and directories have:

- L The file or directory is locked so it may not be deleted or overwritten
- W The file may be written to and updated
- R The file may be read from and loaded
- D The object is a directory

The default attributes for a file are `WR`. Hence the file can be both written to and read from. Those for a directory are `DL`, so that it cannot be deleted.

When using the Advanced Network Filing System, two additional attributes or access restrictions can be associated with each file. These attributes are used to control how other users of the file server may use your files. The access which others may have to files is separated from your own access attributes by a '/' symbol. For example, an attribute description of WR/R specifies that the file may be both written to and read from by you, and that other users may only read from the file.

The *ACCESS command: changing file attributes

The attributes (other than D) for a file or directory can be altered using the command *ACCESS. For example,

```
*ACCESS prog4 R
```

alters the file prog4 so that it is read only.

The *INFO command: displaying file size

Further information about the size of a file can be obtained by entering the *INFO command. If you type

```
*INFO prog1
```

the following information is displayed:

```
prog1    WR    FFFFFFFB40  1C65E33A  00000155  00202600
```

The first three sets of numbers are in hexadecimal (hex) notation. The chapter: BASES discusses hexadecimal numbers.

Each column of data is defined as follows:

```
prog1          This is the name of the file.
```

```
WR             These are the file attributes.
```

FFFFFB40 This gives the file type and part of the date stamp.
1C65E33A This is the remainder of the date stamp.
00000155 This is the size of the file in bytes.
00202600 This is the disc address where the file starts.

The wildcard symbols '#' and '*' can also be included in a *INFO command. For example:

```
*INFO $.dir1.dir2.prog#
```

The *INFO and *EX commands will convert the load and execution addresses of a file into a type name and date stamp if the file has been date stamped correctly. The example below shows typical *INFO information for a date stamped and unstamped file:

DSFile	WR BASIC	17:37:26	17-Jun-1987	00004387	00001100
UnDSFile	WR	00123456	00123456	00002310	00015400

The *EX command: displaying file information

Information about all the files in a directory can be obtained using the *EX command. For example, when '\$' is the current directory, typing

```
*EX $.dir1.dir2
```

displays information about all the files in dir2.

The *OPT command: setting file system options

The *OPT command determines what filing system information is displayed during load, save and at auto-start when a disc is in the disc drive.

The following options are possible:

*OPT 0	Restores default settings
*OPT 1 0	File information suppressed during load and save
*OPT 1 1	Some file information to be displayed
*OPT 1 2	Full file information to be displayed
*OPT 4 0	Disable the auto-start facility
*OPT 4 1	*RMLoad the !BOOT file on auto-boot
*OPT 4 2	*RUN the !BOOT file
*OPT 4 3	*EXEC the !BOOT file

When using the Advanced Network Filing System, the auto-start option differs in the following ways:

- The file that is used by the auto-start system is called !ARMBOOT instead of !BOOT.
- The auto-start option is remembered by the file server and only comes into effect when you log on to the file server by typing

*I AM <name>

The auto-start option determines what happens when **Shift Break** is pressed with a disc in the disc drive. It can either have no effect, or it can access a file called !BOOT in directory '\$'. This file can be a machine code program, in which case it is either loaded into memory (*LOAD) or executed (*RUN), or it can be an ordinary text file.

If the file is an ordinary text file created using the *BUILD command (see the section below: **Command files**), the *EXEC option may be used.

The auto-start option is usually set to *EXEC on commercially available software. This enables the user to press **Shift Break**, so causing the !BOOT file to execute the commands needed to load the appropriate files into memory and start the program.

The action of `[Break]` and `[Shift][Break]` may be reversed using the `*CONFIGURE BOOT` command. When this is issued, and `[Ctrl][Break]` pressed to bring it into effect, subsequent resets will cause the auto-boot action to occur. Holding down `[Shift]` during a reset suppresses the auto-boot option.

You can restore the auto-boot action to the normal state by issuing the command `*CONFIGURE NOBOOT` and pressing `[Ctrl][Break]`.

The `*FREE`, `*MAP`, and `*COMPACT` commands: displaying disc space

To see how much free space is left on the disc in drive 0, type

```
*FREE 0
```

This produces the following output:

```
Bytes Free    &00097E00 = 622080
Bytes Used    &00008200 = 33280
```

`*FREE`, however, displays the total amount of free space on the disc. It does not give any information as to how this free space is distributed on the disc. The space might be all in one large block, or it might be divided into small sections across the disc. (Fragmentation of the free space occurs when files are frequently deleted or overwritten.)

On the other hand, when using the Advanced Network Filing System, `*FREE` displays information about the total free space on the file server discs, and also information about the free space available to you as a user. This user free space can be set by the manager of the file server.

It is possible to find that, although there is sufficient space in total on the disc to save a particular file, there is no single section of free space large enough to hold it. If this occurs, the error message `Compaction required` is displayed. You should then type

```
*COMPACT 0
```

This moves files around on the disc, collecting all the free space into a continuous block.

The message `Compaction recommended` may also occur. The ADFS keeps a map of all the segments of free space. When the map is approximately three-quarters full, the `Compaction recommended` message is given, indicating that it is advisable to compact the disc at once to prevent the map from filling up completely, making it impossible for you to complete a command.

The amount of free space applies to the disc containing the current directory unless the `*MAP` command is followed by a different disc name, in which case that disc's free space is displayed.

Entries printed by the `*MAP` command have the format `(start, length)`, where `start` is the start address of the free area on the disc, and `length` is its size. Both are in bytes.

DATA FILES

Programs can create and read information from files, called data files. For example, if you write a program that creates a list of names and telephone numbers, you may wish to save the names and telephone numbers as a data file.

The data file is specified in a program by one of the `OPEN` keywords.

You can create a data file using the keyword `OPENOUT`. For example, typing

```
A = OPENOUT "books"
```

creates a data file named `books` and opens it so that it is ready to receive data. The variable 'A' is called a channel number and allows the computer to distinguish this data file from other data files. All future communication with the file `books` is made via the file channel number in 'A' rather than via the name of the file.

Writing information to a data file is done using `PRINT#`. For example:


```

10 A = OPENOUT "books"
20 FOR I = 1 TO 5
30 READ book$
40 PRINT# A, book$
50 NEXT I
60 CLOSE# A
70 END
80 DATA "Black Beauty"
90 DATA "Lord of the Rings"
100 DATA "The Wind in the Willows"
110 DATA "The House at Pooh Corner"
120 DATA "Little Women"

```

Closing a data file is done using CLOSE#.

You can read data from a file using OPENIN and INPUT#. OPENIN opens an existing data file so that information may be read from it. INPUT# then reads the individual items of data. For example:

```

10 channel = OPENIN "books"
20 REPEAT
30   INPUT# channel, title$
40 UNTIL EOF# channel
50 CLOSE# channel
60 END

```

EOF# is a logical operator which is TRUE when the end of a file is reached.

Other useful keywords for reading or writing data are:

- BPUT# which writes a single byte to a file
- BGET# which reads a single byte from a file.

The following writes all the upper-case characters to a file using BPUT# as part of the program:

```

10 channel = OPENOUT "characters"
20 FOR N% = ASC("A") TO ASC("Z")
30   BPUT# channel, N%
40 NEXT
50 CLOSE# channel

```

BGET# is used as part of a program that allows each character to be read into a string as follows:

```

10 channel = OPENIN "characters"
20 string$ = ""
30 REPEAT
40   string$ += CHR$(BGET# channel)
50 UNTIL EOF# channel
60 CLOSE# channel

```

The BPUT# statement and GET\$# function can also be used to write text to a file, and read text from a file. These write and read the text in a form compatible with other programs, such as text editors, unlike PRINT# and INPUT# which write and read strings in BASIC string format.

When you PRINT# an expression to a file, it is written as an encoded sequence of bytes. For example, an integer is stored on the file as the byte &40 followed by the binary representation of the number. A string is written as &00 followed by the length of the string, followed by the string itself in reverse order.

To write information as pure text, you can use:

```
BPUT#channel, "string"
```

The characters of the string, which may be any string expression, are written to the file. If there is no semi-colon at the end of the statement (as in the example above), then a newline character (ASCII 10) is written after the last character of the string. If the semi-colon is present, no newline is appended to the string.

To read an ASCII string from a file, you can use:

```
string$=GET$#channel
```

This function reads characters from the file until a newline (ASCII 10), carriage return (ASCII 13), or nul (ASCII 0) character is read. This terminates the string, but is not returned as part of it.

COMMAND FILES

As you use the computer, you may find yourself typing the same sequence of commands over and over to perform frequently used tasks. With the ADFS you can put the command sequence into a special file called a command file, and then execute the entire sequence by typing the filename along with the appropriate ADFS command (*BUILD or *SPOOL). The commands in the command file are processed as if they were typed at the keyboard.

There are two types of command file:

- The first type contains commands which would be entered using standard keyboard characters. They are sent directly to the file from the keyboard.
- The second type contains VDU control codes and characters.

To set up a command file from the keyboard, use the *BUILD command. For example, if you type

```
*BUILD keyfile
```

everything subsequently typed from the keyboard is sent directly to the file called `keyfile`. If there is a file named `keyfile` already, it is deleted when the command is entered.

When you finish entering the commands, press `Escape` to end keyboard input to `keyfile`.

To execute the commands in `keyfile`, type

```
*EXEC keyfile
```

The contents of `keyfile` are read and executed a character at a time as if they were being typed at the keyboard.

Another way to make a command file directly from the keyboard is to use the `*SPOOL` command. If you type

```
*SPOOL vdufile
```

a new file called `vdufile` is created. If there is already a file named `vdufile`, it is deleted when the command is entered.

All subsequent VDU output is sent to this file until `*SPOOL` is used again, either with a different name, in which case the current file is closed and spooling continues with the new file, or no name, in which case the current file is closed and VDU data is directed to the screen as normal.

*CONFIGURE OPTIONS

If you are using a disc frequently, you can set the computer to default automatically to the ADFS whenever you turn the machine on. This is done by typing

```
*CONFIGURE FILE 8
```

The '8' indicates that you are configuring the computer for ADFS.

- *Note:* if you want to configure the Archimedes for ANFS, type `*CONFIGURE 5`. The '5' indicates that you are using ANFS.

You may also need to set other configuration parameters. What these parameters are, and how to set them are described below (the discussion assumes you have typed `*CONFIGURE FILE 8`):

- *Indicating a drive number*

If you are using the Archimedes with one floppy disc drive (drive 0), when you start the computer the ADFS automatically accesses that drive.

If you are using more than one disc drive with the Archimedes (a hard or floppy drive) you can tell the ADFS to go automatically to one of these drives.

To set a drive, type the following:

```
*CONFIGURE DRIVE n
```

'n' is the number of the drive which the ADFS is to select.

Once you configure the computer to default to a specific drive using *CONFIGURE DRIVE n, you can tell the Archimedes to look for data in another drive by typing the following:

```
*DRIVE n
```

This temporarily redirects the computer to the drive indicated until you do one of three things:

- choose another drive using *DRIVE n
- switch the computer off
- press the reset button

– *Indicating the number of floppy disc drives*

If you are using the Archimedes with more than one floppy disc drive, you need to tell the computer another drive is attached. To do so, type

```
*CONFIGURE FLOPPIES n
```

'n' is the number of floppy disc drives you are using.

– *Indicating the number of hard disc drives*

If you are using the Archimedes with a hard disc drive as well as a floppy disc drive, you need to tell the computer the hard drive is attached. To do so, type

*CONFIGURE HARDDISCS n

'n' is the number of hard disc drives you are using.

In addition to the *CONFIGURE options mentioned in this section, there are: *CONFIGURE BOOT/NOBOOT, DIR/NODIR, and STEP. The first option has already been described. The DIR/NODIR option controls whether the FS will access the disc drive on reset. If you type

*CONFIGURE DIR

and press **Ctrl Break**, the subsequent resets will cause the ADFS to load in the root directory from the configured drive. On the other hand, if the NODIR option is configured, the disc will not be accessed on reset and the current directory will be "Unset".

The STEP configuration allows you to set the track-to-track stepping time for any single (floppy) drive or all of the drives in the system. It has the form:

*CONFIGURE STEP <delay> [<drive>]

The <delay> value is 0, 1, 2 or 3. This corresponds to a step time of 6, 12, 2 and 3ms respectively. The CMOS RAM default value is 3, which is suitable for the built-in drive. If you give the second parameter, then the step speed is set for just that drive. If you omit it, the step times for all drives are set.

SCREEN MODES

THE MODES AVAILABLE

The display produced on a standard monitor can be in any of 21 different modes. These are referred to by numbers from zero to 20. Each mode gives a different combination of values to the following four items:

- the number of characters you can display on the screen
- the graphics resolution
- the number of colours available on the screen at any one time
- the amount of memory allocated to the screen display.

For example, mode 0 allows 32 rows of text to be displayed, each containing up to 80 characters. It provides high resolution graphics, but allows just two colours to be displayed on the screen. In contrast, mode 1 can display just 40 characters on a row and provides medium resolution graphics; it supports, however, up to four colours. Different modes use different amounts of memory to hold the picture; the amount of memory is determined by the resolution and by the number of colours. Mode 0, for example, requires 20K.

The mode which uses the most memory is the one with the highest resolution and the highest number of colours available. This is mode 15 which combines the highest resolution with the facility to use all 256 colours at once. The last three modes, numbered 18 to 20, require a special 'multisync' monitor. They provide double the usual vertical resolution: 512 lines instead of 256. The horizontal resolution is 640 pixels, and the numbers of colours available are 2, 4 and 16 respectively.

Changing mode

To change mode, type `MODE` followed by the mode number you want. For example,

```
MODE 12
```


changes the display to mode 12. This is one of the most useful modes since it provides high resolution graphics in 16 colours.

Shadow modes

In addition to mode numbers 0 to 20, you can use 128 to 148. These modes use the so-called 'shadow' memory. If you imagine that there are two separate areas of memory which may be used to hold the screen information, then selecting a normal mode will cause one area to be used, and selecting a shadow mode (in the range 128 to 148) will cause the alternative bank to be used.

You can force all subsequent mode changes to use the shadow bank by issuing the command:

```
*SHADOW
```

After this, you can imagine 128 to be added to any mode number in the range 0 to 20. To disable the automatic use of the shadow memory, issue the command:

```
*SHADOW 1
```

In order to use the shadow bank, the `ScreenSize` configuration must reserve at least twice as much screen memory as the amount required for the shadow mode. For example, if you want to use both mode 0 and mode 128, 40K of screen memory must be available, as mode 0 takes 20K.

In fact, for a given mode, there may be several banks available. You can work out how many by dividing the amount of configured screen memory by the requirement of the current mode. On the Archimedes 305, for example, 80K is reserved for the screen by default. This means that you can have four banks of mode 0.

The normal, non-shadow bank is numbered bank 1, and the shadow bank, used by mode 128, is bank 2. There are two more, banks 3 and 4. Using operating system calls, you can choose which of the four banks is displayed, and which is used by the VDU drivers when displaying text and graphics.

A full description of the available modes is given in APPENDIX E.

TEXT SIZE

The number of characters displayed on the screen is affected by the number which are allowed per row (ie the width of each character) and the number of rows which can be displayed on the screen (ie the spacing between the rows). The permitted number of characters per row is either 20, 40, 80 or 132. There are two possibilities for the number of rows on the screen, either 25 or 32. The former is particularly useful for text displays since the larger separation between the rows makes the text easier to read.

GRAPHICS RESOLUTION

The graphics resolution is specified by the number of pixels (rectangular dots) which can be displayed horizontally and vertically. The greater the number of pixels which the screen can be divided into the smaller each pixel is. Since all lines have to be at least one pixel thick, smaller pixels enable the lines to appear less chunky. To see the difference the pixel size makes try typing the following in BASIC:

```
10 MODE 2
20 MOVE 100,100
30 DRAW 100,924
40 MOVE 100,100
50 DRAW 1180,100
60 MOVE 100,100
70 DRAW 1180,924
```

and then:

```
10 MODE 0
20 MOVE 100,100
30 DRAW 100,924
40 MOVE 100,100
50 DRAW 1180,100
```

```
60 MOVE 100,100
70 DRAW 1180,924
```

COLOURS

The number of colours available on the screen at any given time is either two, four, 16 or 256. When you first turn your computer on and enter a particular mode, the computer selects the default colours which it uses for that particular mode. These are assigned to colour numbers:

Two-colour mode

0 = black
1 = white

Four-colour modes

0 = black
1 = red
2 = yellow
3 = white

16-colour modes

- 0 = black
- 1 = red
- 2 = green
- 3 = yellow
- 4 = blue
- 5 = magenta
- 6 = cyan
- 7 = white
- 8 = flashing black-white
- 9 = flashing red-cyan
- 10 = flashing green-magenta
- 11 = flashing yellow-blue
- 12 = flashing blue-yellow
- 13 = flashing magenta-green
- 14 = flashing cyan-red
- 15 = flashing white-black

256-colour modes

These modes are explained fully in the section: **Control of the palette in 256-colour modes.**

The computer chooses one colour to display text and graphics and another for the background. These two colours are chosen so that under default conditions the text and graphics are in white and the background is black. For example, in four-colour modes the computer chooses to draw text and graphics in colour three (white) on a background which is colour zero (black).

You may, however, choose to display your text, graphics, or background in a different colour. To do this, use the following commands:

- COLOUR n selects colour 'n' for text
- GCOL n selects colour 'n' for graphics

Each command can affect both the foreground and background colours, depending on the value it is given:

- If 'n' is less than 128, the foreground colour is set to colour 'n'.
- If 'n' is 128 or greater, the background colour is set to colour 'n' - 128.

If the colour number is greater than the number of colours available in a particular mode then it is reduced to lie within the range available. For example, in a four-colour mode COLOUR 1 is equivalent to COLOUR 5 and COLOUR 9, and so on.

Try the following example:

```
10 MODE 1
20 COLOUR 129
30 COLOUR 2
40 PRINT "Hello There"
```

Using the colour palette

Besides being able to select the colour in which numbers, text and so on are displayed, you can also change the physical colour associated with each colour number.

At the simplest level, you can reassign the colour numbers to produce a different one in the standard set of eight steady and eight flashing colours. This requires the command:

```
COLOUR n, m
```

where 'n' is the colour number (often called the logical colour) to be assigned to, and 'm' is the physical colour (the colour which you actually see) you wish to assign to it. For example:

```

10 MODE 1
20 COLOUR 0,4 : REM make colour number 0 appear as blue
30 COLOUR 128 : REM choose this as the background
40 COLOUR 1,3 : REM make colour number one appear as yellow
50 COLOUR 1 : REM choose this for the foreground text
60 PRINT "Yellow on Blue"

```

Alternatively, you can define the amount (as one of 16 levels) of red, green, and blue which go to make up the colour displayed for each of the logical colour numbers. Thus, any of the 16 colour numbers can be made to appear as a shade selected from the full range, or 'palette', of $16*16*16 = 4096$ colours.

To assign any of the shades available to a logical colour use the following command:

```
COLOUR n,r,g,b
```

This assigns 'r' parts red, 'g' parts green and 'b' parts blue to logical colour 'n'. Each of 'r', 'g' and 'b' must be values between zero and 255. A value of zero specifies that none of that colour should be used and a value of 255 that the maximum intensity of that colour should be used. Thus setting all of them to zero gives black and setting all to 255 gives white.

The following program allows you to try out various combinations and to see each displayed:

```

10 REPEAT
20  MODE 1
30:
40  REM Input values from the user
50:
60  INPUT"Amount of red   (0 - 15) "red%
70  INPUT"Amount of green (0 - 15) "green%
80  INPUT"Amount of blue  (0 - 15) "blue%
90:
100 REM Force the numbers into the range required
110:

```

```

120  red%  = red%  << 4
130  green% = green% << 4
140  blue%  = blue%  << 4
150:
160  COLOUR 0,red%,green%,blue%
170  GCOL 0
180  RECTANGLE FILL 540,412,200,200
190:
200  Now=TIME
250  REPEAT UNTIL TIME > Now + 500
260:
270 UNTIL FALSE : REM Repeat forever

```

This program asks you for three values, one for each of the amounts of red, green and blue you require. It then plots a rectangle in that colour. After it has displayed it for five seconds it clears the screen and starts again. To stop the program at any stage press `Escape` .

To return to the default settings for each of the colours type

```
VDU 20
```

- *Note:* the current hardware only supports 16 levels for each colour component numbered 0, 16, 32 ... up to 240. Intermediate numbers will give the next lowest level.

Also note that full control is not available over the colour palette setting in 256-colour modes. See the section below for details.

256-colour modes

In these modes, a choice of 64 colours is available directly from the simple COLOUR and GCOL commands. For example:

```

10 MODE 15
20 FOR Col% = 0 TO 63
30   COLOUR Col%
40   PRINT ":";Col%;
50 NEXT

```

As in the other modes the colour of the background can be changed by adding 128 to the parameter of the COLOUR command. Try modifying line 30 of the above program and running it again.

To understand the manner in which the colour number dictates the actual shade of colour which you see you need to consider the binary pattern which makes up the number. Only the bottom six bits are relevant. For an explanation of '%' and binary numbers, see the chapter: BASES.

In common with the other modes colour zero (%000000) is black.

```

1 (%000001) is dark-red
2 (%000010) is mid-red
3 (%000011) is bright-red

4 (%000100) is dark-green
8 (%001000) is mid-green
12 (%001100) is bright-green

16 (%010000) is dark-blue
32 (%100000) is mid-blue
48 (%110000) is bright-blue

63 (%111111) is white

```

Of the six bits which are used for the colour, the bottom two control the amount of red, the middle two the amount of green and the top two the amount of blue.

For example, COLOUR 35 is composed as follows:

35 = %100011 and so contains two parts of blue, no green and three parts of red, and appears as a purple shade. The remaining two bits of the eight bits of colour information cannot be provided via COLOUR because the very top bit has already been used to signify foreground or background colour. They are, therefore, supplied via a special TINT keyword.

The range of the TINT value is 0 to 255; but there are only four distinct tint levels within this range, and so the following ranges all have the same effect:

0-63	No extra brightness
64-127	Some extra brightness
128-191	More extra brightness
192-255	Maximum extra brightness

For example:

```
COLOUR 35 TINT 128
```

or

```
GCOL 17 TINT 0
```

The effect of the TINT is to change the small amount of white tint which is used in conjunction with the base colour. This gives four subtle variations to each colour.

The following program demonstrates how to get 256 different shades on the screen at the same time just using COLOUR and TINT:

```
10 MODE 15
20 FOR Col% = 0 TO 63
30   FOR tint% = 0 TO 192 STEP 64
40     COLOUR 128 + Col% TINT tint%
50     PRINT " ";
60   NEXT
70 NEXT
```

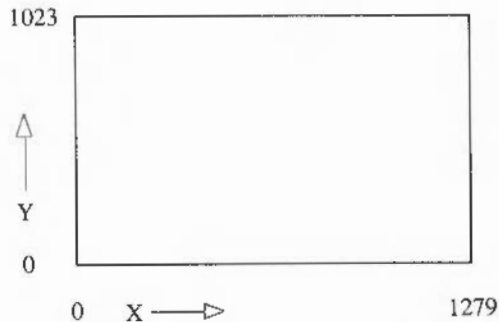
Control of the palette in 256-colour modes

This is only possible with an understanding of the VIDC video controller chip and is beyond the scope of this guide. For further information see the Reference guide.

The default settings for the palette have been very carefully chosen to enable the use of COLOUR and TINT as defined above. If the settings of the palette are changed, these simple relationships no longer apply.

THE GRAPHICS SCREEN

Whatever mode is being used, the graphics screen always has the same coordinate system:



Points outside these values may be used. For example, a line may be drawn between $(-144,-350)$ and $(1060,1200)$. What appears on the screen in this case is the portion of the line which crosses the region $(0,0)$ to $(1279,1023)$.

THE LINE COMMAND

BASIC provides a very simple way of drawing lines on the screen. All you need to do is to work out the positions of the two ends of the line. You can then draw a line with a single instruction such as:

```
LINE 120,120,840,920
```

You could draw the line the other way and produce the same result:

```
LINE 840,920,120,120
```

The following program uses `LINE` four times to draw a box on the screen.

```

10 MODE 0
20 left% = 100
30 right% = 400
40 bottom% = 200
50 top% = 800
60:
70 LINE left%,bottom%,right%,bottom%
80 LINE left%,top%,right%,top%
90 LINE left%,bottom%,left%,top%
100 LINE right%,bottom%,right%,top%

```

RECTANGLE AND RECTANGLE FILL

The `RECTANGLE` commands provide an easier way of drawing boxes on the screen. The first two parameters of `RECTANGLE` are the 'x' and 'y' coordinates of one of the corners. The second two parameters are the width and height of the rectangle. For example:

```
RECTANGLE 440,412,400,200
```

If the width and height are equal, ie for a square, the fourth parameter may be omitted:

```
RECTANGLE 400,312,400
```

`RECTANGLE FILL` is used in exactly the same way as `RECTANGLE`, but instead of drawing the outline of a rectangle, it produces a solid rectangle. The following program plots solid squares of gradually decreasing size in different colours:

```

10 MODE 15
20 FOR I% = 63 TO 1 STEP -1
30 GCOL I%
40 RECTANGLE FILL 640-I%*8,512-I%*8,I%*16
50 NEXT

```

CIRCLE AND CIRCLE FILL

To draw the outline of a circle or to plot a solid circle, you need to provide the centre of the circle and the radius.

For example:

```
CIRCLE 640,512,100
CIRCLE FILL 640,512,50
```

This produces the outline of a circle centred at (640,512), which is the centre of the screen, and of radius 100. Inside this is a solid circle, again centred at (640,512), which has a radius of 50.

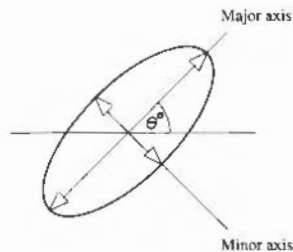
Try the following program:

```
10 MODE 15
20 REPEAT
30 GCOL RND (64):MOUSE x,y,z
40 CIRCLE FILL x, y, RND(400)+50
50 UNTIL FALSE
```

This program produces circles in random colours, centred at any position on the screen between (1,1) and (1279,1023) and with a radius of between 51 and 450. To stop it press Escape.

ELLIPSE AND ELLIPSE FILL

To draw the outline of an ellipse or to plot a solid ellipse you need to provide its centre point and the size of its major and minor axes. In addition, you may also give the angle by which it is rotated from the horizontal.



For example:

```
ELLIPSE 640,512,200,100,PI/4
```

This produces the outline of an ellipse centred at (640,512). The length of it is 200, the width is 100 and it is rotated by $\text{PI}/4$ radians (45 degrees) from the horizontal.

If the angle is omitted, an axis-aligned ellipse is produced:

```
ELLIPSE 400,500,320,80
```

Try the following program:

```
10 MODE 1
20 GCOL 1
30 FOR angle = 0 TO 3*PI/4 STEP PI/4
40 ELLIPSE FILL 640,512,200,60,angle
50 NEXT
60 GCOL 2
70 FOR angle = PI/8 TO 3*PI/4+PI/8 STEP PI/4
```

```
80 ELLIPSE FILL 640,512,100,30,angle
90 NEXT
```

This plots eight ellipses in two different sizes with the same centre point to form multi-petalled flowers.

GCOL

In previous examples, GCOL has taken one parameter, a number which selects the current logical colour for the graphics foreground or background. For example,

```
GCOL 3
GCOL 129
```

selects the graphics foreground colour to be logical colour three and the background colour to be one.

GCOL may, however, take two parameters, in which case the second selects the foreground and background graphics colours, and the first selects the manner in which this colour, 'c', is applied to the screen as follows:

- | | |
|---|---|
| 0 | The colour 'c' |
| 1 | OR the colour on the screen with 'c'. |
| 2 | AND the colour on the screen with 'c' |
| 3 | EOR the colour on the screen with 'c' |
| 4 | Invert the colour on the screen (disregards 'c') |
| 5 | Leave the colour on the screen unchanged (disregards 'c') |
| 6 | AND the inverse of the colour on the screen with 'c'. |
| 7 | OR the inverse of the colour on the screen with 'c'. |

Two of the options ignore the second parameter and either leave the colour on the screen unchanged or invert it. Inverting a colour means that all the bits in the colour number are altered: zeros are set to ones and vice versa. For example:

```
10 MODE 9 : REM 16 colours 0(%0000) - 15 (%1111)
20 GCOL 128+5
30 CLG
40 GCOL 4,0 : DRAW 100,100
```

The colour on the screen is colour 5 (%0101). The colour used to draw the line is, therefore, colour 10 (%1010).

The OR, AND and EOR operators act on the bits of the colour already on the screen and on the colour given as the second GCOL parameter as described in the chapter: **VARIABLES AND EXPRESSIONS**. Thus:

```
10 MODE 2 : REM 16 colours 0(%0000) - 15(%1111)
20 GCOL 128+5
30 CLG
40 GCOL 0,6 : DRAW 100,100
50 GCOL 1,6 : DRAW 200,200
60 GCOL 2,6 : DRAW 300,300
70 GCOL 3,6 : DRAW 400,400
80 GCOL 6,6 : DRAW 500,500
90 GCOL 7,6 : DRAW 600,600
```

The colour already on the screen when the lines are drawn is colour five (%0101). The foreground colour is selected as colour six (%0110) in all cases. The method of applying it to the screen, however, alters the actual colour displayed as follows:

- The first line appears in colour six
- The second line appears in colour seven
(%0101 OR %0110 = %0111)

- The third line appears in colour four
(%0101 AND %0110 = %0100)
- The fourth line appears in colour three
(%0101 EOR %0110 = %0011)
- The fifth line appears in colour one
(%0101 AND %1001 = %0001)
- The sixth line appears in colour two
(%1010 OR %0110 = %0010)

THE GRAPHICS CURSOR

The graphics cursor is an invisible point on the screen which affects where lines and other items are drawn from. For example:

```
10 MODE 1
20 MOVE 100,100
30 DRAW 200,200
```

This moves the graphics cursor to (100,100), then draws a line to (200,200) and leaves the graphics cursor at this position. Now, if a further line is added to the program as follows:

```
40 DRAW 300,100
```

this adds a line from (200,200) to (300,100).

USING PLOT TO PRODUCE OTHER SHAPES

The commands such as MOVE, DRAW, CIRCLE, etc are special cases of a more general PLOT command. This command can give a far wider range of options over what kind of shape you produce and how you produce it. Of course, the added functionality it provides makes it more complicated to use.

PLOT takes the following format:

PLOT *k, x, y*

where 'k' is the mode of plotting, and 'x' and 'y' are the coordinates of a point to be used to position the shape. PLOT takes one pair of coordinates. To produce shapes which need more than one pair to define them, such as rectangles, it uses the previous position or positions of the graphics cursor to provide the missing information. This means that you must pay careful attention to the position of the graphics cursor after a shape has been drawn. Otherwise future plots may produce unexpected results.

Each type of plot has a block of eight numbers associated with it. These are listed below in both decimal and hexadecimal notation. (See the chapter: BASES).

0-7	(&00 - &07)	Solid line including both end points
8-15	(&08 - &0F)	Solid line excluding final points
16-23	(&10 - &17)	Dotted line including both end points
24-31	(&18 - &1F)	Dotted line excluding final points
32-39	(&20 - &27)	Solid line excluding initial point
40-47	(&28 - &2F)	Solid line excluding both end points
48-55	(&30 - &37)	Dotted line excluding initial point
56-63	(&38 - &3F)	Dotted line excluding both end points
64-71	(&40 - &47)	Point plot
72-79	(&48 - &4F)	Horizontal line fill (left & right) to non-background
80-87	(&50 - &57)	Triangle fill
88-95	(&58 - &5F)	Horizontal line fill (right only) to background
96-103	(&60 - &67)	Rectangle fill
104-111	(&68 - &6F)	Horizontal line fill (left & right) to foreground
112-119	(&70 - &77)	Parallelogram fill
120-127	(&78 - &7F)	Horizontal line fill (right only) to non-foreground
128-135	(&80 - &87)	Flood to non-background
136-143	(&88 - &8F)	Flood to foreground
144-151	(&90 - &97)	Circle outline
152-159	(&98 - &9F)	Circle fill

160-167	(&A0 - &A7)	Circular arc
168-175	(&A8 - &AF)	Segment
176-183	(&B0 - &B7)	Sector
184-191	(&B8 - &BF)	Block copy/move
192-199	(&C0 - &C7)	Ellipse outline
200-207	(&C8 - &CF)	Ellipse fill
208-215	(&D0 - &D7)	Graphics characters
216-223	(&D8 - &DF)	Reserved for Acorn expansion
224-231	(&E0 - &E7)	Reserved for Acorn expansion
232-39	(&E8 - &EF)	Sprite plot
240-247	(&F0 - &F7)	Reserved for user programs
248-255	(&F8 - &FF)	Reserved for user programs

Within each block of eight, the offset from the base number has the following meaning:

- 0 **8** move cursor relative (to last graphics point visited)
- 1 **9** draw relative using current foreground colour
- 2 **A** draw relative using logical inverse colour
- 3 **1** draw relative using current background colour
- 4 **C** move cursor absolute (ie move to actual co-ordinate given)
- 5 **O** draw absolute using current foreground colour
- 6 **6** draw absolute using logical inverse colour
- 7 **F** draw absolute using current background colour

PLOT is a good example of where using hexadecimal notation helps to make things clearer. Each block of eight starts at either &x0 or &x8, where 'x' represents any hexadecimal digit, so a plot absolute in the current foreground colour, for example, has a plot code of &x5 or &xD. Thus, it is obvious which mode of plotting is being used. Similarly, it is obvious which shape is being plotted, and

so, for example, if the plot is between &90 and &9F, then it is a circle. This is a far easier range to recognise than 144 to 159.

Each of the types of plot is described in further detail below.

Plotting simple lines;

A line is plotted between the coordinates given by the PLOT and the previous position of the graphics cursor. The following examples draw a line from (200,200) to (800,800):

```
10 MODE 0
20 PLOT &04, 200, 200
30 PLOT &05, 800, 800
```

These two PLOT statements are equivalent to MOVE 200, 200 and DRAW 800, 800 respectively.

The same line can be drawn by a different PLOT code:

```
10 MODE 0
20 PLOT &04, 200, 200
30 PLOT &01, 600, 600
```

This demonstrates the use of relative plotting. The coordinate (600,600) which has been given in line 30 is relative to the position of the graphics cursor. The absolute value is obtained by adding this offset to the previous position ie (600,600) + (200,200) which gives a position of (800,800). This is equivalent to DRAW BY 600, 600.

Dot-dash lines

Straight lines do not have to be drawn as a continuous line. Instead you can set up a pattern of dots and dashes and use that.

A dot-dash pattern is set up using:

```
VDU 23, 6, n1, n2, n3, n4, n5, n6, n7, n8
```

where n1 to n8 define a bit pattern. Each bit which is set to one represents a point plotted and each bit set to zero represents no point. The maximum pattern repeat is 64. However, you can set up any repeat between one and 64 using:

```
*FX 163, 242, n
```

If you set 'n' to zero, this sets up the default pattern which has a repeat of eight and is alternately on and off, ie %10101010 (&AA).

There are four different methods which may be used to plot the line:

PLOT range	Effect
&10-&17	Both end points included, the pattern being restarted when each new line is drawn.
&18-&1F	Final point omitted, the pattern being restarted when each new line is drawn.
&30-&37	Initial point omitted, the pattern being continued when each new line is drawn.
&38-&3F	Both end points omitted, the pattern being continued when each new line is drawn.

Triangles

To draw a triangle, you need the coordinates given with the triangle PLOT code and two previous points which mark the other corners. For example:

```
10 MODE 12
20 MOVE 200, 200
```

```
30 MOVE 600,200
40 PLOT &55,400,400
```

This plots a triangle with corners (200,200), (600,600) and (400,400). Adding a further line:

```
50 PLOT &55,800,400
```

plots a further triangle using corners (600,200), (400,400) and (800,400).

Rectangles

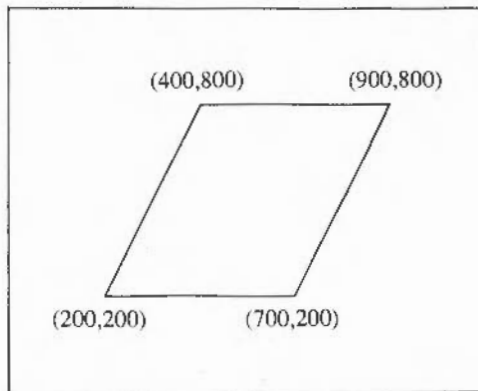
An axes-aligned rectangle can be plotted between the coordinates given by the PLOT and the previous position of the graphics cursor. For example:

```
MOVE 200,200
PLOT &65,800,800
```

This is equivalent to `RECTANGLE FILL 200,200,600,600`.

Parallelograms

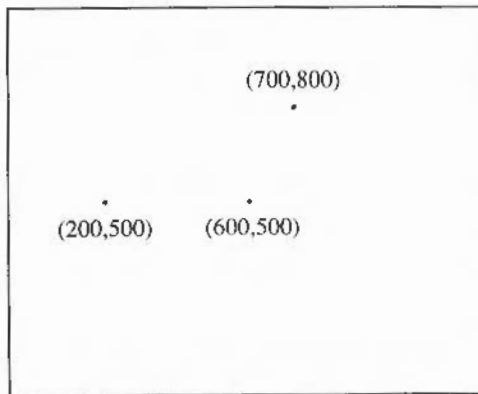
Parallelograms are constructed as rectangles which have been sheared sideways.
For example:



These require three points to define them. Thus to plot the parallelogram shown above the following could be used:

```
MOVE 200,200  
MOVE 700,200  
PLOT &75,900,800
```

Although any three corners of the parallelogram may be used to define it, the order in which these are given affects which way round the parallelogram appears. Consider the three points given below:

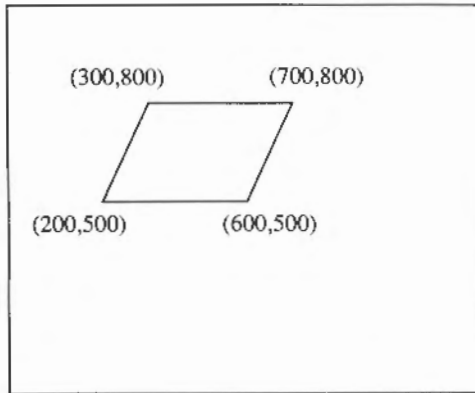


These could produce any of the following three parallelograms, depending on the order in which they were used:

```
MOVE 200,500  
MOVE 600,500  
PLOT &75,700,800
```

OR

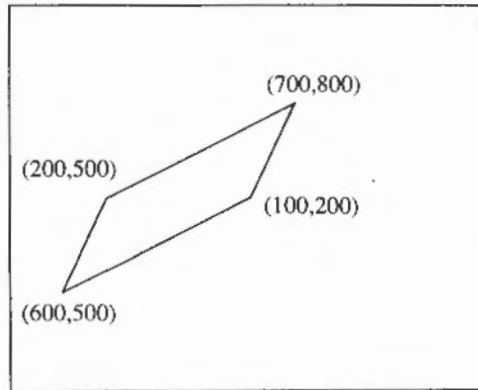
```
MOVE 700,800  
MOVE 600,500  
PLOT &75,200,500
```



```
MOVE 200,500  
MOVE 700,800  
PLOT &75,600,500
```

or

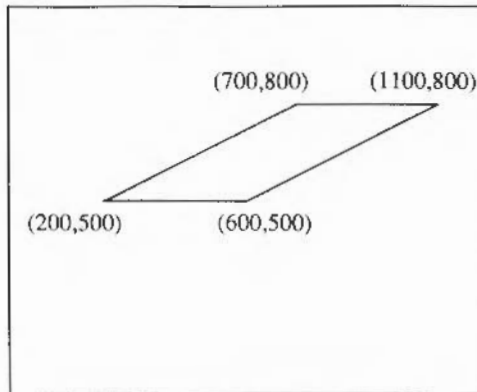
```
MOVE 600,500  
MOVE 700,800  
PLOT &75,200,500
```



```
MOVE 600,500  
MOVE 200,500  
PLOT &75,700,800
```

OR

```
MOVE 700,800  
MOVE 200,500  
PLOT &75,600,500
```



Hence the points are taken as moving around the edge of the parallelogram in sequence, the final point being opposite the middle one defined.

Circles

To plot a circle define the centre by moving it, and then use PLOT with the relevant plot code and the coordinates of a point on its circumference. For example, to plot a solid circle in the centre of the screen with a radius of 100, type

```
MOVE 640,512      :REM centre
PLOT &9D,740,512  :REM Xcentre+radius,Ycentre
```

Alternatively you could use relative plotting :

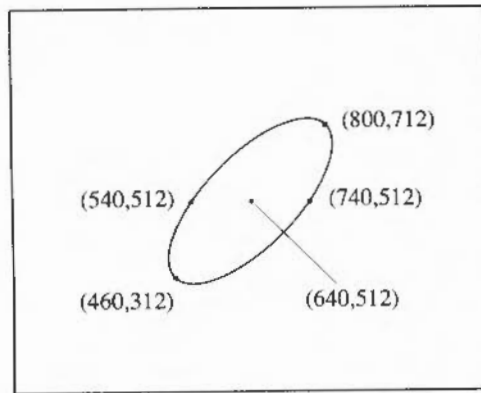
```
MOVE 640,512      :REM centre
PLOT &99,100,0     :REM radius,0
```

In both these examples the circles are solid and could have been produced using the CIRCLE FILL command. The equivalent of the CIRCLE command for producing outlines of circles would be PLOT &95 and PLOT &91.

Ellipses

Ellipses are more complicated to define than circles. To plot an ellipse the following information is required:

- the centre point
- an outermost point (either to the right or left) at the same height as the centre
- the highest or lowest point of the ellipse.



For example, to draw the ellipse above, you could use:

MOVE 640, 512	the centre
MOVE 740, 512	the right-hand point
PLOT &C5, 800, 712	the top point

or alternatively:

MOVE 640, 512	the centre
MOVE 540, 512	the left-hand point
PLOT &C5, 480, 312	the bottom point

Note that only the 'x' coordinate of the second point is relevant, although for clarity it is good practice to give the same 'y' coordinate as for the centre point.

The following example creates a pattern using a number of differently shaped ellipses:

```

10 MODE 0
20 FOR step% = 0 TO 400 STEP 25
30 MOVE 640,512
40 MOVE 215+step%,512
50 PLOT &C5,640,512+step%
60 NEXT

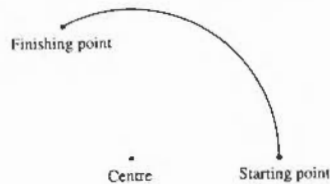
```

Solid ellipses are drawn in the same way using the plot codes &C8 to &CF.

The BASIC ELLIPSE keyword provides an easier way of specifying rotated ellipses.

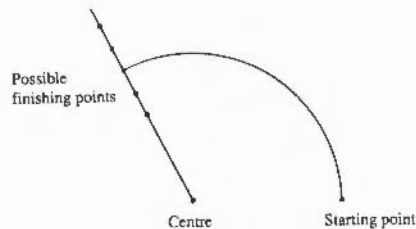
Arcs

We saw above how circle outlines are defined and drawn. In a similar way, just a portion of the circle outline may be drawn to produce an arc. In this case, three points are required: the centre of the circle and two points to indicate the starting and finishing points of the arc. Ideally, these would be given as follows:



In the example above, however, both the starting and finishing points are on the arc itself. This is a design which requires a large amount of calculation. It is easier for the starting point to be taken as being on the arc and used to calculate the radius, the finishing point being used just to indicate the angle the arc subtends.

For example:



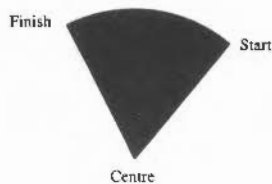
This is the method used by BASIC. To draw an arc, you need to specify the centre of the circle it is based upon and the starting point of the arc, and then to plot to a third point to specify the angle.

The example below draws an arc based on a circle whose centre is at (640,512). It draws the portion of the arc from 0 to 270. Since arcs are drawn anticlockwise this means that its starting position is the point (440,512) (270) and its finishing position (640,512+n) (0):

```
MOVE 640,512
MOVE 440,512
PLOT &A5,640,1000
```

Sectors

A sector is a filled shape enclosed by two straight radii and the arc of a circle.



Sectors are defined in a similar manner to arcs. For example:

MOVE 640, 512	centre point
MOVE 440, 512	starting point on the circumference
PLOT &B5, 640, 1000	point indicating angle of sector

Again the sector is taken as going anti-clockwise from the starting point to the finishing point.

Segments

A segment is an area of a circle between the circumference and a chord as shown below:



Segments are defined in exactly the same way as arcs and sectors.

PATTERN FILLS

Any of the colours which are available in a given mode may be interwoven to give a tremendous range of colour patterns. When using modes with a limited number of colours, for example any of the four-colour modes, this feature may be used to extend the colours available, since combining similar colours produces further shades which look like pure colours. Alternatively, contrasting colours may be used to give checks, wavy lines, and so on.

Default patterns

Default patterns are set up for you as follows:

Mode(s)	Pattern	Colour
0	1	Dark grey
	2	Grey
	3	Light grey
	4	Hatching
4,18	1	Dark grey
	2	Grey
	3	Light grey
	4	Hatching
1,5,8,19	1	Red-orange
	2	Orange
	3	Yellow-orange
	4	Cream
2,9,12,20	1	Orange
	2	Pink
	3	Yellow-green
	4	Cream
13,15	1	White-grey stripes
	2	Black-grey stripes
	3	Green-black stripes
	4	Pink-white stripes

To use these patterns you issue a GCOL with a plot action which depends on the pattern desired. In general, to use pattern 'n', the GCOL command should be

```
GCOL n*16+action,col
```

where *action* is the plotting action you want to use with the pattern (for example 0 for force, 1 for OR etc, as described earlier), and *col* is 0 if you want to set the foreground colour as a pattern or 128 for a background pattern. The parameter 'n' is in the range 1 to 4 for the normal patterns, or 5 for a large pattern which is formed by placing patterns 1 to 4 next to each other.

Plotting using pattern fills

All the shapes which have been described above can be plotted using these colour patterns. A pattern may be selected using GCOL. The first parameter to GCOL affects the plotting action as was seen in the chapter: **SCREEN MODES**. Patterns can be used in future plots by using values in the following ranges:

16-31 Pattern 1
 32-47 Pattern 2
 48-63 Pattern 3
 64-79 Pattern 4

Try the following:

```
10 MODE 9
20 GCOL 16,0
30 MOVE 100,100
40 MOVE 800,800
50 PLOT &55,700,200
```

or

```
10 MODE 1
20 GCOL 32,0
30 MOVE 640,512
40 PLOT &9D,740,512
```

It is possible to plot lines using these colour patterns in a similar manner, but the effects may be rather strange. Consider, for example, a line drawn at 45 degrees in mode one. If the pattern being used was alternate black and white pixels, then this line would be drawn either in all white or all black, the latter not being visible on a black background.

Defining your own patterns

You may define your own colour patterns using VDU commands as follows:

VDU 23, 2, n1, n2, n3, n4, n5, n6, n7, n8 defines GCOL 16, 0 ie pattern one.

VDU 23, 3, n1, n2, n3, n4, n5, n6, n7, n8 defines GCOL 32, 0 ie pattern two

VDU 23, 4, n1, n2, n3, n4, n5, n6, n7, n8 defines GCOL 48, 0 ie pattern three

VDU 23, 5, n1, n2, n3, n4, n5, n6, n7, n8 defines GCOL 64, 0 ie pattern four

The pattern produced by a set of parameters depends upon which pattern mode is being used. There are two modes available, one where the parameters are interpreted in the same manner as on a BBC Master 128 and another simpler method used by this machine only. The default is the BBC Master 128 mode. To change to the Archimedes (native) mode type

```
VDU 23, 17, 4, 1, 0, 0, 0, 0, 0, 0
```

To revert back again to the Master mode type

```
VDU 23, 17, 4, 0, 0, 0, 0, 0, 0, 0
```

The pattern fill works with blocks of pixels. The size of these blocks depends on the number of colours available in the mode:

Pixels horizontally	Pixels vertically	Colours in mode
8	8	2
4	8	4
2	8	16
1	8	256

In all cases, each pixel may be assigned a colour independently of the others. Each parameter of the VDU command corresponds to a row in the pixel block. The first parameter contains the value of the top row, the second the value of the second row, and so on. The way the value of the parameter is interpreted depends on the mode being used.

Native mode patterns

In native mode the bits of the parameter are used in a straightforward manner to give the colour of the pixels.

- *Two-colour modes*: Each bit of the parameter is assigned to a pixel, the least significant bit applying to the pixel on the left. For example, to set a row of the pattern as follows:

black,	white,	white,	white,	black,	black,	black,	white
%0	%1	%1	%1	%0	%0	%0	%1

the value required is 142 (%10001110).

- *Four-colour modes*: Each pair of bits of the parameter is assigned to a pixel, the least significant pair applying to the pixel on the left. For example, to set a row of the pattern as follows:

yellow	red	white	yellow
%10	%01	%11	%10

the value required is 182 (%10110110).

- *16-colour modes*: Each set of four bits of the parameter is assigned to a pixel, the least significant set applying to the pixel on the left. For example, to set a row of the pattern as follows:

green	white
%0010	%0111

the value required is 114 (%01110010).

- *256-colour modes*: The value of the parameter defines the colour assigned to the pixel directly.

BBC Master 128 compatible patterns

The patterns in these cases are more complex since they involve interweaving the bits from the colour to obtain the parameter value.

- *Two-colour modes*: This is the easiest case to understand. Each pixel in the block corresponds to one bit of the parameter, the least significant bit applying to the pixel on the right. For example, to set a row of the pattern as follows:

```
black, white, white, white, black, black, black, white
%0    %1    %1    %1    %0    %0    %0    %1
```

the value required is 113 (%01110001).

Defining a pattern in a two-colour mode is equivalent to setting up a user-defined character.

- *Four-colour modes*: In four-colour modes each colour is defined using two bits as follows:

```
yellow (2) red (1)      white (3)      yellow (2)
%10        %01        %11        %10

  1          0
   0          1
    1          1
     1          0
    1 0 1 1 0 1 1 0
```

and the value required is 182 (%10110110).

- *16-colour modes*: In 16-colour modes the situation is different again. There are just two pixels in a row, four bits of the parameter being used to hold the value of each colour. However, it is not the case that the top four bits correspond to the first colour and the bottom four bits to the other. Instead, the bits of each are interleaved:

green (2) white (7)

%0111 %0111

```

    0  0  1  0
      0  1  1  1
    0  0  0  1  1  1  0  1

```

and the value required is 29 (%00011101).

To get the colours the other way around different numbers are required:

white (7) green (2)

%0111 %0010

```

    0  1  1  1
      0  0  1  0
    0  0  1  0  1  1  1  0

```

and the value required is 46 (%00101110).

Thus a cross-hatch pattern of alternate white and green pixels can be defined:

VDU 23,2,29,46,29,46,29,46,29,46

Giant patterns

Giant patterns can be set up which take all four of the separate patterns and place them side by side, giving an overall pixel size as shown below:

Pixels horizontally	Pixels vertically	Colours in mode
32	8	2
16	8	4
8	8	16
4	8	256

To produce a giant pattern in this way, the first parameter given to GCOL should be in the range 80 to 95.

Simple patterns

Often the most effective way of using the pattern fills is for simple cross-hatch patterns. If you want to use this sort of colour pattern, a simpler way of defining it is available. In this method, just a small block of eight pixels is defined which is used to form the normal-sized block as shown below.

1	2
3	4
5	6
7	8

16 colour modes

1	2
3	4
5	6
7	8

4 colour modes

The eight pixel colours are set up using

VDU 23, 2, n1, n2, n3, n4, n5, n6, n7, n8

where n1 to n8 correspond to the actual colour to be used. The numbers are given in the following order:

1	2		
3	4		
5	6		
7	8		

Mode 4

Double pixels

1	2		
3	4		
5	6		
7	8		

Mode 8

FLOOD-FILLS

This section is concerned with how to fill the inside of any closed region, however awkward the shape. The method used is flood-filling; with this you can start off at any point within the boundaries of the shape. The whole shape is then filled at once.

Flood to non-background

This can be used on shapes which are in the current background colour and bordered by non-background colours. The shape is filled with the current foreground colour.

To use this flood-fill method, type, for example:

```
FILL 640,512
```

This starts filling from the point (640,512): the middle of the screen. If this point is in a non-background colour then it returns immediately. Otherwise it fills in all directions until it reaches either a non-background colour or the edge of the screen.

Flood-fills may be performed using either pure colours or colour patterns. Note that if you wish to colour in a shape it must be totally enclosed by a solid border. If there is a gap anywhere then the colour leaks out into other regions.

Flood until foreground

Whereas the previous flood-fill filled a shape currently in the background colour, this one fills a shape currently in any colour except the present foreground one, with the present foreground colour. This is performed by a PLOT command with plot codes &88 to &8F. For example:

```
PLOT &8D, 640, 512
```

Flood-filling will only succeed when the region being filled does not already contain any pixels in the colour being used. For example, if you are attempting a flood to non-background when the background colour is black, you should not try to flood in black or in a pattern which contains black pixels.

COPYING AND MOVING

Using RECTANGLE ... TO and RECTANGLE FILL ... TO, you can pick up a rectangular area of the screen and either make a copy of it elsewhere on the screen or move it to another position, replacing it with a block of the background colour. For example:

```
RECTANGLE FILL 400, 600, 60, 80 TO 700, 580
```

This marks out the source rectangle as having one corner at coordinates (400,600), a width of 60 and a height of 80. It then moves this rectangular area so that the bottom left of it is at the coordinates (700,580). The old area is replaced by background.

The new position can overlap with the rectangular area, as in the example above, and the expected result is still obtained.

The rectangle move and copy commands may also be expressed in terms of PLOT codes. The relevant range of codes is &B8 to &BF: first move to two points which denote the bottom left and top right of the rectangle to be copied or moved; then plot, using one of the range of codes described above, to the bottom left corner of the destination rectangle. The meanings of the plot codes are as follows:

&B8	Move relative (no copy/move operation)
&B9	Relative rectangle move
&BA	Relative rectangle copy
&BB	Relative rectangle copy
&BC	Move absolute (no copy/move operation)
&BD	Absolute rectangle move
&BE	Absolute rectangle copy
&BF	Absolute rectangle copy

The rectangle move operations erase the source rectangle, whereas the copy operations leave it intact. So, the `RECTANGLE FILL ... TO` example above could also be expressed as:

```
MOVE 400,600
MOVE BY 60,80
PLOT &BD,700,580
```

PRINTING TEXT AT THE GRAPHICS CURSOR

Printing text at the text cursor positions gives only limited control over the places at which characters may be located. In addition it does not allow characters to overlap. Attempting to print one character on top of an existing one deletes the existing one first. You may find that you would like to be able to place text in different positions, for example to label the axes of a graph or to type two characters on top of each other, so as to add an accent '^' to a letter. To do either of these type

```
VDU 5
```

You are now in VDU 5 mode. Whilst you are in this mode of operation, any characters you print are placed at the graphics cursor position. The text cursor is ignored. Use the `MOVE` statement to locate the text precisely.

Since this method of printing makes use of graphics facilities, it is not allowed in text-only modes. If the command `VDU 5` is given in any of these screen modes it has no effect.

Each character is actually placed so that the top left of it is at the position of the graphics cursor. After the character has been printed, the graphics cursor moves to the right by the width of one character. Although the graphics cursor also automatically moves down by the height of a character (32 units or 16 units in modes 18 to 0) when the right-hand side of the screen is reached, the screen does not scroll when a character is placed in the bottom right-hand corner. Instead the cursor returns to the top left.

To return to the normal mode of operation type

VDU 4

The Archimedes operating system allows the programmer to set up special rectangular areas of the screen, called windows, in order to restrict where text or graphics can appear on the screen.

Text windows provide automatic scrolling of text written into the window area, and so are also referred to as 'scrolling windows'.

Graphics windows restrict the area affected by graphics operations, so that, for example, lines are clipped to lie within the window area. Graphics windows are therefore also referred to as 'clipping windows'.

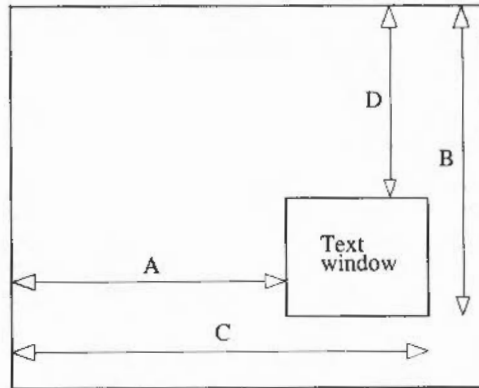
Text and graphics windows are supported directly by the VDU drivers, and are quite distinct from the bordered, moveable windows used by the desktop manager software.

TEXT WINDOWS

Normally, text may appear anywhere on the screen. However, you can set a text window, which allows the text to appear only inside the window. To set up a text window, use the VDU 28 command as follows:

```
VDU 28, a, b, c, d
```

where a,b is the bottom left-hand and c,d the top right-hand position inside the window given in text coordinates:



Nothing outside the text window is affected by text statements, such as CLS to clear the text screen, or screen scrolling. Note that TAB (X, Y) positions the text cursor relative to the position of the top left of the current text window. The following program demonstrates how text windows behave:

```
10 MODE 1
20 REM Set up a text window 6 characters square
30 VDU 28,5,10,10,5
40 REM Change the background colour to colour 1 (red)
50 COLOUR 129
60 REM Clear the text screen to show where it is
70 CLS
80 REM Demonstrate scrolling
90 FOR N% = 1 TO 20
100 PRINT N%
110 NEXT N%
120 REM Show position of character (2,3)
130 PRINT TAB(2,3); "*"
140 END
```

To revert back to having the whole screen as the text window type

VDU 26

The precise actions of the VDU 26 command are as follows:

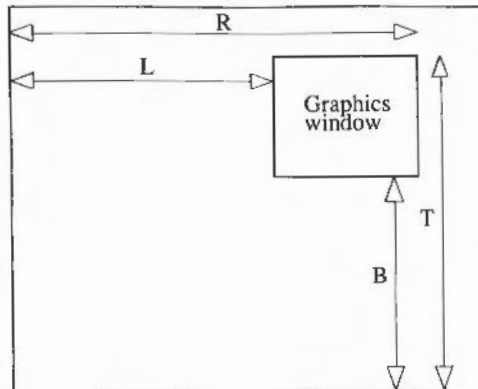
- Restore text window to the whole screen
- Restore graphics window to the whole screen
- Home the text cursor
- Restore graphic origin to bottom left of screen
- Home graphics cursor to (0,0)

GRAPHICS WINDOWS

Just as text may have a text window defined, so a graphics window may be set up using

VDU 24,c;b;r;t;

where (c,b) and (r,t) are the coordinates of the lower left-hand and upper right-hand pixels inside the window. (Be sure to use semi-colons as indicated, not commas.)



Nothing outside the graphics window is affected by graphics commands, such as CLG to clear the graphics screen. When a graphics window is set up, the graphics origin (0,0) is unaltered. The following program demonstrates how graphics windows behave:

```
10 MODE 12
20 REM Set up a graphics window, a quarter of the screen size
30 VDU 24,320;256;960;768;
40 REM Change the background colour to colour 1 (red)
50 GCOL 129
60 REM Clear the graphics window
70 CLG
80 REM Show position of 0,0
90 CIRCLE 0,0,600
100 END
```

To revert back to having the whole screen as the graphics window type

```
VDU 26
```

SPRITES

A sprite is a user-defined graphics character which can be any size or shape and which can contain different colours. Sprites can be used for animating pictures, games, and so on. A sprite editor is provided to help you to define sprites for use in a program. The sprites can then be plotted and unplotted (removed from the screen) using a simple PLOT command.

When sprites are being defined or used they reside in memory. The amount of memory allocated for storing sprites is configurable in chunks of 8K. For example,

```
*CONFIGURE SPRITESIZE 3
```

configures the machine so that 24K (3 * 8K) is reserved automatically for sprite workspace when the machine is powered on.

THE SPRITE EDITOR

You can use the sprite editor provided on the Welcome Disc to create new sprites and to modify existing ones. To load and run the editor, place the Welcome Disc in the disc drive and (unless you happen to be in the Utilities directory already) type

```
CHAIN "$.UTILITIES.SEDIT"
```

You may also, of course, select the sprite editor from its icon in the desktop program.

The program displays a list of the sprites currently in memory. The name, width, height and mode of each sprite is given.

The first time you use this program, the list is empty since you have not yet created any sprites. To create a sprite, move the pointer to the CREATE option at the bottom of the screen and press any of the mouse buttons. You are then prompted for a name and a mode. The name can contain up to ten characters: `sprite1`, `man_right`, `2`, and so on. The mode is the screen mode which you will use when you plot the sprite.

When one or more sprites exist, one of the entries is highlighted. To edit the sprite concerned, move the pointer so that it is pointing to the highlighted entry and press any of the mouse buttons. The mode changes to that of the sprite selected and the screen changes to the sprite editing screen.

To edit a different sprite, you must first select it by pointing to it and pressing any of the mouse buttons. This highlights it so that you can select it as described above. If you have a large number of sprites in memory, you may find that they will not all fit on the screen. You can scroll through them by pointing above the first sprite entry or below the bottom one and pressing any button.

It is important at this stage to note that when you edit an existing sprite you are updating the only copy of it which exists in memory. There is no way of leaving the editor and retaining the original state of the sprite. Consequently, it is a good idea to save a copy of the original sprite to disc first, so that if you make a mistake whilst editing it and wish to return to the original, you can load it back into memory again.

Editor display

The sprite editor shows you two versions of the sprite, one actual-sized one in a display box on the right and another larger version in an enlargement box on the left. The version in the display box shows you what your sprite will look like when you use it in your program. The version in the enlargement box is the one you use to make your alterations. If your sprite is too large for all of it to be displayed in the enlargement box, only the bottom left-hand corner of it is visible. The lines in the border of the display box indicate the position of the pixel in the centre of the enlargement box.

At the top of the screen some information representing the status of the current sprite is displayed. The name of the sprite is printed on the top line. Below this is the mode you are using and the current size of the sprite given in bytes horizontally and vertically. Beneath this is a letter 'S' or 'T', which indicates whether the sprite is solid (has no mask) or transparent (has a mask defined).

Below the text is a small solid block which is plotted in the current colour. The current logical colour can be selected by pointing to one of the available colours displayed in the bar beneath it and pressing any mouse button.

In the two-, four- and 16-colour modes the colour bar is one row deep and contains two, four or 16 sections, each representing one of the colours which may be used. In addition, there are three boxes below this which can be used to change the palette for the current colour. Each has a line across it indicating the amount of red, green and blue contained in the current colour. The higher up the line, the more of that primary colour the actual colour contains. To alter any of these settings, point to a new position within one of the boxes and press a button. Pressing the left-hand button selects it as a solid colour, the middle button selects it as the first flash state and the right-hand button selects it as the second flash state. Note that altering the palette also alters any of the pixels in the sprite which are currently plotted in the current colour.

In 256-colour modes the colour bar is four rows deep and 64 sections across, giving 256 different colours which may be selected. No control of the palette is available in these modes.

Colouring in the pixels

The colour of a pixel can be altered by pointing to it in the enlargement box and pressing a button. The left-hand button sets it to the current colour as displayed at the top of the screen, the middle button sets it to the logical colour for text or graphics, which is by default white, and the right-hand button sets it to the logical colour for background, which is by default black.

Moving about the sprite

If the sprite is too large to be displayed in the enlargement box, you can move around it by pointing to the part of the complete sprite, shown in the display box, which you want to appear in the centre of the enlargement box and pressing any button.

Altering the size of a sprite

The size of a sprite can be altered by adding rows and columns to the top and right of it using the following keys:

- **F3** adds another row at the top
- **F4** adds another column at the right
- **Shift F3** deletes the top row
- **Shift F4** deletes the right-hand column

It is possible for the sprite to reach such a size that it cannot allow you to add any further rows or columns. This occurs when the size of a sprite (in bytes) would be greater than the amount of memory left in the space reserved for your sprites. If this occurs, you need to leave the sprite editor and alter the sprite space as described above.

It is good practice to remove any unused rows and columns when you have finished designing a sprite since these make the sprite appear larger than necessary. They also waste memory and increase the time taken to plot the sprite.

Often, when designing symmetrical shapes, it is easier to work inwards from the edges towards the centre. This invariably means that you end up wanting to add or delete a number of rows and columns in the middle of the sprite.

To add an extra row or column, press **F5** or **F6** respectively and a new blank row or column is added at the current pointer position (which must be within the enlargement box).

Shift F5 and **Shift F6** have the opposite effects. They delete the row or column currently pointed at.

Other useful commands

You can use **f1** to speed up the colouring process. This fills the row sideways out from the pointer position in the current colour until it finds a pixel which is in a different colour from the one it is changing. Thus, if the cursor is on top of a white pixel and the current colour is red, it looks along to the left and right to find the first non-white pixel in each direction and changes all the pixels in between to red. **f2** works in a similar manner on columns.

Two other commands are **f7** and **f8** which allow you to reflect your sprite so that it either faces the other way or is turned upside down. **f7** is particularly useful for games in which the shape you are moving around the screen often needs to face the way it is moving, so two copies are required which are mirror images of each other.

To create a mask for the sprite you are editing, press **Shift f9**. This changes the 'S' at the top of the screen to a 'T' so indicating that the sprite has a mask. Pressing **f9** sets the current colour to transparent which is indicated by a black and white cross-hatch pattern. This allows the mask to be defined. To delete a mask press **Ctrl f9**.

Finally when you are happy with the sprite you have designed, press **Escape** to leave the sprite editor.

SPRITE * COMMANDS

Several * commands are provided to enable you to act on the sprites in memory. These are totally separate from the sprite editor and can either be issued whilst in BASIC command mode or included within programs.

Changing the name of a sprite

You can change the name of a particular sprite by typing

```
*SRENAME name, newname
```

This renames sprite name as sprite newname. Note that if there is a sprite in memory already with the identifier newname, an error message is displayed.

Copying sprites

To copy a sprite definition type

```
*SCOPY name,newname
```

This copies the definition of sprite name and uses it to identify the new sprite newname.

Deleting sprites

To delete a sprite, type

```
*SDELETE name
```

This deletes the sprite whose identifier is name.

To delete all the sprites, type

```
*SNEW
```

Loading and saving sprites

All the sprites which you currently have in memory can be saved as a file by typing

```
*SSAVE filename
```

You can also load a sprite file back into memory so that you can either plot or edit the sprites in it by first reserving a sufficient area of memory for it using *SSPACE, and then typing

```
*SLOAD filename
```

If you wish to merge a file of sprites with those you have in memory, then you can use:

```
*SMERGE filename
```

This command, however, should be used with care if there is a sprite currently in memory which has the same number as one of the sprites in the file, since when the two are merged, the version which was in memory will be lost.

PLOTTING SPRITES

It is very simple to plot one of the sprites which you currently have in memory. All you have to do is select which sprite you wish to plot and where you want it to appear on the screen. To select a particular sprite, use:

```
*SCHOOSE name
```

A PLOT command is then used to put it on the screen. The possible plot numbers which may be used are &E8 to &EF. For example:

```
*SCHOOSE horse
PLOT &ED, 640, 512
```

plots sprite `horse` with its bottom left-hand corner exactly in the centre of the screen.

As an alternative to *SCHOOSE, the following VDU command may be used for certain sprites:

```
VDU 23,27,0,n,0,0,0,0,0,0
```

This is equivalent to *SCHOOSE when the name of the sprite contains just digits, the values of which are numbers between zero and 255. The advantage of using the VDU command is that it can contain variables. If you wish to plot eight sprites, whose names are 1 to 8, you can use, for example:

```
FOR sprite_num% = 1 TO 8
VDU23,27,0, sprite_num%,0,0,0,0,0,0
PLOT &ED,x%,y%
NEXT
```

Any * command, however, can only be used with constant parameters, and so if *SCHOOSE is to perform an equivalent task you will have to type

```
*SCHOOSE 1
PLOT &ED,x%,y%
*SCHOOSE 2
PLOT &ED,x%,y%
```

```
.....
.....
```

```
*SCHOOSE 8
PLOT &ED,x%,y%
```

Plotting with a mask

If a sprite has a mask defined, this can be used whilst plotting so that the area of the mask leaves the background unchanged: the mask area is treated as if it were transparent. To plot using the mask, use the GCOL plotting modes 8 to 15. For example:

```
GCOL 8,0
*SCHOOSE horse
PLOT &ED, 640, 512
```

Plotting within graphics windows

Sprites are clipped to the edges of the screen so they appear to scroll on and off. This is handled for you by the plotting routines which only display the part of the sprite which should be on the screen. This clipping also applies to graphics windows.

DEFINING SPRITES FROM THE SCREEN

As well as using the editor to design a sprite, it is also possible to pick up any rectangular area of the screen and make this into a sprite. To do this you should

mark the rectangle by moving to its bottom left-hand corner and then its top right-hand corner. Then use the command:

```
*SGET name
```

or alternatively:

```
VDU 23,27,1,n,0,0,0,0,0,0
```

Either of these defines a sprite which contains whatever is currently in the rectangle. This sprite may then be used in exactly the same way as any other. The *SGET commands can give the sprite any name it likes, but the VDU 23 command is limited to creating sprites with names 0 to 255.

TELETEXT MODE

The teletext mode, mode 7, is unique in the way its displays text and graphics. Commands such as COLOUR, GCOL, MOVE and DRAW do not work in this mode. Instead colourful displays are produced using teletext control codes.

Mode 7 is compatible with the teletext pages broadcast by CEEFAX and Oracle. You can produce your own teletext displays using the limited but effective graphics which are available.

TEXT DISPLAYS

Coloured text

Type in the following program and run it:

```
10 MODE 7
20 PRINT "THIS";CHR$(129);"demonstrates";CHR$(130);"the";CHR$(131);"use"
30 PRINTCHR$(132);"of";CHR$(133);"control"; CHR$(134);"codes"
```

The characters 129, etc, which are printed using CHR\$(129) are the control codes. Note that the control codes are invisible but take up a character position, so the words are separated by a space. Each code affects the way in which the remaining characters on that particular line are displayed. For example, printing CHR\$(129) makes the computer display the text in red. The full list of colours and their associated control codes is given below:

Code	Text colour
129	Red
130	Green
131	Yellow
132	Blue
133	Magenta
134	Cyan
135	White (default)

Every line starts off with the text in white. So, if you want several rows of text to appear in red, for example, you must start each of these rows with CHR\$(129).

Making text flash

Text can be made to flash. For example:

```
10 MODE 7
20 PRINT CHR$(136)"Flash";CHR$(137)"Steady"; CHR$(136)"Flash"
```

Flashing coloured text can be produced by using two control codes:

```
10 MODE 7
20 PRINT "Steady white";CHR$(129);CHR$(136)"Flashing red"
```

Since each control code occupies a character position, the words `white` and `Flashing` are separated by two spaces on the screen.

Double-height text

Double-height text can be produced as follows:

```
10 MODE 7
20 PRINT CHR$(141)"Double height"
30 PRINT CHR$(141)"Double height"
```

To obtain double-height text, the same text must be printed on two successive lines beginning with `CHR$(141)`. If the text is only printed once, only the top half of the letters is displayed.

To revert back to single-height graphics on the same line, the control code is 140. For example:

```
10 MODE 7
20 PRINT CHR$(141)"Double Height";CHR$(140)"Single Height"
30 PRINT CHR$(141)"Double Height";CHR$(140)"Single Height"
```

Changing the background colour

Changing the background colour requires two codes:

```
10 MODE 7
20 PRINT CHR$(131);CHR$(157) "Hello"
```

The first code is for yellow text. The second tells the computer to use the previous control code as the background colour. The net effect of the two codes is to give yellow text on a yellow background as you can see when you run the program above. Hence to print text visibly on a coloured background, three control codes are required, two to change the background colour, and a third to change the colour of the text. For example:

```
10 MODE 7
20 PRINT CHR$(131);CHR$(157);CHR$(132) "Blue on yellow"
```

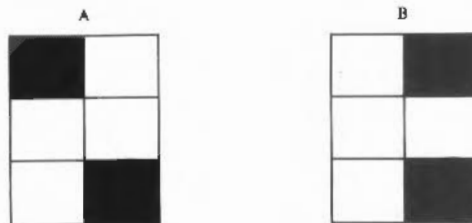
TELETEXT GRAPHICS

Certain characters, such as the lower-case letters, may either be printed normally as text or made to appear as graphics shapes by preceding them with one of the graphics control codes. These are:

Code	Graphics colour
145	Red
146	Green
147	Yellow
148	Blue
149	Magenta
150	Cyan
151	White
156	Set background to black
157	Set background colour to the current foreground colour

Each line of the Teletext display starts with the following attributes: white, alpha (ie non-graphics) characters on a black background.

Each graphics shape is based on a two by three grid:



A full list of graphics shapes is given in APPENDIX C. It is possible, however, to calculate the code for any particular graphics shape, since each of the six cells contributes a particular value to the code as follows:

1	2
4	8
16	64

The base value for the codes is 160, so that they lie in the ranges 160 to 191 and 224 to 255. For example,



has a code of $160 + 1 + 8 + 16 = 185$ and so may be produced on the screen in red. To do this, type

```
PRINT CHR$(145);CHR$(185)
```

Normally, the blocks of colours are continuous. For example,

```
PRINT CHR$(145);CHR$(255)
```

produces a solid block of red. Nevertheless, the graphics can be separated, with a thin black line around all the segments. To see the effect of this, try typing

```
PRINT CHR$(145);CHR$(154);CHR$(255)
```

So far we have seen that each of the teletext control characters appears on the screen as a space. This means that it is not normally possible to have graphics blocks of different colours touching each other. They have to be separated by at least one space to allow for the graphics colour control codes.

However, if you wish to use different colours next to each other, you can do so by using some of the more advanced teletext controls. For example, try typing

```
PRINT CHR$(145)CHR$(152)CHR$(255)CHR$(158)CHR$(146)CHR$(147)CHR$(159)
```

The code 152 conceals the display of all graphics characters until a colour change occurs. Hence the solid red graphics block is not displayed.

The code 158 holds the graphics. This means that it remembers the previous graphics character, in this case the solid block, and displays all future graphics shapes and control codes as the remembered character.

The code 146 is the first colour change. As a result, it reverses the concealing effect of code 152 so that future characters are displayed, and also selects green graphics.

The code 147 is a control code displayed as a solid graphics block in the current colour which is green. It selects yellow graphics.

The code 159 is a control code displayed as a solid graphic block in the current colour which is yellow. It releases the graphics, ie it reverses the effect of any previous 158 codes.

The computer contains a sound synthesizer which enables you to imitate up to eight different instruments playing at once, giving either mono or stereo sound production for each instrument.

The sound system can be activated or de-activated using the statements

```
SOUND ON
```

and

```
SOUND OFF
```

You need to select how many different sound channels you wish to use. The default value is one, but you can alter this by typing

```
VOICES n
```

The maximum number allowed is eight. Any number between one and eight can be specified, but the number which the computer is to handle has to be a power of two, and so the number you give is rounded up by the computer to either one, two, four or eight.

Since the sound system uses up a lot of the computer's processing power, you should minimise the number of active channels; otherwise, the computer will take longer to perform other tasks such as drawing to the screen.

For each active channel, the stereo position of the sound can be altered using:

```
STEREO chan, pos
```

`pos` can take any value between -127 (indicating the sound is fully to the left) and $+127$ (indicating the sound is fully to the right). The default value of each is zero which gives central (mono) production.

Although the range of the `pos` argument in the `STEREO` keyword is -127 to 127 , there are actually only seven discrete stereo positions. These are:

-127 to -80	Full left
-79 to -48	2/3 left
-47 to -16	1/3 left
-15 to +15	Central
+16 to +47	1/3 right
+48 to +79	2/3 right
+80 to +127	Full right

THE SOUND STATEMENT

BASIC provides a SOUND statement to create a note on any of the channels. This requires four parameters which can be summarised as follows:

SOUND channel, amplitude, pitch, duration, after

Channel

There are eight different channels, numbered 1 to 8. Each of these is identical.

Amplitude

Setting the second parameter to an integer between 0 and -15 determines how loud a note is to be played. A value of -15 is the loudest, -7 is half-volume and zero produces silence.

Alternatively, a logarithmic scale can be used, by giving a value between 256 (&100) and 383 (&17F). A change of 16 represents a doubling or halving of the volume.

Pitch

The pitch can be controlled in steps of a quarter of a semitone by giving a value between 0 and 255. The lowest note (0) is the B one octave and a semitone below middle C. The highest note is the D four octaves and a tone above middle C. A value of 53 produces middle C itself. The following table is a quick reference guide to help you find the pitch you require:

Note	Octave number					
	1	2	3	4	5	6
A		41	89	137	185	233
A#		45	93	141	189	237
B	1	49	97	145	193	241
C	5	53	101	149	197	245
C#	9	57	105	153	201	249
D	13	61	109	157	205	253
D#	17	65	113	161	209	
E	21	69	117	165	213	
F	25	73	121	169	217	
F#	29	77	125	173	221	
G	33	81	129	177	225	
G#	37	85	133	181	229	

Alternatively, a finer control is available by giving a value between 256 (&0100) and 32767 (&7FFF). Each number consists of 15 bits. The top three bits control the octave number. The bottom 12 bits control the fractional part of the octave. This means that each octave is split up into 4096 different pitch levels. Middle C has the value 16384 (&4000).

Using hexadecimal notation is a particularly useful way of seeing what level a particular value defines. Each value in hexadecimal notation is comprised of four digits. The top one gives the octave number and the bottom the fractional part of the octave. The following table illustrates this:

Note	Octave number								
	1	2	3	4	5	6	7	8	9
A		&0C00	&1C00	&2C00	&3C00	&4C00	&5C00	&6C00	&7C00
A#		&0D55	&1D55	&2D55	&3D55	&4D55	&5D55	&6D55	&7D55
B		&0EAA	&1EAA	&2EAA	&3EAA	&4EAA	&5EAA	&6EAA	&7EAA
C		&1000	&2000	&3000	&4000	&5000	&6000	&7000	
C#	&0155	&1155	&2155	&3155	&4155	&5155	&6155	&7155	
D	&02AA	&12AA	&22AA	&32AA	&42AA	&52AA	&62AA	&72AA	
D#	&0400	&1400	&2400	&3400	&4400	&5400	&6400	&7400	
E	&0555	&1555	&2555	&3555	&4555	&5555	&6555	&7555	
F	&06AA	&16AA	&26AA	&36AA	&46AA	&56AA	&66AA	&76AA	
F#	&0800	&1800	&2800	&3800	&4800	&5800	&6800	&7800	
G	&0955	&1955	&2955	&39AA	&49AA	&59AA	&69AA	&79AA	
G#	&0AAA	&1AAA	&2AAA	&3AAA	&4AAA	&5AAA	&6AAA	&7AAA	

Duration

This determines the duration of a sound. A value of 0 to 254 specifies the duration in twentieths of a second. For example, a value of 20 causes the note to sound for one second. A value of 255 causes the note to sound continuously, stopping only when you press `Escape`.

THE BEATS STATEMENT

The channels can be synchronised by using the beat counter. You can set the value that this counter will count up to by typing

```
BEATS n
```

The counter then counts from '0' to n-1 and when it reaches 'n' it resets itself to zero. To find the current beat counter value, type

```
PRINT BEATS
```

THE BEAT STATEMENT

In addition, the current beat value is found by typing

```
PRINT BEAT
```

THE TEMPO STATEMENT

The rate at which the beat counter counts depends on the tempo which can be set as follows:

```
TEMPO n
```

'n' is a hexadecimal fractional number, in which the three least-significant digits are the fractional part. A value of &1000 corresponds to a tempo of one tempo beat per centi-second; doubling the value causes the tempo to double (2 tempo beats per centi-second), halving the value halves the tempo (to half a beat per centi-second).

To find the current tempo type

```
PRINT TEMPO
```

Sounds can be scheduled to happen a given number of beats from the last reset by giving a fifth parameter to the SOUND statement. For example, the listing below repeatedly waits for the start of the 'bar, then schedules the sounds to be made after 50 beats and 150 beats respectively. Given that a bar is 200 beats long, this corresponds to the second and fourth beat of a 4/4 time:

```

10 BEATS 200
15 VOICES 2
20 *CHANNELVOICE 1 1
30 *CHANNELVOICE 2 1
40 REPEAT
50   REPEAT UNTIL BEAT=0
60   SOUND 1, -15, 100, 5, 50
70   SOUND 2, -15, 200, 5, 150
80   REPEAT UNTIL BEAT<>0
90 UNTIL FALSE

```

Notice that having scheduled the sounds, the program waits in another REPEAT loop until the current beat is not zero. This prevents the sounds from being scheduled more than once in a bar. Note also that if other things were happening in a program, such as screen updating, it would not be very safe to test for BEAT=0, in case the program missed the centi-second period where that was true. It would be better to test for something like BEAT<10 and treat beat 10 as the 'start' of the bar.

Increasing the number of beats increases the time taken before the two notes are repeated. It has no effect on the time interval between the two notes themselves.

Increasing the tempo decreases both the time taken before the two notes are repeated and the time interval between the two notes.

The after parameter

The optional 'after' parameter in the SOUND statement specifies the number of beats which should elapse before the sound is made. The beats are counted from the last time the beat counter was set to zero (ie the start of the bar). If the beat counter is not enabled (because no BEATS statement has been issued), the beats are counted from the time the statement was executed.

If the parameter is given as -1, then instead of being scheduled for a given number of beats, the sound is synchronised with the last sound which was scheduled. For example,

SOUND 1, -10, 200, 20, 100

SOUND 2, -10, 232, 20, -1

will cause two sounds, an octave apart, to be made 100 beats from the present moment, assuming that at least two channels are active and assigned voices.

KEYBOARD, MOUSE, AND FUNCTION KEYS

THE KEYBOARD

Waiting for input

We saw in the chapter: **INPUTTING INFORMATION** how a program can wait for a key to be pressed, either indefinitely using `GET` and `GET$` or for a defined length of time using `INKEY` and `INKEY$`.

Normally the keyboard allows type-ahead. Every time you press a key it is placed in the keyboard buffer which is a temporary block of memory. The `GET` and `GET$` instructions look in this buffer for a key, not at the keyboard itself. Hence they take note of keys which were pressed before the input instructions were executed.

To get around this problem, you can empty or flush the buffer before using these instructions. Then you can be sure that the key obtained is in response to the prompt and not just an accidental press of the keyboard a few moments before. To do this, use the command:

```
*FX 15,1
```

The cursor editing keys can be made to generate ASCII codes when they are pressed, rather than performing their normal cursor editing functions, by typing

```
*FX 4,1
```

The codes they return are:

Key	Code
<code>Copy</code>	135
<code>←</code>	136
<code>→</code>	137
<code>↓</code>	138
<code>↑</code>	139

The `Tab` key can be made to return any ASCII value you choose by typing


```
*FX 219,n
```

where 'n' is the ASCII code you want it to return.

The following program uses these features to move a block around the screen until **Tab** or **Copy** is pressed, and then to leave it at its current location:

```
10 MODE 1
20 *FX 4,1
30 *FX 219,135
40 x = 600 : y = 492
50 oldx = x : oldy = y
60 RECTANGLE FILL x,y,80,40
70 REPEAT
80   *FX 15,1
90   key = GET
100  CASE key OF
110    WHEN 135 : END
120    WHEN 136 : x -= 20
130    WHEN 137 : x += 20
140    WHEN 138 : y -= 20
150    WHEN 139 : y += 20
160  ENDCASE
170  RECTANGLE FILL oldx,oldy,80,40 TO x,y
180  oldx = x : oldy = y
190 UNTIL FALSE
```

Scanning the keyboard

INKEY, when it is given a positive parameter, waits for a given length of time for a particular key to be pressed; but it has an additional function. If INKEY is given a negative parameter it tests to see if a particular key is pressed at that instant. For example:

```
10 IF INKEY(-66) = TRUE PRINT "You were pressing A"
```

The list of negative values associated with each of the keys is given in APPENDIX F.

This feature is particularly useful for real-time applications where the computer is constantly reacting to the current input it is being given, rather than stopping and waiting for you to decide what to do next.

THE MOUSE

The mouse provides a convenient method of supplying information to a program. This information is in two parts:

- a position
- details of which of the buttons are currently being pressed.

To input this information, type

```
MOUSE x, y, z
```

The values returned in 'x' and 'y' give the position of the mouse. The variable 'z' gives details of the buttons currently pressed as follows:

Variable	Details
0	No buttons pressed
1	Adjust only pressed
2	Menu only pressed
3	Adjust and Menu pressed
4	Select only pressed
5	Select and Adjust pressed
6	Select and Menu pressed
7	All three buttons pressed

The following program is a very simple sketchpad program which draws lines as you move the mouse around and hold down its buttons:

```

10 MODE 12
20 MOVE 0,0
30 REPEAT
40   MOUSE x,y,button
50   GCOL button + 1
40   DRAW x,y
50 UNTIL FALSE

```

In order to be able to see the position of the mouse on the screen, it can be linked to one of four different pointers, numbered 1 to 4.

To select one of these pointers and link it to the mouse so that the mouse drives it, type

```
MOUSE ON n
```

'n' is the pointer number. If you do not specify a number, pointer number one is used.

Before you enable the mouse pointer using `MOUSE ON`, you should give it a shape. By default, all four pointer shapes are invisible, so just using the `MOUSE` statement will not appear to have any effect. The simplest way to give the pointer a shape is to use the `*POINTER` command. This makes pointer 1 into a blue arrow and displays it. Once you have used `*POINTER`, you can turn the pointer on and off using the `BASIC MOUSE` statement.

Now, whenever you move the mouse, the pointer moves with it on the screen indicating its current position. This enables the sketchpad program shown above to be altered so that you can move to the position you want and then draw a line to this new position by pressing any button:

```

5 *POINTER
10 MODE 15
20 MOVE 0,0
30 MOUSE ON
40 REPEAT
50   REPEAT

```

```
60     MOUSE x,y,button%
70     UNTIL button% <> 0
80     DRAW x,y
90     UNTIL FALSE
```

FUNCTION KEYS

The keys across the top of the keyboard labelled **f1** to **f2** are function keys. These can be programmed so that they generate any string you like when they are pressed. For example, type

```
*KEY1 "**CAT"
```

Now when you press **f1** the string *CAT is printed on the screen as though you had typed it.

Try changing the definition to:

```
*KEY1 "**CAT |M"
```

The '|' sign means that the character following it is to be interpreted as a control character. In this case it is a **Ctrl M** which is being included in the string. This performs the same function as pressing **Ctrl M**. A full list of the control characters is given in **APPENDIX H**.

Now when you press **f1**, the string *CAT is printed and **Ctrl M** is pressed automatically so the current directory is catalogued immediately.

A whole series of commands can be stored in one key. The following defines a key to select screen mode 3 and list the current program in paged mode.

```
*KEY2 "MODE 3 |M |N LIST |M"
```

You can even define a key so that it contains a small BASIC program:

```
*KEY 3 "10 MODE 15 |M 20 FOR I% = 1 TO 100|M 30 CIRCLE RND(1279),
RND(1024), 50 + RND (300) |M 40 N. |M RUN |M"
```

The quotation marks around the string are not necessary. However, it is important to remember that everything on the line after the *KEY command is treated as part of the string. So if *KEY is used in a program, it must be the last statement on the line.

The key labelled **Print** acts as function key 0. In addition, the cursor editing keys and **Copy** can be made to behave as function keys 11 to 15 by giving the command:

```
*FX 4, 2
```

Following this command, the keys, instead of having their normal cursor editing effects, return the function key strings assigned to them:

Key	*KEY number
Copy	11
←	12
→	13
↓	14
↶	15

To return them to their normal state, type

```
*FX 4, 0
```

In addition to the use of '**|**' to indicate a control code, the following are also possible in function key strings:

	means ' '
!<ch>	means the following character code + 128
?	means DELETE (ie CHR\$(127))
"	means " (useful for making " the first character)

INDIRECTION OPERATORS

ACCESSING MEMORY LOCATIONS IN GENERAL

Individual memory locations can be accessed from BASIC by using four indirection operators:

Symbol	Purpose	Number of bytes
'?'	Byte indirection operators	1
'!'	Four-byte indirection operator	4
' '	Five-byte indirection operator	5
'\$'	String indirection operator	1 to 256

These operators can be used either to read the value(s) in one or more memory locations or to alter the value(s) there. You must be very careful that you only read from or write to memory locations which you have set aside specially. Using these operators on other areas of the memory can have undesirable effects.

RESERVING A BLOCK OF MEMORY

You can reserve a block of memory using the DIM command. For example:

```
DIM pointer% 100
```

This reserves a block of memory and makes the variable `pointer%` contain the address of the first block. The bytes are at addresses `pointer%+0` to `pointer%+100`, a total of 101 bytes.

Note that this differs from the use of DIM to dimension an array in that the size is not contained in brackets.

THE '?' INDIRECTION OPERATOR

You can set the contents of the first byte of this block of memory to 63 by typing

```
?pointer% = 63
```

To check that this value has been inserted correctly, type

```
PRINT ?pointer%
```

The '?' indirection operator affects only a single byte. Only the least significant byte of the number is stored. Thus, if you give it a value of 256 or more, only the remainder of $n/256$ will be stored.

For example,

```
?pointer% = 356  
PRINT ?pointer%
```

produces the result:

```
100
```

because 356 divided by 256 gives 1 with a remainder of 100.

If you wish to set or examine the contents of the location which is five bytes after `pointer%`, you can do this by typing

```
?(pointer% + 5) = 25
```

Alternatively, a shorter form is available as follows:

```
pointer%?5 = 25
```

The following program prints out the contents of all the memory locations in the reserved block:

```
10 DIM block_of_memory% 100  
20 FOR N% = 0 TO 100  
30 PRINT "Contents of ";N%;" are ";block_of_memory%?N%  
40 NEXT N%
```

THE '!' INDIRECTION OPERATOR

BASIC integer variables are stored in four consecutive bytes of memory. The '!' operator can be used to access these four bytes. For example, type

```
DIM pointer% 100
!pointer% = 356
PRINT !pointer%
```

The bottom byte of the integer is stored in the first memory location, and the top byte in the fourth location. This can be seen in the following example:

```
10 DIM pointer% 100
20 !pointer% = &12345678
30 PRINT ~pointer%?0
40 PRINT ~pointer%?1
50 PRINT ~pointer%?2
60 PRINT ~pointer%?3
```

This prints:

```
78
56
34
12
```

THE '|' INDIRECTION OPERATOR

Similarly, floating point numbers which are stored in five consecutive bytes can be accessed using '|'. For example:

```
DIM pointer% 100
|pointer% = 3.678
PRINT |pointer%
```


THE '\$' INDIRECTION OPERATOR

Strings can be placed directly in memory, each character's ASCII code being stored in one byte of memory. For example:

```
DIM pointer% 100
$pointer% = "STRING"
PRINT $pointer%
```

The '\$' indirection operator places a carriage return (ASCII 13) after the last character of the string. Thus, the example above uses seven bytes: six for the characters of the word STRING, plus one for the terminating carriage return. To see this, run the following program:

```
10 DIM space% 10
20 REM set all bytes to zero
30 FOR N% = 0 TO 10
40   space%?N% = 0
50 NEXT N%
60 REM Store the string
70 $space% = "STRING"
80 REM Print out the bytes
90 FOR N% = 0 TO 10
100  PRINT space%?N% " ";CHR$(space%?N%)
110 NEXT N%
```

The '|' and '\$' indirection operators may only be used as unary operators, with the address following the operator. For example, although you may use \$(string+1) and |(base+offset), the forms string\$1 and base|offset are not allowed.

We are most familiar with numbers expressed in terms of powers of ten, or decimal numbers. Sometimes it is more convenient to give numbers in a program in another base. BASIC also allows numbers to be given in hexadecimal, or base 16, and binary, or base 2.

HEXADECIMAL NUMBERS

The computer treats any number which is preceded by an '&' sign as a hexadecimal (hex) number.

Whereas decimal numbers can contain ten separate digits, from 0 to 9, hexadecimal numbers can contain sixteen separate digits, 0 to 9 and A to F. The first few hexadecimal numbers and their decimal equivalents are given below:

Hex	Decimal	Hex	Decimal
&1	1	&9	9
&2	2	&A	10
&3	3	&B	11
&4	4	&C	12
&5	5	&D	13
&6	6	&E	14
&7	7	&F	15
&8	8		

The next hexadecimal number is &10 which is equivalent to 16 in decimal notation. Thus, in hexadecimal notation, one in a column represents a power of sixteen rather than a power of ten. For example, &100 represents 256 which is 16×16 .

BINARY NUMBERS AND BITS

You can enter numbers in binary notation, ie in base 2, by preceding them with the percent sign '%'.

Binary numbers consist entirely of the digits 0 and 1. The following table gives the binary equivalents of the decimal values one to 10.

Binary	Decimal	Binary	Decimal
%1	1	%110	6
%10	2	%111	7
%11	3	%1000	8
%100	4	%1001	9
%101	5	%1010	10

A one in a particular column represents a power of two:

... 128 64 32 16 8 4 2 1

Thus:

$$\%1000101 = 1*64 + 0*32 + 0*16 + 0*8 + 1*4 + 0*2 + 1*1 = 69$$

Binary digits are usually referred to as bits.

Shift operators

There are three operators which act upon the 32 bits of an integer, shifting them either left or right by a given number of places.

The simplest of these is <<. This shifts the bits of an integer to the left a given number of times and inserts '0's in the right-hand bits. For example:

```
A% = 10
B% = A% << 1
C% = A% << 2
D% = A% << 3
```

This leaves the variables with the following values:

This leaves the variables with the following binary values:

Variable	Value
C%	%01010000000000000000000000000000
D%	%00010000000000000000000000000000
E%	%11010000000000000000000000000000
F%	%00010000000000000000000000000000

The shift operators may be used for multiplication and division by powers of two. The left shift operator acts as a multiply. The expression `val<<n` is equivalent to `val * 2^n`. So `fred<<3` is the same as `fred*8`. Although using shift can be faster than the equivalent multiply, you should bear in mind that bits may be shifted off the end of the number, so leading to incorrect results which will not be trapped as errors. For example, `&1000<<16` yields 0, whereas the correct 'multiply' result is `&100000000` (which cannot be represented in a 32-bit integer).

The two right shift operators perform a similar role in division. The `>>` operator gives division of 'signed' numbers by a power of two. This means that both positive and negative numbers may be divided; the result is always rounded towards the integer less than or equal to the exact value. For example, `-3>>1` is the same as `INT(-3/2)` (not `-3 DIV 2`), which is `-2`. The `>>>` operator ignores the sign when shifting negative numbers, so should only be used to divide positive numbers by a power of two.

AND, OR and EOR

The operators AND, OR and EOR produce a result which depends upon the bits of two other integers:

- In the case of AND, the bits in the two integers are compared and if they are both one, then a one is placed in the corresponding bit of the result.
- In the case of OR, a one is placed in the corresponding bit of the result if either or both of the bits in the integers are one.

- In the case of EOR, a one is placed in the corresponding bit of the result if either (but not both) of the bits in the integers is one.

For example:

A% = %1010
 B% = %1100
 C% = A% AND B%
 D% = A% OR B%
 E% = A% EOR B%

This leaves the variables with the following values:

Variable	Value
A%	10 (%1010)
B%	12 (%1100)
C%	8 (%1000)
D%	14 (%1110)
E%	6 (%0110)

The logical operators AND, OR and EOR are symmetrical, like '+' and '*'. Thus $x \text{ AND } y = y \text{ AND } x$ for all possible values of 'x' and 'y'. This applies to the other two operators as well.

TRUE and FALSE

The truth values TRUE and FALSE have the values -1 and zero respectively. This means that:

- TRUE AND TRUE gives TRUE (-1 AND -1 = -1)
TRUE AND FALSE gives FALSE (-1 AND 0 = 0)
FALSE AND FALSE gives FALSE (0 AND 0 = 0)

- TRUE OR TRUE gives TRUE (-1 OR -1 = -1)
TRUE OR FALSE gives TRUE (-1 OR 0 = -1)
FALSE OR FALSE gives FALSE (0 OR 0 = 0)

- TRUE EOR TRUE gives FALSE (-1 EOR -1 = 0)
TRUE EOR FALSE gives TRUE (-1 EOR 0 = -1)
FALSE EOR FALSE gives FALSE (0 EOR 0 = 0)

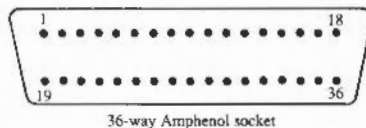
CONNECTING YOUR PRINTER

This computer contains two different sockets to which printers can be connected. These are:

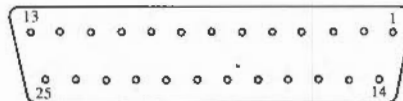
- Parallel printer (Centronics-compatible) port
- Serial printer (RS232/423) port

Printers which should be connected to the parallel ports usually have a 36-way Amphenol socket. A suitable lead to connect a parallel printer to your computer is available.

The standard connector plug for serial printers is a Cannon 25-way D-type. This needs to be connected to the 9-way D-type connector on the computer.

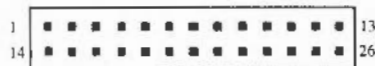


36-way Amphenol socket



25-way D-type socket

Connects to a 9-way D-type plug



Connects to a 26-way plug

Refer to your printer documentation for details of connections. Your dealer should be able to supply a cable.

Refer to **Appendix L** for details of the computer connections. Please note that all connections must be used.

DEFINING THE PRINTER TYPE

Once you have connected your printer, you need to tell the computer which of the ports it is connected to so that the computer knows where to send its printer output.

If you are using a parallel printer, type

```
*FX 5,1
```

If you are using a serial printer, type

```
*FX 5,2
```

To select a printer on an Econet system, type

```
*FX 5,4
```

You can configure your machine so that it remembers which type of printer you are using when you turn it on each time by typing, for example:

```
*CONFIGURE PRINT 1
```

SELECTING THE BAUD RATE

The computer defaults to sending characters at 9600 baud. However, some serial printers run at other rates. If your printer expects to receive characters at a different rate, then you should select the one required as follows:

*FX 8,1	75 baud
*FX 8,2	150 baud
*FX 8,3	300 baud
*FX 8,4	1200 baud
*FX 8,5	2400 baud
*FX 8,6	4800 baud
*FX 8,7	9600 baud
*FX 8,8	19200 baud

You can configure the baud rate that is used by default using the command:

```
*CONFIGURE Baud <n>
```

The number given in the command has the same meaning as that in the *FX8 command, with the addition that 0 means 9600 baud.

You can also configure the default ignore character. When the configurations are reset, the default ignore character is set to 10. To change it, use

```
*CONFIGURE ignore <n>
```

where <n> is the ASCII code of the character which you want to be suppressed by default. If you want all characters to be sent to the printer, omit the number from the command. As with all configure options, this will not come into effect until the next **Ctrl|Break** or power-on.

PRINTER IGNORE CHARACTERS

The final item which should be selected before starting to print anything is the printer ignore character. This specifies a particular character which the computer is to ignore and not send to the printer.

At the end of each line, the computer sends a carriage return (ASCII 13) and linefeed (ASCII 10). However, some printers automatically move the paper up by one line when they receive a carriage return. Normally on these printers, the paper would move up two lines instead of just one. The printer ignore character can, however, be set to be the linefeed character to get around this problem. Type

*IGNORE 10

To make the computer send all characters type

*IGNORE

SENDING OUTPUT TO THE PRINTER

When you are ready to send output to the printer, you should enable it by pressing **Ctrl Break**, or by typing

VDU 2

All output after this is sent to both the screen and the printer.

The printer can be disabled again by pressing **Ctrl C**, or by typing

VDU 3

ERROR HANDLING AND DEBUGGING

GLOBAL ERROR HANDLING

By default, when the computer finds an error it halts execution of the program and prints an error message on the screen. Some errors are generated by incorrect programming, such as using a variable which has not had a value assigned to it. You have to correct this kind of error to make the program work. However, even if the syntax of the program is correct, errors can occur whilst it is being executed, because it cannot cope with the data it is given. For example:

```
10 REPEAT
20   INPUT "Number",N
30   L = LOG(N)
40   PRINT "LOG of ";N" is ";L
50 UNTIL FALSE
```

This program takes a number from the keyboard and prints the log of that number. If you type in a negative number, however, the program gives the message:

```
Logarithm range at line 30
```

The same thing happens if you type '0', or a character such as 'w', or a word such as TWELVE.

You may decide that you would like to trap such an error and print a message to tell the user what he or she has done wrong instead of having the program end abruptly. You can do this using the ON ERROR statement. For example:

```
5 ON ERROR PROCerror
10 REPEAT
20   INPUT "Number",N
30   L = LOG(N)
40   PRINT "LOG of ";N" is ";L
50 UNTIL FALSE
60 END
100 DEFPROCerror
110 IF ERR=22 THEN
```

```
120 PRINT "The number must be greater than 0"  
130 ELSE REPORT  
140 PRINT " at line ";ERL  
150 END  
160 ENDIF  
170 ENDPROC
```

The ON ERROR statement can be followed by a series of statements given on the same line. In many cases, it is more convenient to follow it with a call to an error handling procedure, as in the example above, which can then be as complex as you like.

Each error has an error number associated with it. When a particular error occurs, its number is placed in a variable called ERR. A full list of error numbers is given in APPENDIX B.

In the example above, the error handling procedure tests for error 22 which is the Logarithm range error. If it was this error which occurred, it is dealt with appropriately. If a different error occurred, the program executes the REPORT instruction which prints out the error message and then prints the line number where the error occurred which is stored in the variable ERL. Then it executes the END to end the execution of the program. Trapping all errors is not necessarily a good idea since you then would not be able to press Escape, which is treated as an error, to stop the program.

If a program contains more than one ON ERROR statement, the most recently executed one is used when an error occurs.

Error handling can be turned off at any stage in the program using the instruction ON ERROR OFF.

LOCAL ERROR HANDLING

When an error occurs, the ON ERROR command can be used to deal with it. The computer, however, forgets all about what it was doing at the time the error happened. For example, if it was in the middle of a FOR ... NEXT loop or

executing a procedure, it is not possible to jump back to the place the error occurred and carry on as though nothing had happened.

The ON ERROR LOCAL command can be used to get around this problem. This command traps errors which occur inside an individual procedure or function and then continues executing within the procedure or function rather than jumping back to the top level. For example:

```
10 PROCcalculate(100)
20 END
100 DEFPROCcalculate(A)
110 LOCAL I
130 FOR I = -15 TO 15
140   ON ERROR LOCAL PRINT"Infinite Result":NEXT I:ENDPROC
150   PRINT A / I
160 NEXT I
180 ENDPROC
```

Normally, when one ON ERROR or ON ERROR LOCAL statement is used, all previous ERROR statements are forgotten about, but it is possible to use one error handler and then restore the previous one. To do this, use the instruction LOCAL ERROR to store the old error handler, and RESTORE ERROR to activate it again. For example:

```
1 ON ERROR PRINT "Error ";REPORT$;:END
10 PROCcalculate(100)
15 this line will give an error !!!
20 END
100 DEFPROCcalculate(A)
110 LOCAL I
120 LOCAL ERROR
130 FOR I = -15 TO 15
140   ON ERROR LOCAL PRINT"Infinite Result":NEXT I:ENDPROC
150   PRINT A / I
160 NEXT I
170 RESTORE ERROR
180 ENDPROC
```

This shows that the local error handler is in force during the procedure, but that the original one set up by the first line of the program is restored when the PROC has finished.

Strictly speaking, the RESTORE ERROR is not required here because it is done automatically when the ENDPROC is reached.

DEBUGGING

A program may contain errors which cause it to behave differently from the way you intended. In these circumstances, you may wish to watch more closely how the program is being executed.

One option you have available is to place a STOP statement at a particular point in the program. When this line is reached, execution of the program stops and you can then investigate the values assigned to any of its variables using the PRINT statement.

Another option is to use the TRACE facility. The standard trace prints the BASIC line numbers in the order these lines are executed, thus showing the path being taken through the program. This can be invoked by typing

```
TRACE ON
```

To trace only those lines with a line number below 1000, for example, type

```
TRACE 1000
```

Alternatively you may trace procedures only as follows:

```
TRACE PROC
```

Tracing can be performed in single-step mode where the computer stops after each line or procedure call and waits for a key to be pressed before continuing. Single-step tracing can be invoked by typing

```
TRACE STEP ON
```

to stop after every line traced, or

```
TRACE STEP n
```

to trace all lines below 'n' and stop after each one, or

```
TRACE STEP PROC
```

to stop after every procedure call. Any TRACE option affects all programs which are subsequently run until tracing is turned off by typing

```
TRACE OFF
```


BASIC KEYWORDS

Complete descriptions of the statements, commands and functions that BASIC understands follow, in alphabetical order.

If you have a ROM version of BBC BASIC, you can obtain a list of the keywords on the screen by typing

HELP

ABS

Function giving magnitude of its numeric argument.

Syntax

ABS<factor>

Argument

Any numeric.

Result

Same as the argument if this is positive, or -(the argument) if it is negative.

Note

The largest negative integer does not have a legal positive value, so that if $a\% = -2147483648$, ABS (a%) yields the same value: -2147483648.

Example

```
diff=ABS (length1-length2)
```

ACS

Function giving the arc-cosine of its numeric argument.

Syntax

ACS<factor>

Argument

Real or integer between -1 and 1 inclusive.

Result

Real in the range 0 to PI (radians)

Examples

```
ang = ACS(normvec1(1)*normvec2(1) + normvec1(2)*normvec2(2))
angle=DEG(ACS(cos1)) : PRINT ACS(0.5)
```

ADVAL

Function reading data from an analogue port if fitted, or giving buffer data.

Syntax

ADVAL<numeric factor>

Argument

Negative integer -n, where 'n' is a buffer number between 1 and 9.

Result

The number of spaces or entries in the buffer is given in the table below:

n	Buffer name	Result
-1	Keyboard (input)	Number of characters used (0-31)
-2	RS-423 (input)	Number of characters used (0-255)
-3	RS-423 (output)	Number of characters free (0-191)
-4	Printer (output)	Number of characters free (0-63)
-5	Sound 0 (output)	Number of bytes free (0-15, step 3)
-6	Sound 1 (output)	Number of bytes free (0-15, step 3)
-7	Sound 2 (output)	Number of bytes free (0-15, step 3)
-8	Sound 3 (output)	Number of bytes free (0-15, step 3)

In the table step 3 means that one entry in the buffer uses three bytes.

The ADVAL function only returns a result for positive arguments if the optional analogue/digital converter module is fitted. If this is absent, the function ADVAL (1), for example, will result in a Bad command error.

Example

```
IF ADVAL(-1)=0 THEN PROCinput
```

AND

Operator giving logical or bitwise AND.

Syntax

```
<relational> AND <relational>
```

Operands

Relational expressions, or bit values to be ANDed.

Result

The logical bitwise AND of the operands. Corresponding bits in the integer operands are ANDed to produce the result. Hence a bit in the result is one if both of the corresponding bits of the operands are one. Otherwise it is zero.

If used to combine relational values, AND operands should be either TRUE (-1) or FALSE (0). Otherwise, unexpected results may occur. For example, 2 and 4 are both true: non-zero, but 2 AND 4 yields FALSE: zero.

Examples

```
a = x AND y : REM a is set to binary AND of x and y

PRINT variable AND 3 : REM print lowest 2 bits of variable

IF day=7 AND month$="March" THEN PRINT "Happy birthday"

IF temp>50 AND NOT windy THEN PROCgo_out ELSE PROCstay_in

REPEAT
a=a+1
b=b-1
UNTIL a>10 AND b<0

isadog = feet=4 AND tails=1 AND hairy
REM set isadog to logical true if all conditions are met
```

APPEND

Command to append a file to a BASIC program.

Syntax

```
APPEND <expression>
```

Argument

<expression> is a string which should evaluate to a filename that is valid for the filing system in use.

Purpose

The file specified is added to the end of the BASIC program currently in memory. If the file contains a BASIC program, the line numbers and any references to them in the added section are renumbered so that they start after the last line of the current program.

Examples

```
APPEND ":0.lib"
```

```
APPEND second_half$
```

ASC

Function giving the ASCII code of the first character in string.

Syntax

```
ASC<factor>
```

Argument

String of 0 to 255 characters.

Result

ASCII code of the first character of the argument in the range 0 to 255, or -1 if the argument is a null string.

Examples

```
x2=ASC(name$)
```

```
IF code >= ASC("a") AND code <= ASC("z") THEN PRINT "Lower case"
```

ASN

Function giving the arc-sine of its numeric argument.

Syntax

```
ASN<factor>
```

Argument

Numeric between -1 and 1 inclusive.

Result

Real in the range $-\pi/2$ to $+\pi/2$ radians.

Examples

```
PRINT ASN(opposite/hypotenuse)
```

```
angle=DEG(ASN(0.2213))
```

ATN

Function giving the arc-tangent of its numeric argument.

Syntax

```
ATN<factor>
```

Argument

Any numeric.

Result

Real in the range $-PI/2$ to $+PI/2$ radians.

Examples

```
ang = DEG(ATN(sin/cos))
```

```
PRINT "The slope is ";ATN(opposite/adjacent)
```

AUTO


Command initiating automatic line numbering.

Syntax


```
AUTO [<integer>] [,<step>]
```

Parameters

<integer> is an integer constant in the range 0 to 65279 and is the first line to be generated automatically

<step> is an integer constant in the range 1 to 65279 and is the amount by which the line numbers increase when  is pressed. If omitted, 10 is assumed.

Purpose

AUTO is used when entering program lines to produce a line number automatically, so that you do not have to type them yourself. To end automatic line numbering use . AUTO will stop if the line number becomes greater than 65279.

Examples

AUTO

AUTO 1000

AUTO 12, 2

BEAT

Function returning the current beat value.

Syntax

BEAT

Result

An integer giving the current beat value. This is the value yielded by the beat counter as it counts from zero to the number set by BEATS at a rate determined by TEMPO. When it reaches its limit it resets to zero. Synchronisation between sound channels is performed in relation to the last reset of the beat counter.

Example

PRINT BEAT

BEATS

Function returning or statement altering the beat counter.

Syntax

(1) BEATS <expression>

(2) BEATS

Arguments (1)

<expression> gives the value beyond which the beat counter is to reset. This counter is used in conjunction with the SOUND and TEMPO statements to synchronise sound outputs from different sound channels.

Result (2)

An integer giving the current value of the beat counter.

Examples

```
BEATS 2000 PRINT BEATS
```

BGET #

Function returning the next byte from a file.

Syntax

```
BGET#<factor>
```

Argument

A channel number returned by an OPEN function.

Result

The ASCII code of the character read (at position PTR#) from the file, in the range 0 to 255.

Note

PTR# is updated to point to the next character in the file. If the last character in the file has been read, EOF# for the channel will be TRUE. The next BGET# will return an undefined value and the one after that will produce an EOF error.

Examples

```
char%=BGET#(channel)  char$=CHR$(BGET#fileno)

WHILE NOT EOF#(channel)
char% = BGET#(channel)
PROCprocess(char%)
ENDWHILE
```

BPUT

Statement to write a byte to a file.

Syntax

- (1) BPUT#<factor>,<numeric expression>
- (2) BPUT#<factor>,<string expression>

Arguments (1)

<factor> is a channel number as returned by an OPEN function. The <numeric expression> is truncated to an integer 0 to 255, and is the ASCII code of the character to be sent to the file.

Arguments (2)

<factor> is a channel number as returned by an OPEN function. <string expression> is a string containing 0 to 255 characters. The ASCII codes of all the characters in the string are sent to the file followed by a newline (ASCII value 10), unless the statement is terminated by a ';'.

Note

PTR# is updated to point to the next character to be written. If the end of the file is reached, the length (EXT#) increases too. It is only possible to use BPUT# with OPENUP and OPENOUT files, not OPENIN ones.

Examples

```
BPUT#outputfile,byte%
```

```
BPUT#channel,ASC(mid$(name$,pos,1))
```

```
BPUT#file,"Hello"
```

```
BPUT#chan,A$+B$
```

BY

Part of the MOVE BY, DRAW BY, POINT BY, or FILL BY statements.

Syntax

- (1) MOVE BY <expression>,<expression>
- (2) POINT BY <expression>,<expression>
- (3) DRAW BY <expression>,<expression>
- (4) FILL BY <expression>,<expression>

Arguments (1)

The <expression>s are 'x' and 'y' offsets from the graphics cursor in the range -32768 to 32767, ie two-byte integers. This is equivalent to PLOT 0.

Arguments (2)

The <expression>s are integer numerics in the range -32768 to +32767. They are offsets from the graphics cursor at which the point will be placed. The point is plotted in the current graphics foreground colour and the graphics cursor is updated to these coordinates. This is equivalent to PLOT 65.

Arguments (3)

The <expression>s are integer numerics in the range -32768 to +32767. They are the offsets from the current graphics cursor to which a line is drawn in the

current graphics foreground colour. The graphics cursor position is updated to these coordinates. DRAW BY is equivalent to PLOT 1.

Arguments (4)

This is equivalent to PLOT &81

Examples

MOVE BY 4*x%, 4*y%

POINT BY 100, 0

DRAW BY x%*16, y%=4

FILL BY x%, y%

CALL

Statement to execute a machine code subroutine.

Syntax

CALL<expression> [, <variable>] etc

Arguments

<expression> is the address of the routine to be called. The zero or more parameter variables may be of any type, and must exist when the CALL statement is executed. They are accessed through a parameter block which BASIC sets up. The format of this parameter block and of the variables accessed through it is described below.

Purpose

CALL can be used to enter a machine code program from BASIC. Before the routine is called, the assembler's registers are set up as follows:

R0	A%
R1	B%
R2	C%
R3	D%
R4	E%
R5	F%
R6	G%
R7	H%
R8	Pointer to BASIC's workspace
R9	Pointer to list of L-values of the parameters
R10	Number of parameters
R11	Pointer to BASIC's string accumulator (STRACC)
R12	BASIC's LINE pointer (points to the current statement)
R13	Pointer to BASIC's stack
R14	Link back to BASIC and environment information pointer.

Format of the CALL parameter block

R9 contains a list giving details of each variable which is passed as a parameter to CALL. For each variable two word-aligned words are used. The first one is the L-value of the parameter. This is the address in memory in which the value of the variable is stored. The second is the type of variable. This list is in reverse order. The pointer to the list is always valid, even when if the list is null. The values for the types are as follows:

BASIC	Type	Address points to
?name	0	byte-aligned byte
!name	4	byte-aligned word
name%	4	word-aligned word
name%(n)	4	word-aligned word
name	5	byte-aligned 5 bytes
name	5	byte-aligned 5 bytes
name(n)	5	byte-aligned 5 bytes
name\$	128	byte-aligned 5 bytes (address of string and length)
name\$(n)	128	byte-aligned 5 bytes (address of string and length)
\$fred	129	byte-aligned bytes, terminated by ASCII 13
name%()	256+4	word-aligned word pointing to array of name%
name()	256+5	word-aligned word pointing to array of name
name\$()	256+128	word-aligned word pointing to array of name\$

For types zero, four and five, the address points to the actual byte, four-byte integer or five-byte floating point value.

For types 128, the address points to five bytes which contain a byte-aligned word pointing to the first character of the string which is on a word boundary, followed by a byte-aligned byte containing the length of the string.

For types 256+'n' the value points to a word-aligned word. If the array has not been allocated, this word contains zero, signifying an undimensioned array. Otherwise, the word points to a word-aligned list of integer subscript limits (the values in the DIM statement plus 1) terminated by a word of zero, followed by a word which contains the total number of entries in the array followed by the zeroth element of the array.

For example, suppose the statements

```
100 DIM a%(10,20)
200 CALL code, a%()
```

were executed. The type at address $[R9+4]$ would be $256+4$ or 260. The word at $[R9]$ would contain a pointer to a word. The contents of this word would be the address of the subscript list, of the form:

```
word+0      11
word+4      21
word+8      0
word+12     231
word+16     a%(0,0)
```

The access method into the arrays is given by the following algorithm:

```
position = 0
number = 0
REPEAT
IF subscript(number) > array(number) THEN fault
number = number+1
IF number<>total THEN position = (position+subscript)*array(number)
UNTIL no_more_subscripts
position = position*size(array)
```

This means that the last subscript references adjacent elements. For a simple two dimensional array DIM A (LIMI-1, LIMJ-1) the address of A (I, J) is $(I*LIMI+J)*size+base_address$.

MOV PC, R14 returns to the value R14, but it also points to an array of useful values:

B CALL2REAL	0th entry in table is return address the following values are words containing an offset from ARGP (R8) word-aligned 256 bytes
& STRACC	string accumulator word-aligned words offset from ARGP
& PAGE	current program PAGE
& TOP	current program TOP
& LOMEM	current variable start

& HIMEM	current stack end
& MEMLIMIT	limit of available memory
& FSA	free space start (high water mark/FD stack limit)
& TALLY	value of COUNT
& TIMEOF	not used
& ESCWORD	exception flag word (contains <code>escflg</code> , <code>trcflg</code>) byte-aligned bytes offset from ARGP
& WIDTHLOC	value of WIDTH internal BASIC routines
B VARIND	get value of L-value
B STOREA	store value into L-value
B STSTORE	store string into type 128 strings
B LVBLNK	convert string variable name to L-value address and type
B CREATE	create new variable
B EXPR	use expression analyser on string
B MATCH	lexical analyse source string to destination string
B TOKENADDR	pointer to string for given token
& 0	end of list

The word at address [R14] is a branch instruction which returns you to the BASIC interpreter. The following words contain useful addresses which are not absolute, but are offsets from the contents of the ARGP register, R8.

The first offset word, at [R14+4], gives the location of the string accumulator, STRACC, where string results are kept. Thus if you execute

```
LDR  R0, [R14, #4]
ADD  R0, R8, R0
```

R0 will give the base address of the string accumulator. The offsets from PAGE to WIDTHLOC give the addresses, when added to R8, of useful word-size quantities. After these come the branches useful to BASIC routines. For example, to call STOREA, which is at offset 42 from R14, you might use:

```

        STMFD R13!,R14      ;Save BASIC return address
        LDR   R0,[R14,#42];Get address of B STOREA
        ADR   R14,myReturn;Set up return address to my code
        MOV   PC,R0        ;Do the 'branch'
.myReturn
        ....
        LDMFD R13!,PC      ;Return to BASIC

```

The internal routines are only guaranteed to work in processor user mode. The following functions are provided:

VARIND

Entry with R0:

```

R0    =    address of L-value
R9    =    type of L-value
R12   =    LINE

```

Returns with R0..R3 as the value, R9 the type of the value as follows:

R9	type	where
0	string	in STRACC, R2 points to end (R2-STRACC is length)
&40000000	integer	in R0
&80000000	float	in R0..R3

Uses no other registers (including stack). Possible error if asked to take value of an array `fred()`: will need R12 valid for this error to be reported correctly.

STOREA

Entry with R0..R3 value

R4 = address of L-value
R5 = type of L-value
R8 = ARGP
R9 = type of value
R12 = LINE (for errors)

Converts between various formats, for example integer and floating point numbers, or produces an error if conversion is impossible.

Returns with R0 to R7 destroyed. Stack not used.

STSTORE

Entry with:

R2 = length (address of end)
R3 = address of start
R4 = address of L-value
R8 = ARGP
R12 = LINE (for out of store error)

The string must start on a word boundary and the length must be 255 or less.

Uses R0, R1, R5, R6, R7. Preserves input registers. Stack not used.

LVBLNK

Entry with:

R8 = ARGP
R11 = pointing to start of string
R12 = LINE (many errors possible, such as subscript error in array)
R13 = stack (used for evaluation of subscript list: calls EXPR)

The string is processed to read one variable name and provide an address and type which can be given to VARIND.

Returns with NE status if a variable has been found. Address in R0, type (see above) in R9. If there is an EQ status then if the carry is set it cannot possibly be a variable, or else if the carry is clear it could be, but is not known to the interpreter (and registers are set to values for CREATE).

Uses all registers.

CREATE

Create a variable. Input is the failure of LVBLNK to find something. Thus we have:

- R3 = second char of item or 0
- R4 = pointers to start of other chars
- R8 = ARGV
- R9 = contains the number of zero bytes on the end
- R10 = first char of item
- R11 = pointers to end of other chars
- R12 = LINE
- R13 = STACK

It is recommended that CREATE is only called immediately after a failed LVBLNK.

Uses all registers. Returns parameters as LVBLNK. The LVBLNK and CREATE routines can be combined together to provide a routine which checks for a variable to assign to, and creates it if necessary:

```
.SAFELV STMFD SP!,R14
    BL LVBLNK
    LDMNEFD SP!,PC
    LDMCSFD SP!,PC
    BL CREATE
    LDMFD SP!,PC
```

EXPR

Entry with:

R8	=	ARGP
R11	=	pointing to start of string
R12	=	LINE
R13	=	STACK

EXPR stops after reading one expression (like those in the PRINT statement).

The value is returned like VARIND. If there is an EQ status it reads a string. If there is a NE status and plus it reads an integer word (in R0). If there is a NE status and minus it reads a floating point value (in R0..R3). R9 contains the type: the status can be recreated by TEQ R9, #0. R10 contains the delimiting character; R11 points to the one after.

MATCH

Entry with:

R1	=	pointing to the source string (terminated by ASCII 13 <CR>),
R2	=	pointing to the destination string
R3	=	MODE
R4	=	CONSTA
R13	=	STACK

Note that MATCH does not need ARGP or LINE.

The MODE value is 0 for LEFT-MODE (before an '=' sign) and 1 for RIGHT-MODE (in an expression).

The CONSTA value is 0 for do not pack constants using &8D, 1 for PACK.

Both MODE and CONSTA will be updated during the use of the routine. For example, GOTO will set CONSTA to 1 to read the line number, PRINT will change

MODE to 1 to read an expression. Starting values of zero for both will lexically analyse a statement:

MODE=1 and CONSTA=0	will analyse an expression
MODE=0 and CONSTA=1	is used to extract line numbers in command mode.

MODE affects the values assigned to tokens for HIMEM, PAGE etc.

The routine uses R0 to R5.

On exit, R1 and R2 are left pointing one byte beyond the terminating CR codes of the strings. R5 contains status information, it can usually be disregarded: values \geq &1000 imply mismatched brackets, bit8 set implies that a number which was too large to be encoded using &8D (ie was greater than 65279) was found. If R5 AND 255 = 1 then mismatched string quotes were found.

TOKENADDR

Entry:

R0	=	the token value
R12	=	pointer to next byte of token string.

The value of R12 is only used when two byte tokens are required. No other registers are used or required.

Returns R1 as a pointer to the first character of the string, terminated by a character \geq &7F (Note that &7F is a valid token!). R0 is set to the address of the start of the token table itself. R12 will have been incremented by 1 if a two byte token has been used.

If the CALL statement is used with an address which corresponds to a MOS entry point on the BBC Micro/Acorn Electron/Master 128 series machines and there are no other parameters, then BASIC treats the call as if it had been made from one of those machines. The way in which the registers are initialised is then as follows:

```
R0    A%
R1    X%
R2    Y%
Carry C%
```

This means that programs written to run on earlier machines using legal MOS calls can continue to work. For example, the sequence

```
10  osbyte=&FFF4
1000 A%=138
1010 X%=0
1020 Y%=65
1030 CALL osbyte
```

will be interpreted as the equivalent `SYS OS_Byte` call:

```
1000 SYS "OS_Byte",138,0,65
```

This facility is provided for backwards compatibility only. You should not use it in new programs. Also, you must be careful that any machine code you assemble in a program does not lie in the address range `&FFCE` to `&FFF7`; otherwise when you call it, it might be mistaken for a call to an old MOS routine.

CASE

Statement marking the start of an `CASE ... OF ... WHEN ... OTHERWISE ... ENDCASE` construct. It must be the last statement on the line.

Syntax

```
CASE <expression> OF
```

Arguments

<expression> can be any numeric or string expression. The value of <expression> is checked against the values of each of the expressions in the list following the first `WHEN` statement. If a match is found, then the block of

statements following the `WHEN` down to either the next `WHEN` or `OTHERWISE` or `ENDCASE` is executed. Then control moves on to the statement following the `ENDCASE`. If there is no match, then the next `WHEN` is used, if it exists. `OTHERWISE` is equivalent to a `WHEN` but matches any value.

Examples

```
CASE A$ OF
```

```
CASE Y*2 + X*3 OF
```

```
CASE GET$ OF
```

CHAIN

Statement to load and run a BASIC program.

Syntax

```
CHAIN<expression>
```

Argument

<expression> should evaluate to a string which is a valid filename for the filing system in use.

Notes

A filing system error may be produced if, for example, the file specified cannot be found. When the program is loaded, all existing variables are lost (except the system integer variables and installed libraries).

Examples

```
CHAIN "partB"
```

```
CHAIN a$+"2"
```


CHR\$

Function giving the character corresponding to an ASCII code.

Syntax

```
CHR$<factor>
```

Argument

An integer in the range zero to 255

Result

A single-character string whose ASCII code is the argument.

Examples

```
PRINT CHR$(code)
```

```
lower$=CHR$(ASC(upper$) OR &20)
```

CIRCLE

Statement to draw a circle.

Syntax

- (1) CIRCLE <expression1>,<expression2>,<expression3>
- (2) CIRCLE FILL<expression1>,<expression2>,<expression3>

Arguments

<expression1>,<expression2> and <expression3> are integer numerics in the range -32768 to +32768. The first two values give the 'x' and 'y' coordinates of the centre of the circle. The third gives the radius. CIRCLE produces a circle outline, whereas CIRCLE FILL plots a solid circle.

Note

In both cases, the position of the graphics cursor is updated to lie at the radial point: at the position on the circumference which has an 'x' coordinate of <expression1> + <expression3> and a 'y' coordinate of <expression2>. The 'previous graphics cursor' position (as used by, for example, triangle plotting) will be updated to lie at the centre of the circle plotted.

Examples

```
CIRCLE 640,512,50
```

```
CIRCLE FILL RND(1278),RND(1022),RND(200)+50
```

CLEAR

Statement to remove all program variables.

Syntax

```
CLEAR
```

Purpose

When this statement is executed, all variables are removed and so become undefined. In addition, any currently active procedures, subroutines, loops, and so on are forgotten. The exceptions to this are the system variables and installed libraries which still remain.

CLG

Statement to clear the graphics window to the graphics background colour.

Syntax

```
CLG
```

Notes

This statement also moves the graphics cursor back to the graphics origin (0,0).

Examples

```
CLG
```

```
MODE 1
```

```
GCOL 2
```

```
VDU 24,200;200;1080;824;
```

```
CLG
```

CLOSE

Statement to close an open file.

Syntax

```
CLOSE#<factor>
```

Argument

A channel number as returned by the `OPEN` function. If zero is used all open files on the current filing system are closed. Otherwise, only the file with the channel number specified is closed.

Purpose

Closing a file ensures that its contents are updated on whatever medium is being used. This is necessary as a certain amount of buffering is used to make the transfer to data between computer and mass-storage device more efficient. Closing a file, therefore, releases a buffer for use by another file.

Examples

```
CLOSE#indexFile
```

```
CLOSE#0 : REM This closes all open files on the current filing system.
```

CLS

Statement to clear the text window to the text background colour.

Syntax

```
CLS
```

Notes

CLS also resets COUNT to zero and moves the text cursor to its home position, which is normally the top left of the screen.

Examples

```
CLS
```

```
MODE 1
COLOUR 129
VDU 28, 4, 28, 35, 4
CLS
```

COLOUR (COLOR)

Statement to set the text colours or alter the palette settings.

Syntax

- (1) COLOUR<expression> [TINT <expression1>]
- (2) COLOUR<expression>, <expression>
- (3) COLOUR<expression>, <expression>, <expression>, <expression>

Arguments (1)

<expression> is an integer in the range zero to 255. The range zero to 127 sets the text foreground colour. Adding 128 to this (ie 128 to 255) sets the text background colour. The colour is treated MOD the number of colours in the current mode. The argument is the logical colour. For a list of the default logical colours, see the chapter: **SCREEN MODES**.

The optional **TINT** is only effective in 256-colour modes. It selects the amount of white to be added to the colour. The value can lie in the range 0 to 255, with only the values 0, 64, 128 and 192 currently being used to obtain different whiteness levels.

Arguments (2)

The first expression is an integer in the range zero to 15 giving the logical colour number. The second expression is an integer in the range zero to 15 giving the actual colour to be displayed when the logical colour is used. The actual colour numbers correspond to the default colours available in 16-colour modes: eight steady colours and eight flashing colours. The colour list is given in the chapter: **SCREEN MODES**.

Arguments (3)

The first expression is an integer in the range zero to 15 giving the logical colour number. The next three expressions are integers in the range zero to 255 giving the amount of red, green and blue which are to be assigned to that logical colour. Only the top four bits of each are relevant.

Notes

The keyword is listed as **COLOUR** in programs, even if it was typed in using the alternative spelling.

In all modes the default state, before any changes to the palette, dictates that colour 0 is black and colour 63 is white.

Note that only colours zero and one are unique in two-colour modes. After that the cycle repeats. Similarly, only colours zero, one, two and three are distinct in the four-colour modes.

Examples

```
COLOUR 128+1 : REM Sets background colour to colour one
```

```
COLOUR 1,5 : REM Sets logical colour one to actual colour five (magenta)
```

```
COLOUR 1,255,255,255 : Sets logical colour one to white
```

COS

Function giving the cosine of its numeric argument.

Syntax

```
COS<factor>
```

Argument

<factor> is an angle in radians.

Result

Real between -1 and +1 inclusive.

Notes

If the argument is outside the range -8388608 to 8388608 radians, it is impossible to determine how many PIs to subtract. The error message `Accuracy lost in sine/cosine/tangent` is displayed.

Examples

```
PRINT COS (RAD (45) )
```

```
adjacent = hypotenuse * COS (angle)
```

COUNT

Function giving the number of characters printed since the last newline.

Syntax

```
COUNT
```

Result

Positive integer, giving the number of characters output since the last newline was generated by BASIC.

Notes

COUNT is reset to zero every time a carriage return is printed (which may happen automatically if WIDTH is being used). It is incremented every time a character is output by PRINT, INPUT or REPORT, but not when output by VDU or any of the graphics commands. COUNT is also reset to zero by CLS and MODE.

Examples

```
REPEAT PRINT " " ;
```

```
UNTIL COUNT=20
```

```
chars = COUNT
```

DATA

Passive statement marking the position of data in the program.

Syntax

```
DATA [<expression>] [, <expression>], etc
```

Arguments

The expressions may be of any type and range, and are only evaluated when a READ statement requires them.

Notes

The way in which DATA is interpreted depends on the type of variable in the READ statement. A numeric READ evaluates the data as an expression, whereas a string READ treats the data as a literal string. Leading spaces in the data item are ignored, but trailing spaces (except for the last data item on the line) are counted. If it is necessary to have leading spaces, or a comma or quote in the data item, it must be put between quotation marks. For example:

```
DATA " HI", "A, B, ", """ABCD"
```

If an attempt is made to execute a DATA statement, BASIC treats it as a REM. In order to be recognised by BASIC, the DATA statement, like other passive statements, should be the first on a line.

Examples

```
DATA Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec
```

```
DATA 3.26, 4, 4.3, 0
```


DEF

Passive statement defining a function or procedure.

Syntax

- (1) DEF FN<proc part>
- (2) DEF PROC<proc part>

where <proc part> has the form <identifier> [<parameter list>]

Parameters (1) and (2)

The optional parameters, which must be enclosed between round brackets and separated by commas, may be of any type. For example: parm, parm%, parm\$, !parm, \$parm. In addition, whole arrays may be passed as parameters.

Purpose

The DEF statement marks the first line of a user-defined function or procedure, and also indicates which parameters are required and their types. The parameters are local to the function or procedure, and are used within it to stand for the values of the actual parameters used when it was called.

Notes

Function and procedure definitions should be placed at the end of the program, so that they cannot be executed except when called by the appropriate PROC statement of FN function. The DEF statement should be the first on the line. If not, it is (like REM) ignored.

Examples

```
DEF FNmean(a,b)
```

```
DEF PROCinit
```

```
DEF PROCthrow_dice (d%, tries, msg$)
```

```
DEF PROCarray_determinant (A())
```

DEG

Function returning the number of degrees of its radian argument.

Syntax

```
DEG<factor>
```

Argument

Any numeric value.

Result

A real equal to $180 * n / \text{PI}$ where 'n' is the argument's value.

Example

```
angle=DEG(ATN(a))
```

```
PRINT DEG(PI/4)
```


DELETE

Command to delete a section of the program.

Syntax

```
DELETE<integer>, <integer>
```

Arguments

Integer constants in the range zero to 65279. They give the first and last line to be deleted respectively. If the first line number is greater than the second, no lines are deleted. To delete just a single line the DELETE command is not necessary. Instead type the line number and press .

Examples

```
DELETE 5,22
```

```
DELETE 110,150
```

DIM

Statement declaring arrays or reserving storage.

Syntax

```
DIM <dim part>[,<dim part>] etc
```

where <dim part> is:

(1) <identifier>[% or \$] (<expression>[,<expression>]etc)

or

(2) <numeric variable><space><expression>

or as a function:

(3) DIM (<arrayname>())

(4) DIM (<arrayname>(),<numeric expression>)

Arguments (1)

The <identifier> can be any real, integer or string variable name. The expressions are integers which should be greater than or equal to zero. They declare the upper bound of the subscript; the lower bound is always zero.

This is the way to declare arrays in BASIC. They may be multi-dimensional: the bounds are limited only by the amount of memory in the computer. Numeric arrays are initialised to zeros and string arrays to null strings.

Arguments (2)

The <numeric variable> is any integer or real name. The <expression> gives the number of bytes of storage required minus one, and should be -1 or greater. It is limited only by the amount of free memory.

The use of this form of DIM is to reserve a given number of bytes of memory, in which to put for example, machine code. The address of the first byte reserved is placed in the <numeric variable>. The byte array is uninitialised.

Arguments (3)

The <array name> is the name of any previously DIMed array. The result of the function is the number of dimensions which that array has.

Arguments (4)

The <array name> is the name of any previously DIMed array. The <numeric expression> is a number between one and the number of dimensions of the array. The result of the function is the subscript of the highest element in that dimension, or one less than the number of elements since subscripts start at 0.

Notes

It is possible to have local arrays, whose contents are discarded when the procedure or function in which they are created returns. See LOCAL.

The address returned by the byte form of DIM is guaranteed to lie on a word boundary: it will be a multiple of four. This is useful when you DIM space for machine code.

Examples

```
DIM name$(num_names%)  
  
DIM sin(90)  
  
DIM matrix%(4,4)  
  
DIM A(64), B%(12,4), C$(2,8,3)  
  
DIM bytes% size*10+overhead  
  
PRINT DIM(name$( ))  
  
size%=DIM(name$( ),1)
```

DIV

Integer operator giving the quotient of its operands.

Syntax

```
<operand>DIV<operand>
```

Operands

Integer-range numerics. Reals are converted to integers before the divide operation is carried out. The right-hand side must not evaluate to zero.

Result

The (integer) quotient of the operands, always rounded towards zero. The remainder can be found using MOD.

Examples

```
PRINT (first-last) DIV 10
```

```
a%=space% DIV &100
```

DRAW

Statement to draw a line to specified coordinates.

Syntax

- (1) DRAW<expression>, <expression>
- (2) DRAWBY<expression>, <expression>

Arguments (1)

The <expression>s are integer numerics in the range -32768 to +32767. They are the coordinates to which a line is drawn in the current graphics foreground colour. The graphics cursor position is updated to these coordinates. DRAW is equivalent to PLOT 5.

Arguments (2)

The <expression>s are integer numerics in the range -32768 to +32767. They are the offsets from the current graphics cursor to which a line is drawn in the current graphics foreground colour. The graphics cursor position is updated to these coordinates. DRAWBY is equivalent to PLOT 1.

Examples

```
DRAW 640,512 : REM Draw a line to the middle of the screen
```

```
DRAW BY x%*16, y%-4
```

EDIT

Command to enter the BASIC screen editor.

Syntax

EDIT

Purpose

EDIT enters the BASIC screen editor to allow you to create a new program or amend the current one. Full details of the editor are given in the chapter: **THE BASIC SCREEN EDITOR**.

ELLIPSE

Statement to draw an ellipse.

Syntax

- (1) ELLIPSE <exp1>, <exp2>, <exp3>, <exp4>
- (2) ELLIPSE FILL <exp1>, <exp2>, <exp3>, <exp4>
- (3) ELLIPSE <exp1>, <exp2>, <exp3>, <exp4>, <exp5>
- (4) ELLIPSE FILL <exp1>, <exp2>, <exp3>, <exp4>, <exp5>

Arguments

<exp1> to <exp4> are integer numerics giving the 'x' and 'y' coordinates of the centre of the ellipse, the length of the semi-major axis and the length of the semi-minor axis respectively. <exp5> is an optional parameter giving the angle in radians between the 'x' axis and the semi-major axis. If this parameter is absent then the angle is zero and the ellipse is axes-aligned.

ELLIPSE draws the outline of an ellipse. ELLIPSE FILL plots a solid ellipse.

Notes

The ELLIPSE statement has some restrictions about the size of its operands: if both of the semi-axes are of length zero, then you are not allowed to specify a rotation value. If the semi-minor axis length is zero, then the rotation, if specified, must not be zero. The result of trying to draw any of these 'illegal' ellipses is a Division by zero error.

Examples

```
ELLIPSE 640,512,200,100
```

```
ELLIPSE FILL x%,y%,major%,minor%,ang
```

ELSE

Part of the ON GOTO/GOSUB/PROC ... ELSE or IF ... THEN ... ELSE or IF ... THEN ... ELSE ... ENDIF constructs.

Syntax

- (1) ELSE<statements>
- (2) ELSE<statements>
 <statements>
 <statements>
 ENDIF

Notes (1)

ELSE may occur anywhere in the program, but is only meaningful after an IF or ON ... GOSUB/GOTO/PROC statement. If the expression after the IF evaluates to FALSE (zero), or the expression after the ON is not in the correct range, then the statements following the ELSE are executed. Elsewhere ELSE is treated as a REM statement.

Notes (2)

When ELSE is used as part of a block structured IF construct, and the expression after the IF evaluates to FALSE, all statements after the ELSE down to an ENDIF statement are executed. The ELSE must be the first non-space object on the line.

Examples

```
If a=b THEN PRINT "hello" ELSE PRINT "goodbye"
```

```
IF ok ELSE PRINT "Error"
```

```
ON choice GOSUB 100,200,300,400 ELSE PRINT"Bad choice"
```

```
IF num>=0 THEN  
PRINT SQR(num)  
ELSE  
PRINT "Negative number"  
PRINT SQR(-num)  
ENDIF
```

END

Statement terminating the execution of a program or a function returning the top of memory used.

Syntax

as a statement:

(1) END

as a function:

(2) END

Purpose (1)

The END statement terminates the execution of a program.

Note

This statement is not always necessary in programs; execution stops when the line at the end of the program is executed. However, END (or STOP) must be included if execution is to end at a point other than at the last program line. This prevents control falling through into a procedure, function or subroutine. END is also useful in error handlers.

Purpose (2)

The END function returns the address of the top of memory used by a program and its variables.

Example

```
PRINT END
```

ENDCASE

Statement marking the end of a CASE ... OF ... WHEN ... OTHERWISE ... ENDCASE construct.

Syntax

```
ENDCASE
```

Notes

ENDCASE must be the first non-space object on the line. When the statements corresponding to a WHEN or OTHERWISE statement have been executed, control then jumps to the statement following the ENDCASE. If ENDCASE itself is executed, it signals the end of the CASE statement, no matches having been made. Control then continues as normal.

ENDIF

Terminates an IF ... THEN ... ELSE ... ENDIF construct.

Syntax

```
ENDIF
```

Notes

ENDIF marks the END of a block-structured IF statement. It must be the first non-space object on a line. When the statements corresponding to the THEN or ELSE statement have been executed, control jumps to the statement following the ENDIF. If ENDIF itself is executed, it signals the end of the IF statement, nothing having been executed as a result of it. Control then continues as normal.

ENDPROC

Statement marking the end of a user-defined procedure.

Syntax

```
ENDPROC
```

Purpose

When executed, an ENDPROC statement causes BASIC to terminate the execution of the current procedure and to restore local variables and actual parameters. Control is passed to the statement after the PROC which called the procedure. ENDPROC should only be used in a procedure. Otherwise, when it is encountered, a Not in a procedure error message is displayed.

Examples

```
ENDPROC
```

```
IF a<=0 THEN ENDPROC ELSE PROCrecurse(a-1)
```

ENDWHILE

Statement to terminate a WHILE ... ENDWHILE loop.

Syntax

ENDWHILE

Notes

When an ENDWHILE is executed, control loops back to the corresponding WHILE statement. The statements forming the WHILE ... ENDWHILE loop are executed until the condition following the matching WHILE evaluates to FALSE when control jumps to the statement following the ENDWHILE.

Example

```
MODE 15
INPUT X
WHILE X > 0
  GCOL X
  CIRCLE 640,512,X
  X -= 4
ENDWHILE
```

EOF

Function indicating whether the end of a file has been reached.

Syntax

EOF#<factor>

Argument

A channel number returned by an OPEN function.

Result

TRUE if the last character in the specified file has been read, FALSE otherwise.

Examples

```
REPEAT  
VDU BGET#file  
UNTIL EOF#file
```

```
IF EOF#invoices PRINT "No more invoices"
```

EOR

Operator giving the logical or bitwise exclusive-OR.

Syntax

```
<relational>EOR<relational>
```

Operands

Relational expressions, or bit values to be exclusive-ORed

Result

The logical bitwise exclusive-OR of the operands. Corresponding bits in the operands are ex-ORed to produce the result. Each bit in the result is zero if the corresponding bits in the operands are equal, and one otherwise.

Examples

```
PRINT height>10 EOR weight<20
```

```
bits = mask EOR value1
```

ERL

Function returning the last error line.

Syntax

ERL

Result

Integer between zero and 65279. This is the line number of the last error to occur. An error line of zero implies that the error happened in immediate mode or that there has not been an error.

Note

If an error occurs inside a LIBRARY procedure, ERL is the number of the last line of the main program. It does not indicate where in the library the error occurred.

Examples

```
REPORT  
IF ERL<>0 THEN PRINT " at line "; ERL  
  
IF ERL=3245 PRINT "Bad function, try again"
```

ERR

Function returning the last error number.

Syntax

ERR

Result

A four-byte signed integer. Errors produced by BASIC are in the range one to 127.

Notes

The error number zero is classed as a fatal error and cannot be trapped by the ON ERROR statement. An example of a fatal error is that produced when a BASIC STOP statement is executed.

Examples

```
IF ERR=18 THEN PRINT "Can't use zero; try again!!"
```

```
IF ERR=17 THEN PRINT "Sure?" : IF GET$="Y" or GET$ = "y" THEN STOP
```

ERROR

Generates an error, or is part of the ON ERROR statement.

Syntax

- (1) ON ERROR
- (2) ERROR<expression1>, <expression2>

Notes (1)

See ON ERROR for details of the error handling statements.

Arguments (2)

<expression1> evaluates to a four-byte signed integer corresponding to an error number.

<expression2> evaluates to a string associated with this error number. The error described is generated.

Examples

```
ERROR 6, "Type mismatch: number needed"
```

```
ERROR ERR, REPORT$
```

EVAL

Function causing its argument to be evaluated.

Syntax

```
EVAL<factor>
```

Argument

A string which EVAL evaluates as a BASIC expression.

Result

EVAL can return anything that could appear on the right-hand side of an assignment statement. It can also produce the same errors that occur during assignment. For example: Type mismatch: number needed, and No such function/procedure.

Examples

```
INPUT hex$
PRINT EVAL("&"+hex$)

f$="LEFT$( " : e$=EVAL(f$+"""ABCDE"" , 2) "
```

EXP

Function returning the exponential of its argument.

Syntax

```
EXP<factor>
```

Argument

Numeric from the largest negative real (about $-1E38$) to approximately $+88$.

Result

Positive real in the range zero to the largest positive real (about $1E38$). The result could be expressed as $e^{(\text{argument})}$, where 'e' is the constant 2.718281828.

Example

```
DEF FNcosh(x)=(EXP(x) + EXP(-x))/2
```

EXT#

Pseudo-variable returning or controlling the length (extent) of an open file.

Syntax

- (1) EXT#<factor>
- (2) EXT#<factor>=<expression>

Argument (1)

Channel number, as allocated by one of the OPEN functions.

Result

Integer giving the current length of the file from zero to, in theory 2147483648, although in practice the extent is limited by the file medium in use.

Argument (2)

Channel number as allocated by one of the OPEN functions.

<expression> is the desired extent of the file, whose upper limit depends on the filing system. The lower limit is zero. The main use of the statement is to shorten a file. For example: EXT#file=EXT#file-&1000. A file may be lengthened by using PTR#.

Note

The operators += and -= cannot be used.

Examples

```
IF EXT#file>90000 THEN PRINT "File full":CLOSE#file
```

```
EXT#op=EXT#op+&2000
```

FALSE

Function returning the logical value FALSE.

Syntax

FALSE

Result

The constant zero. The function is used mnemonically in logical or conditional expressions.

Examples

```
flag=FALSE
```

```
REPEAT  
CIRCLE RND (1279) , RND (1024) , RND (200)  
UNTIL FALSE
```

FILL

Flood-fill an area in the current foreground colour.

Syntax

```
FILL <expression> , <expression>  
FILL BY <expression> , <expression>
```

Arguments

The <expression>s are integer numerics in the range -32728 to +32767. They are the coordinates from which flooding is to commence. If this point is in a non-background colour, nothing happens. Otherwise, it fills in all directions until it reaches either a non-background colour, the edge of the screen, or the edge of the graphics window.

FN

Word introducing or calling a user-defined function.

Syntax

- (1) DEF FN<proc part>
- (2) FN<proc part>

Arguments (1)

For the format of <proc part>, see DEF above. It gives the names and types of the parameters of the function, if any. For example:

```
1000 DEF FNmin(a%,b%) IF a%<b% THEN =a% ELSE =b%
```

a% and b% are the formal parameters. They stand for the expressions passed to the function (the actual parameters) when FNmin is called. The result of a user-defined function is given by a statement starting with '='. As the example above shows, there may be more than one '=' in a function. The first one which is encountered during execution terminates the function.

Notes

User-defined functions may span several program lines, and contain all the normal BASIC statements. For example, FOR loops, IF statements, and so on. They may also declare local variables using the LOCAL keyword.

Arguments (2)

<proc part> is an identifier followed by a list of expressions corresponding to the formal parameters in the DEF statement for the function. The result depends on the assignment that terminated the function, and so can be of any type and range. An example function call is:

```
PRINT FNmin(2*bananas%, 3*apples%+1)
```

Examples

```
DEF FNfact(n%) IF n%<1 THEN =1 ELSE =n%*FNfact(n%-1)

DEF FNhex4(n%)=RIGHT$("000"+STR$(n%),4)

REPEAT PRINT FNhex4(GET): UNTIL FALSE
```

FOR

Part of the FOR ... NEXT statement.

Syntax

```
FOR <numeric variable>=<expression> TO <expression> [STEP<expression>]
```

Arguments

The <numeric variable> can be any numeric variable reference. The <expression>s can be any numeric expressions, though they must lie in the integer range if the <numeric variable> is an integer one. It is recommended that integer looping variables are used for the following reasons:

- the loops go faster
- rounding errors are avoided.

If the STEP part is omitted, the step is taken to be +1.

Notes

The statements between a FOR and its corresponding NEXT are executed at least once: the test for loop termination is performed at the NEXT rather than the FOR. Thus a loop started with: FOR I=1 TO 0 ... executes once, with 'I' set to '1' in the body of the loop. The value of the looping variable when the loop has finished should be treated as undefined, and should not be used before being reset by an assignment.

Examples

```
FOR addr%=200 TO 8000 STEP 4
```

```
FOR I=1 TO LEN(a$)
```

GCOL

Statement to set the graphics colours and actions.

Syntax

(1) GCOL <expression> [TINT <expression1>]

(2) GCOL <expression1>,<expression2> [TINT <expression3>]

Argument (1)

<expression> is an integer between zero and 255 which determines the graphics foreground and background colours to be used in subsequent graphics commands. If it is in the range zero to 127 it affects the foreground colour. If it is in the range 128 to 255 it affects the background colour. The colour is treated MOD the number of colours in the current mode.

Hence, in two-colour modes only zero and one are unique. After this, the cycle repeats. Similarly, in four-colour modes, only zero, one, two and three are unique, etc.

Arguments (2)

<expression1> is the plot action in the range 0 to 255. It determines the effect of future PLOT commands on the screen. Currently defined values are:

Plot action	Meaning
0	Store the colour <expression2> on the screen
1	OR the colour on the screen with <expression2>
2	AND the colour on the screen with <expression2>
3	EOR the colour on the screen with <expression2>
4	Invert the current colour, disregarding <expression2>
5	Do not affect the screen at all
6	AND the colour on the screen with the inverse of <expression2>
7	OR the colour on the screen with the inverse of <expression2>

<expression2> determines the colour that will combine with the screen for PLOT actions 0 to 3, 6 and 7. It is in the range zero to 127, and is treated MOD the number of colours in the present mode. Adding 128 to the expression causes the background colour to be changed instead of the foreground.

If 16, 32, 48, or 64 is added to the values of <expression1> above, the first, second, third, or fourth extended colour fill pattern respectively is used instead of <expression2>. Adding 80 causes all four ECF patterns to be used, placed side by side.

VDU 23, 2 to VDU 23, 5 are used to set the colour fill patterns. If the currently selected pattern is re-defined, it becomes active immediately.

Notes

See the keyword TINT for details of the optional TINT value.

In addition to the plot actions listed, those in the range 8 to 15 are available. The effect of these is the same as the corresponding values in the range 0 to 7, except that in the range 8 to 15 it is possible to plot 'transparently'. This facility is used in two situations. When a sprite is plotted in one of these modes, and it has a 'mask', then only pixels where the mask is a 1 bit are plotted; the rest of sprite is made transparent.

The other situation is where colour patterns are used. Where a pixel in the pattern is the same as the current background colour, then that pixel is not plotted.

For example, suppose the display is a four-colour one, and the current background colour is 129 (red).

Now, if pattern 1 was selected as the foreground colour (GCOL 16, 0), a solid rectangle would be red-yellow, as pattern 1 consists of alternating red and yellow pixels. However, if the foreground colour was set using GCOL 24, 0 (adding 8 to the plot action number), then a solid rectangle would appear yellow, with transparent 'holes' where the red pixels would have been plotted.

Examples

```
GCOL 2 : DRAW 100,100 : REM Draw a line in logical colour 2
```

```
GCOL 4,128 : CLG : REM Invert the graphics window
```

```
GCOL 1,2: REM OR the screen with colour 2
```

GET

Function returning a character code from the input stream.

Syntax

```
GET
```

Result

An integer between zero and 255. This is the ASCII code of the next character in the buffer of the currently selected input stream (keyboard or RS423). The function will not return until a character is available, and so it can be used to halt the program temporarily.

Note

The character entered is not reflected on the screen. To make it appear you must explicitly PRINT it.

Examples

```
PRINT "Press space to continue"  
REPEAT UNTIL GET=32
```

```
ON GET-127 PROCa, PROCb, PROCc ELSE PRINT "Illegal entry"
```

GET\$

Function returning a character from the input stream.

Syntax

```
GET$
```

Result

A one-character string whose value would be CHR\$(GET) if GET had been called instead. This is provided so you can use statements like IF GET\$="*"... rather than IF CHR\$(GET)="*"...

Examples

```
PRINT "Do you want another game ";response$ = GET$  
IF response$ = "Y" or response$ = "y" CHAIN "program"  
STOP
```

```
PRINT "Input a digit ";  
PRINT GET$
```

GET\$#

Function returning a string from a file.

Syntax

```
GET$#<factor>
```

Argument

A channel number returned by an OPENIN, OPENOUT, or OPENUP function.

Result

A string of characters read until a linefeed (CHR\$10), carriage return (CHR\$13), null character (CHR\$0) or the end of the file is encountered, or else the maximum of 255 characters is reached.

Note

PTR# is updated to point to the next character in the file. If the last character in the file has been read, EOF# for the channel will be TRUE.

Examples

```
string$ = GET$# channel
```

```
PRINT GET$# fileno
```

GOSUB

Statement to call a subroutine.

Syntax

```
(1) GOSUB <expression>
```

```
(2) ON <expression> GOSUB <expression1> [,<expression>] [ELSE<statement>]
```

Argument (1)

<expression> should evaluate to an integer between zero and 65279, in other words a line number. If the expression is not a simple <integer> (eg 1030) it should be between round brackets. The line given is jumped to, and control is returned to the statement after the GOSUB by the next RETURN statement.

Arguments (2)

<expression> should evaluate to an integer. If this integer is 'm' then the mth subroutine listed after the GOSUB is jumped to. If the integer is zero or negative or greater than the number of line numbers given, the statement following the ELSE, if it is present, is executed.

Notes

The RENUMBER command will only work correctly if all GOSUB, GOTO and RESTORE line numbers are <integer>s. Line numbers that are expressions cannot be renumbered, so the program will stop working correctly.

Procedures should be used in preference to subroutines since they are more flexible and produce a better structured program.

Examples

```
GOSUB 2000
```

```
GOSUB (2300+20*opt)
```

```
ON x% GOSUB 100,200,300 ELSE PRINT "number out of range"
```

GOTO

Statement to transfer control to another line.

Syntax

- (1) GOTO <expression>
 (2) ON <expression> GOTO <expression1> [, <expressionn>] [ELSE <statement>]

Argument (1)

<expression> should evaluate to an integer between zero and 65279: a line number. If <expression> is not a simple <integer>, it should be between round brackets. This line number is then jumped to and execution carries on from this new line.

Arguments (2)

<expression> should evaluate to an integer.
 <expression1> - <expressionn> should evaluate to integer line numbers between zero and 65279. If the first integer is 'm' then the mth line after the GOTO is jumped to. If the integer is zero or negative or greater than the number of line numbers given, the statement following the ELSE, if it is present, is executed.

Notes

The RENUMBER command only works correctly if all GOSUB, GOTO and RESTORE line numbers are <integer>s. Line numbers that are expressions cannot be renumbered, so the program stops working correctly.

Examples

```
GOTO 230
```

```
IF TIME<1000 THEN GOTO 1000
```

```
ON x GOTO 20,50,30,160
```

HELP

Command giving help information.

Syntax

HELP

Purpose

HELP displays a list of useful information about the status of BASIC.

HIMEM

Pseudo-variable holding address of the top of the BASIC stack.

Syntax

- (1) HIMEM
- (2) HIMEM=<expression>

Result (1)

An integer giving the address of the location above the end of user memory. The amount of user memory is given by $HIMEM - LOMEM$ and the amount of free memory by $HIMEM - END$.

Argument (2)

<expression> should be an integer between LOMEM and the top of usable memory. It restricts the amount of memory which the current program can use for workspace etc, hence giving an area where data, or machine code routines can be stored.

Notes

If HIMEM is set carelessly, running the program may produce the No room error.

When an attempt is made to set HIMEM, LOMEM, or PAGE to an illegal value, a warning message is displayed, but the program nevertheless continues to run. This means that such errors cannot be trapped using ON ERROR.

Examples

```
PRINT "Memory available = ";HIMEM - LOMEM
```

```
a%=HIMEM-1000 : HIMEM=a%
```

IF

Statement to execute statements conditionally.

Syntax

- (1) IF <expression> [THEN] [<statements>] [[ELSE] [<statements>]]
- (2) IF <expression> THEN
 - <statements>
 - ELSE
 - <statements>
 - ENDIF

Arguments (1)

<expression> is treated as a truth value. If it is non-zero, it is counted as TRUE and any <statements> in the THEN part are executed. If the expression evaluates to zero, then the ELSE part <statements> are executed.

<statements> is either a list of zero or more statements separated by colons, or a line number. In the latter case there is an implied GOTO after the THEN, which has to be present.

Notes

The THEN is optional before <statements> except before '*' commands. For example:

```
IF a THEN *CAT
```

The ELSE part matches any IF, so be wary of nesting IFs on a line. Constructs of the form:

```
IF a THEN... IF b THEN... ELSE...
```

should be avoided by using instead:

```
IF a AND b THEN ... ELSE ...
```

However, the form:

```
IF a THEN... ELSE IF b THEN...
```

can be used.

Arguments (2)

<expression> is treated as a truth value. If it is non-zero, it is counted as TRUE and any <statements> on the line after the THEN down to either an ELSE or an ENDIF are executed. If the expression evaluates to zero, any <statements> following the ELSE until the ENDIF are executed.

Examples

```
IF temp<=10PROC low_temp
```

```
IF a%>b% THEN SWAP a%, b% ELSE PRINT "No swap"
```

```

IF B^2 >= 4*A*C THEN
PROCroots (A,B,C)
ENDIF

IF r$ = "Y" OR r$ = "y" THEN
PRINT "YES"
ELSE
PRINT "NO"
STOP
ENDIF

```

INKEY

Function returning a character code from the input stream or keyboard.

Syntax

- (1) INKEY<positive integer>
- (2) INKEY<negative integer>
- (3) INKEY<-256>

Argument (1)

An integer in the range zero to 32767, which is a time limit in centi-seconds.

Result

The ASCII code of the next character in the current input buffer if one appears in the time limit set by the argument, or -1 when the timeout occurs.

Argument (2)

An integer in the range -255 to -1, which is the negative INKEY code of the key being interrogated (see APPENDIX E for details).

Result

TRUE if the key is being pressed at the time of the call, FALSE if it is not.

Argument (3)

-256

Result

A number indicating which version of the operating system is in use.

Examples

```
DEF PROCwait (secs%)  
dummy=INKEY (100*secs%)  
ENDPROC
```

```
IF INKEY (-99) THEN REPEAT UNTIL NOT INKEY (-99)
```

INKEY\$

Function returning a character from the input stream.

Syntax

```
INKEY$<factor>
```

Argument

As INKEY

Result

Where INKEY would return -1, INKEY\$ returns the null string. In all other situations, it returns CHR\$ (INKEY<argument>).

Example

```
A$ = INKEY$(500)
```

INPUT

Statement obtaining a value or values from the input stream.

Syntax

INPUT is followed by an optional prompt, which, if present, may be followed by a semi-colon or comma, which causes a '?' to be printed out after the prompt. This is followed by a list of variable names of any type, separated by commas. After the last variable, the whole sequence may be repeated, separated from the first by a comma. In addition the position of prompts may be controlled by the SPC, TAB (and '' print formatters (see PRINT).

Notes

Leading spaces of the input string itself are skipped, and commas are taken as marking the end of input for the current item.

Examples

```
INPUT a$ : REM Print a simple "?" as a prompt
```

```
INPUT "How many",num% : REM prompt is "How many?"
```

```
INPUT "Address &"hex$ : REM prompt is "Address &" (no "?" because no ,)
```

```
INPUT TAB(10)"Name ",n$, 'TAB(10)"Address ",a$
```

```
INPUT a,b,c,d,"More ",yn$
```

```
INPUT SPC(5)"Letter",char$
```

INPUT LINE

Syntax

This has the same syntax as INPUT

Result

If the input variable is a string, all the user's input is read into the variable, including leading and trailing spaces and commas. If the input variable is numeric, only a single value will be selected from the beginning of the input line.

Note

INPUT LINE is equivalent to LINE INPUT

Example

```
INPUT LINE ">" basic$
```

INPUT#

Statement obtaining a value or values from a file.

Syntax

```
INPUT#<factor>[,<variable>] etc
```

Arguments

<factor> is the channel number of the file from which the information is to be read, as obtained by OPENIN or OPENUP. The list of zero or more <variable>s may be of any type. The separators may be semi-colons.

Integer variables are written as &40 followed by the twos complement representation of the integer in four bytes, least significant byte first.

Real variables are written as &FF followed by four bytes of mantissa and one byte exponent. The mantissa is sent lowest significant bit (LSB) first. 31 bits represent the magnitude of the mantissa and 1 bit the sign. The exponent byte is in twos complement excess 128 form.

String variables are written as &00 followed by a 1 byte count and then the characters in the string in reverse order.

Examples

```
INPUT#data,name$,addr1$,addr2$,addr3$,age%
```

```
INPUT#data,$buffer,len
```

INSTALL

Command to load a function or procedure library into memory.

Syntax

```
INSTALL <expression>
```

Argument

<expression> is a string which should evaluate to a filename that is valid for the filing system in use.

Purpose

INSTALL loads the chosen function and procedure library into the top of memory and lowers the BASIC stack and value of HIMEM by an appropriate amount. The library remains in memory until you QUIT from BASIC. Any number of libraries may be installed provided that there is enough memory for them.

When searching for a procedure or function, BASIC looks in the following order: first, the current program is searched, in line-number order; next, any procedure libraries loaded using LIBRARY are searched – the most recently loaded file is

searched first; finally, any INSTALLED libraries are examined, again in the reverse order of loading.

The LVAR command lists libraries in the order in which they are examined.

Examples

```
INSTALL "Printout"
```

```
A$ = "Library1"  
INSTALL A$
```

INSTR(

Function to find the position of a substring in a string.

Syntax

```
INSTR(<expression1>,<expression2>[,<expression3>])
```

Arguments

<expression1> is any string which is to be searched for a substring.
<expression2> is the substring required. <expression3> is a numeric in the range zero to 255 and determines the position in the main string at which the search for the substring will start. This defaults to one.

Result

An integer in the range zero to 255. If zero is returned, the substring could not be found in the main string. A result of one means that the substring was found at the first character of the main string, and so on. The position of the first occurrence only is returned.

Notes

If the substring is longer than the main string, zero is always returned. If the substring is the null string, the result is always equal to <expression3>, or one if this is omitted.

Examples

```
REPEAT a$=GET$:UNTIL INSTR("YyNn",a$) <> 0

pos%=INSTR(com$,"*FX",10)
```

INT

Function giving the integer part of a number.

Syntax

```
INT<factor>
```

Argument

Any integer-range numeric.

Result

Nearest integer less than or equal to the argument.

Examples

```
DEF FNround(n)=INT(n+0.5)

size=len%*INT((top-bottom)/100)
```

LEFT\$(

Function returning, or statement altering the left part of a string.

Syntax

- (1) LEFT\$(<expression>)
- (2) LEFT\$(<expression1>,<expression2>)
- (3) LEFT\$(<string variable>) = <expression>
- (4) LEFT\$(<string variable>,<expression1>) = <expression2>

Arguments (1)

<expression> is a string of between zero and 255 characters.

Result

A string containing all characters except the right-most one is returned.

Arguments (2)

<expression1> is a string of between zero and 255 characters.

<expression2> is a numeric in the range zero to 255.

Result

A string taken from the leftmost <expression2> characters of <expression1>. If <expression2> is greater than LEN(<expression1>) then the whole string is returned.

Arguments (3)

<string variable> is the name of the string variable to be altered. The characters in <varname> are replaced, starting from the left-hand character (position 1), by the string <expression>.

Arguments (4)

<string variable> is the name of the string variable to be altered. The characters in <string variable> are replaced, starting from the left-hand character (position 1), by the string <expression2>. <expression1> is the

maximum number of characters which will be displaced. In other words, the number of characters being altered is the lesser of <expression1> and LEN <expression2>.

Examples

```
start$ = LEFT$(a$)
```

```
left_half$=LEFT$(input$,LEN(input$)DIV2)
```

```
LEFT$(A$) = "ABCD"
```

```
LEFT$(A$,n%) = B$
```

LEN

Function returning the length of a string.

Syntax

```
LEN<factor>
```

Argument

Any string of zero to 255 characters.

Result

The number of characters in the argument string, from zero to 255.

Examples

```
REPEAT INPUT a$: UNTIL LEN(a$)<=10
```

```
IF LEN(in$) > 12 THEN PRINT "Too long"
```


LET

Statement assigning a value to a variable.

Syntax

```
LET <variable>=<expression>
```

Arguments

The <variable> is any addressable object, such as 'a', a\$, a%, !a, a?10, \$a, a(), and so on. <expression> is any expression of the range and type allowed by the variable: for reals, any numeric; for integers, any integer-range numeric; for strings, any string of zero to 255 characters, and for bytes any integer in the range zero to 255 (though an integer-range number will be treated MOD 256).

Notes

The LET keyword is always optional in an assignment, and must not be used in the assignment to a pseudo-variable. For example, LET TIME=100 is illegal.

Examples

```
LET starttime=TIME
```

```
LET a$=LEFT$(addr$,10)
```

```
LET table?i=127*SIN(RAD(i))
```

```
LET a() = 1
```

```
LET A%() = B%() + C%()
```

LIBRARY

Command to load a function or procedure library into memory.

Syntax

```
LIBRARY <expression>
```

Argument

<expression> is a string which should evaluate to a filename that is valid for the filing system in use.

Purpose

LIBRARY DIMS an area in the BASIC heap and loads the chosen function and procedure library into this area. It remains there until the heap is cleared. Whilst it is in memory, the current program can call any of the procedures and functions it contains. See also INSTALL.

Examples

```
LIBRARY "Printout"
```

```
A$ = "Library1"  
LIBRARY A$
```

LINE

Draw a line between two points.

Syntax

```
LINE <expression>, <expression>, <expression>, <expression>
```

Arguments

The <expression>s are integer numerics in the range -32768 to +32767. They are two pairs of coordinates between which the line is drawn. The line is drawn in the current graphics foreground colour and the graphics cursor position is updated to the latter pair of coordinates. It is equivalent to a MOVE followed by a DRAW.

Examples

```
LINE 100,100,600,700
```

```
LINE x1,y1,x2,y2
```

```
LINE x1,y1,x1+xoffset,y1+yoffset
```

LINE INPUT

Syntax

This has the same syntax as INPUT

Result

If the input variable is a string, all the user's input is read into the variable, including leading and trailing spaces and commas. If the input variable is numeric, only a single value will be selected from the input line.

Note

LINE INPUT is equivalent to INPUT LINE

Example

```
LINE INPUT "Your message" mess$
```

LIST

Command to list the program.

Syntax

```
LIST [<line range>][IF <string>]
```

Arguments

<line range> gives the start and end lines to be listed. Both values are optional and should be separated by a comma. The first value defaults to zero and the last to 65279. The IF, when present, is followed by a string of ASCII characters. Only lines which contain this string are listed.

Notes

The string given after the IF is tokenised before it is checked against the program. Hence, LIST IF PRINT and LIST IF P . both list lines containing the PRINT keyword. However, LIST IF PR does not.

Because the string after IF is tokenised, only one version of the pseudo-variables (each of which has two tokens) may be found. This is the one acting as a function (as in PRINT TIME), rather than the statement version (as in TIME=<expression>).

Examples

LIST	list the whole program
LIST 1000,	list from line 1000 to the end
LIST ,50	list from the start to line 50
LIST 10,40	list from line 10 to 40 inclusive
LIST IFDEF	list all lines containing a DEF
LIST ,100 IFfred%=	list all lines up to line 100 containing fred%=

LISTO

Command to set the LIST indentation options.

Syntax

LISTO<expression>

Argument

<expression> should be in the range zero to 31 and is treated as a five-bit number. The meaning of the bits is as follows:

Bit	Meaning
0	A space is printed after the line number
1	Structures are indented
2	Lines are split at the ': statement delimiter
3	The line number is not listed. An error is displayed at line number references
4	List tokens in lower case

Notes

BASIC strips trailing spaces from program lines when they are entered. If the current LISTO option is non-zero, it also strips leading spaces. To enter blank lines (eg 1000), either execute LISTO 0 first, or include a colon on the blank line (eg 1000:).

Examples

LISTO 0	Default
LISTO 2	All loops and conditionals indented by two characters
LISTO %10011	Tokens in lower case, structures indented, line numbers given with a space after each

LN

Function returning the natural logarithm of its argument.

Syntax

LN<factor>

Argument

Any strictly positive value: a numeric greater than zero.

Result

Real in the range -89 to +88 which is the log to base 'e' (2.718281828) of the argument.

Examples

```
DEF FNlog2(n)=LN(n)/LN(2)
```

```
PRINT LN(10)
```

LOAD

Command to load a BASIC program at PAGE

Syntax

```
LOAD <expression>
```

Argument

<expression> is a string which should evaluate to a filename that is valid for the filing system in use.

Note

Any program which is currently in memory is overwritten and lost.

Examples

```
LOAD "PeteDisc:disasm"
```

where PeteDisc is the name of a floppy disc.

```
LOAD FNnextFile
```

LOCAL

Statement to declare a local variable in a procedure or function.

Syntax

```
LOCAL [<variable>] [,<variable>], etc
```

Arguments

<variable>s following the LOCAL may be of any type, such as 'a', 'a%', 'a\$', \$buffer, and so on. The statement causes the current value of the variables cited to be stored on BASIC's stack, ready for retrieval at the end of the procedure or function. This means the value inside the procedure may be altered without fear of corrupting a variable of the same name outside the procedure. At the end of the procedure, the old value of the variable is restored.

Notes

Local numerics are initialised to zero, and local strings are initialised to the null string.

Arrays can be declared as being local and then dimensioned using DIM as normal.

Examples

```
LOCAL dx, dy
```

```
LOCAL a$, len%, price
```

```
LOCAL a(), B() DIM a(2), B(4,5)
```


LOCAL ERROR

Makes the error control status local.

Syntax

```
LOCAL ERROR
```

Notes

LOCAL ERROR can be used anywhere inside a program. It remembers the current error handler so a subsequent use of ON ERROR does not overwrite it. This error handler can later be restored using RESTORE ERROR.

If LOCAL ERROR is used within a procedure or function it must be the last item to be made local.

Returning from a procedure or function call which contained a LOCAL ERROR automatically restores any stored error status.

See also ON ERROR LOCAL

Example

```
ON ERROR PROCerror
res = FNdivide(opp,adj)
END
DEFFNdivide(x,y)
LOCAL ERROR
ON ERROR LOCAL PRINT "attempt to divide by zero" :=0
=x/y : REM end of function restores previous error status
```

LOG

Function returning the logarithm to base ten of its argument.

Syntax

LOG <factor>

Argument

Any strictly positive value: a numeric greater than zero.

Result

Real in the range -38 to $+38$, which is the log to base ten of the argument.

Example

```
PRINT LOG(2.4323)
```

LOMEM

Pseudo-variable holding the address of BASIC variables.

Syntax

- (1) LOMEM
- (2) LOMEM=<expression>

Result (1)

The address of the start of the BASIC variables.

Arguments (2)

<expression> is the address at which BASIC variables start. The expression should be in the range TOP to HIMEM to avoid corruption of the program and/or No room errors.

Notes

LOMEM should not be changed after any assignments in a program. If it is, variables assigned before the change are lost. LOMEM is reset to TOP by CLEAR (and thus by RUN).

If you attempt to set LOMEM to an impossible value, a warning message is given and LOMEM is not altered.

Examples

```
LOMEM=TOP+400 : REM reserve 1K above TOP  
  
PRINT~LOMEM
```

LVAR

Command displaying the first line of all current libraries.

Syntax

```
LVAR
```

Purpose

LVAR lists all the values of BASIC variables, sizes of arrays, known procedures and functions. It also lists the first line of all libraries currently loaded. These are displayed in the same order as that in which the libraries are searched when a library procedure or function is called.

Note

In order for LVAR to be useful, you should ensure that the first line of each library includes the full name of the library and the name of a procedure which can be called to provide details of all the routines which the library contains.

MID\$(

Function returning, or statement assigning to a substring of a string.

Syntax

- (1) MID\$(*<expression1>*,*<expression2>*[,*<expression3>*])
 (2) MID\$(*<string variable>*,*<expression1>*[,*<expression2>*]) = *<expression3>*

Arguments (1)

<expression1> is a string of zero to 255 characters. *<expression2>* is the position within the string of the first character required. *<expression3>*, if present, gives the number of characters in the substring. The default value is 255 (or to the end of the source string).

Result

The substring of the source string, of a length given in the third argument, and starting from the position specified. The result string can never be of greater length than the source string.

Arguments (2)

<string variable> is the name of the string variable which is to be altered. *<expression3>* evaluates to a string which provides the characters to replace those in *<string variable>*. *<expression1>* is the position within the string of the first character to be replaced. *<expression2>*, if present, gives the maximum number of characters to be replaced. The replacement stops when the end of the string variable is reached, even if there are characters in *<expression3>* which are unused.

Examples

```
PRINT MID$("ABCDEFGH",2,3);" : REM should print "BCD"
```

```
PRINT MID$(any$,LEN(any$)+1,any%);" : REM gives a null string ""
```

```
right_half$=MID$(any$,LEN(any$) DIV 2)
```

```
MID$(A$,4,4) = B$
```

```
MID$(A$,2,5) = MID$(B$,3,6)
```

MOD

Operator giving the integer remainder of its operands.

Syntax

```
<operand>MOD<operand>
```

Arguments

The <operand>s are integer-range numerics.

Result

Remainder when the left-hand operand is divided by the right-hand one using integer division.

Examples

```
INPUT i%: i% = i% MOD max_num%
```

```
count%=count% MOD max% + 1
```

```
PRINT result% MOD 100
```

MODE

Function returning, or statement changing the display mode.

Syntax

- (1) MODE <expression>
- (2) MODE

Argument (1)

<expression> should be in the range 0 to 255.

There are 21 different modes, from zero to 20, although modes 18, 19 and 20 are only available when using multi-sync monitors. If <expression> is over 128, the mode used is <expression>-128. Sufficient memory, however, for two copies of the screen is reserved wherever possible. This allows you to have one copy on display whilst you are updating the other, which means that smooth animation can be obtained.

Details of all the modes available are given in APPENDIX E.

Note

Changing mode also does the following:

- clears the screen to the current text background colour
- sets COUNT to zero
- sets the text and graphics windows to their defaults of the whole screen
- homes the text cursor
- moves the graphics cursor to (0,0)
- resets the logical-physical colour map to the default for the new mode
- resets the colour-fill patterns to their defaults for the new mode
- sets the dot pattern for dotted lines to &AA.

Result (2)

An integer giving the current screen mode. If the screen mode was entered using a number greater than or equal to 128 (ie a shadow mode), this is not reflected in the value returned by the MODE function. For example, if you typed MODE 129, the MODE function would return 1.

Examples

```
MODE 0
```

```
MODE m%+128
```

```
PRINT MODE
```

MOUSE

Statement returning the mouse position and button status.

Syntax

- (1) MOUSE <variable1>, <variable2>, <variable3>
- (2) MOUSE ON [<numeric expr>]
- (3) MOUSE OFF
- (4) MOUSE COLOUR <numeric expr>, <numeric expr>, <numeric expr>, <numeric expr>
- (5) MOUSE TO <numeric expr>, <numeric expr>
- (6) MOUSE STEP <numeric expr>[, <numeric expr>]
- (7) MOUSE RECTANGLE <numeric expr>, <numeric expr>, <numeric expr>, <numeric expr>

Arguments (1)

The first two variables are assigned the 'x' and 'y' positions of the mouse as values in the range -32768 to 32767. The third integer is assigned a value giving the status of the mouse buttons as follows:

Value	Status
0	No buttons pressed
1	Right button only pressed
2	Middle button only pressed
3	Middle and right buttons pressed
4	Left button only pressed
5	Left and right buttons pressed
6	Left and middle buttons pressed
7	All three buttons pressed

Arguments (2)

This causes the mouse pointer to be displayed. The optional numeric expression is the pointer shape to be used.

Purpose (3)

This turns off the mouse pointer.

Arguments (4)

This sets the colour components of the mouse logical colour given in the first expression to the values given in the second, third and fourth.

Arguments (5)

This moves the mouse pointer to a position given by the first and second numeric arguments.

Arguments (6)

This controls the speed of movement of the mouse. If there is one argument, it is used as a multiplier for both the 'x' and 'y' movements. If there are two, the first is used for 'x' and the second for 'y'.

Arguments (7)

This sets a bounding rectangle outside which the mouse cannot move. The arguments are the left, bottom, right and top of the rectangle in graphics units. If the mouse pointer is outside the box when this command is given, it will be moved to the nearest point within it.

Example

```
MOUSE xpos%, ypos%, button%  
  
MOUSE ON  
  
MOUSE OFF  
  
MOUSE TO 100,100  
  
MOUSE RECTANGLE 640,512,1023,1279  
  
MOUSE STEP 3,2  
  
MOUSE COLOUR Col%, red%, green%, blue%
```

MOVE

Statement to set the position of the graphic cursor.

Syntax

- (1) MOVE <numeric expr>, <numeric expr>
- (2) MOVEBY <numeric expr>, <numeric expr>

Arguments (1)

The <expression>s are 'x' and 'y' coordinates in the range to 32768 to 32767, ie two-byte integers. The usual screen range for the x-coordinate is zero to 1279 and

for the y-coordinate it is zero to 1023, though this changes if a graphics origin has been defined. MOVE is equivalent to PLOT 4.

Arguments (2)

The <expression>s are 'x' and 'y' offsets from the graphics cursor in the range -32768 to 32767, ie two-byte integers. MOVEBY is equivalent to PLOT 0.

Examples

```
MOVE 0,0 : REM Goto the origin
```

```
MOVE BY 4*x%, 4*y%
```

NEW

Command to remove the current program, and to initialise the computer so that it is ready to receive a new program.

Syntax

```
NEW
```

Purpose

The NEW command does not destroy the program, but merely sets a few internal variables as if there were no program in the memory. The effect of NEW may be undone using the OLD command, providing no program lines have been typed in, or variables created, between the two commands. BASIC does an automatic NEW whenever it is entered.

NEXT

Part of the FOR .. TO .. NEXT structure.

Syntax

```
NEXT [<variable>][, [<variable>]], etc
```

Arguments

The <variable>s are of any numeric type, and if present should correspond to the variable used to open the loop.

Notes

The variables after the NEXT should always be specified as this enables BASIC to detect improperly nested loops. If the loop variable given after a NEXT does not correspond to the innermost open loop, BASIC closes the inner loops until a matching looping variable is found. The indentation produced using LISTO to format the listing will help to highlight this kind of error.

Examples

```
NEXT a%
```

```
NEXT      : REM close one loop
```

```
NEXT j%,i% : REM close two loops
```

```
NEXT , , , : REM close four loops
```

NOT

Function returning the logical or bitwise NOT of its argument.

Syntax

```
NOT<factor>
```

Argument

An integer-range numeric.

Result

An integer in which all the bits of the argument have been inverted: ones have changed to zeros and zeros have changed to ones. If the argument is a truth value, NOT can be used in a logical statement to invert the condition. In this case, the truth value should only be one of the values -1 (TRUE) and zero (FALSE).

Examples

```
IF NOT ok THEN PRINT "Error in input"
```

```
inv%=NOT mask%
```

```
REPEAT UNTIL NOT INKEY(-99)
```

OF

Part of the CASE ... OF ... WHEN ... OTHERWISE ... ENDCASE statement.

Syntax

```
CASE <expression> OF
```

Arguments

<expression> may yield any type of value: integer, floating point, or string.

Notes

The OF keyword must be the last item on the line.

Examples

```
CASE n% OF
```

```
CASE LEFT$(answer$) OF
```


OFF

Statement to turn off the cursor or part of the ON ERROR, TRACE, MOUSE, or SOUND statements.

Syntax

- (1) OFF
- (2) ON ERROR OFF
- (3) TRACE OFF
- (4) MOUSE OFF
- (5) SOUND OFF

Purpose (1)

OFF turns the cursor off so that it is no longer visible. When the cursor is disabled using the OFF keyword, it is only temporarily removed. It will reappear if you start to perform cursor editing, and remain visible when you press . To disable the cursor in a more permanent way, use

```
VDU 23,0,10,32|
```

and to re-enable it use

```
VDU 23,0,10,107|
```

Purpose (2)

ON ERROR OFF disables error trapping so that when an error occurs, the default error action of printing the error message (and line number) and terminating the program takes place. See ON ERROR.

Purpose (3)

TRACE OFF turns off the tracing of the current program. This is done automatically when an error occurs. See TRACE.

Purpose (4)

Turns off the mouse pointer. See MOUSE.

Purpose (5)

Turns off all sound output. Cancelled by SOUND ON. See SOUND.

OLD

Command to retrieve a program after `NEW` has been typed.

Syntax

`OLD`

Purpose

The `OLD` command retrieves a program lost by `NEW` or `BREAK` providing no new program lines have been entered, or variables defined. When you recover the previous program using `OLD`, you may notice that the first line number has changed. In particular, it is now its old value `MOD 256`. So if the first line used to be 1000, it will now be 232. You can remedy this slight problem using the `RENUMBER` command.

ON

Statement to turn the cursor on or part of the `ON... GOTO/GOSUB/PROC` and `ON ERROR, MOUSE` and `SOUND` statements.

Syntax

- (1) `ON`
- (2) `ON <expression0> GOTO <expression1> [, <expressionn>] [ELSE<statement>]`
- (3) `ON <expression0> GOSUB <expression1> [, <expressionn>] [ELSE<statement>]`
- (4) `ON <expression0><proc> [, <proc>] [ELSE<statement>]`
- (5) `ON ERROR [<statements>]`
- (6) `ON ERROR LOCAL [<statements>]`
- (7) `SOUND ON`
- (8) `MOUSE ON [<expression>]`

Purpose (1)

`ON` turns the cursor on so that it is visible. This is the default status. However, it can be altered using `OFF`.

Arguments (2) and (3)

<expression0> following the ON is an integer between '1' and 'n', where 'n' is the number of expressions following the GOTO/GOSUB. The expressions from <expression1> to <expressionn> are line numbers (see GOSUB for the rules for line numbers). The optional ELSE part follows the last line number, and is followed by a statement.

Purpose

The ON ... GOTO/GOSUB statement provides a multi-way branch facility. <expression0> is evaluated. If its value is 'm', then the mth line number following the GOSUB/GOTO is jumped to. If 'm' is less than one or greater than the number of line numbers given, the ELSE part is executed. If there is no ELSE part, an ON range error is generated. Note that only a single statement may come after ELSE. Any other following statements, separated by colons, will be executed unconditionally. For example:

```
10 ON a GOSUB 100,200,300 ELSE PRINT"error":PROCerrStuff
```

The call to PROCerrStuff will be made whether the value of 'a' is outside of the range of one to three or not. Only one statement is subject to the ELSE. The consequence of this is that the ELSE part of an ON statement usually contains a procedure call or GOTO statement.

The difference between the GOTO and GOSUB versions is that in the latter case control is returned to the statement following the ON when a RETURN is executed.

Arguments (4)

<expression0> following the ON is an integer between one and 'n', where 'n' is the number of <proc> parts. The <proc> parts are normal calls to procedures, with or without parameters. The optional ELSE part follows the last line number, and is followed by a statement.

Purpose

The ON ... PROC statement is very similar to ON... GOSUB, the difference being that a call is made to a procedure instead of to a subroutine. The note about ELSE made above applies here too.

Arguments (5) and (6)

The <statement>s following ERROR are zero or more legal BASIC statements separated by colons.

Purpose

When an ON ERROR or ON ERROR LOCAL is executed, BASIC remembers the position of the statements following it. When an error is subsequently encountered, BASIC jumps to and executes the statements it has remembered. After an ON ERROR, all loops, procedures, etc are closed. However, after an ON ERROR LOCAL the current status is remembered. Hence ON ERROR NEXT is not sensible, but ON ERROR LOCAL NEXT can be.

Purpose (7)

Enables SOUND output.

Purpose (8)

Turns on the mouse pointer. See MOUSE.

Examples

```
ON  
  
ON choice% GOSUB 1000,2000,3000,4000 ELSE PRINT "Bad choice"  
  
ON ASC(c$) - 127 GOTO 100,120,10,200  
  
ON choice%+1 PROCload(prog$), PROCsave(prog$) ELSE PROCerr
```

```

ON ERROR GOTO 10000

ON ERROR PROCerr

ON ERROR IF ERR=17 THEN RUN ELSE REPORT:PRINT "at line";ERL:END

ON ERROR OFF : REM Disable the error trapping

ON ERROR LOCAL Print"Operation failed":NEXT

```

OPENIN

Function opening a file for input only.

Syntax

```
OPENIN<factor>
```

Argument

A string which evaluates to a valid filename for the filing system in use.

Result

An integer acting as a channel number for the file. The exact value depends upon the filing system and how many files are already open. The value is zero if the file was not found. The file is opened for input only.

Examples

```
in_file%=OPENIN("Invoices")
```

```
data%=OPENIN(":0"+data$)
```

OPENOUT

Function for opening a new file for output.

Syntax

```
OPENOUT<factor>
```

Argument

A string which evaluates to a valid filename for the filing system in use.

Result

An integer acting as a channel number for the file. If the file does not already exist, then a new one is created. If a file of the same name does exist then that file is deleted before the new one is created. The file is opened for output only.

Examples

```
out_file%=OPENOUT("Customers")
```

```
data%=OPENOUT(":0."+data$)
```

OPENUP

Function for opening a file for input and output.

Syntax

```
OPENUP<factor>
```

Argument

A string which evaluates to a valid filename for the filing system in use.

Result

An integer acting as a channel number for the file. If the file does not already exist, a new one is created. If a file of the same name does exist, that file is deleted first and a new one created. The file is opened for input and output.

Example

```
random_file%=OPENUP("records")
```

OR

Operator giving the logical OR of its operands.

Syntax

```
<relational>OR<relational>
```

Arguments

<relational>s can be any integer-range numerics.

Result

An integer obtained by ORing together the corresponding bits in the operands. The operands may be interpreted as bit-patterns, in which case a bit in the result is set to one if either or both of the corresponding bits in the operands are one. Alternatively, they may be interpreted as logical values, in which case the result is TRUE if either or both of the operands are TRUE.

Examples

```
PRINT a% OR &AA55
```

```
IF a<1 OR a>10 THEN PRINT "Bad range"
```

ORIGIN

Statement to move the graphics origin.

Syntax

```
ORIGIN <expression>, <expression>
```

Arguments

The <expression>s are integer numerics in the range -32768 to +32767. They are the absolute coordinates of the new graphics origin: the position of the point (0,0). The graphics origin is used by all commands which create graphics, such as MOVE, LINE, PLOT, CIRCLE, and so on, and also by VDU 24 which creates a graphics window.

Example

```
ORIGIN 640,512 : Set new origin to the centre of the screen
```

OSCLI

Statement to pass a string to the operating system.

Syntax

```
OSCLI<expression>
```

Argument

<expression> should be a string of between zero and 255 characters. It is passed to the operating system OSCLI routine to be executed.

Notes

The difference between passing a string to the operating system via a '*' command and via OSCLI is that the former makes no attempt to process the text following it, whereas the latter evaluates the text as a BASIC string expression.

BASIC provides extra information when using '*' or OSCLI to allow these systems to be ported onto this computer. (Note that this does not happen for SYS 5, "fred").

Because the high user mode registers are not conveniently readable from other modes, the low registers pass values that must be moved to the correct registers. For further information see CALL.

R0	=	contains CLI string pointer
R1	=	contains &BA51Cxxx
R2	=	ARGP
R3	=	LINE
R4	=	current string pointer
R5	=	environment information pointer (as CALL)

The value in register 1 should be inspected by any routine in order to validate that the call is, indeed, from BASIC (it may be a good idea to check R2 to R5 for valid addresses); the value is also at R5!-4. The BASIC interpreter provides &BA51C005, the next &BA51C006 and so on. The value in LINE should not be relied on, except that it is sufficient for BASIC to produce the correct line number error definition. When BASIC is eventually returned to at the end of the SWI CLI call, its (user mode) registers must not have been disturbed.

Examples

```
OSCLI "CAT"
```

```
OSCLI "LOAD "+file#+" "+STR$buff% : REM get the file in buffer
```

```
OSCLI "FX138,0,"+STR$(ASC(c$)) : REM insert a character into input buffer
```

OTHERWISE

Part of the CASE ... OF ... WHEN ... OTHERWISE ... ENDCASE statement.

Syntax

```
OTHERWISE <statements>
```

Notes

The OTHERWISE statement is executed only when the previous WHEN statements have failed to match the value of the CASE expression. OTHERWISE matches any values. If it is present, therefore, all statements following it will be executed. Control then jumps to the statement following the ENDCASE.

Examples

```
OTHERWISE PRINT "Bad input"
```

```
OTHERWISE PROCdraw(x,y) : PROCwait
```

PAGE

Pseudo-variable holding the address of the program.

Syntax

- (1) PAGE
- (2) PAGE=<expression>

Result (1)

An address which is an unsigned number. PAGE is the location at which the current BASIC program starts.

Argument (2)

<expression> is an integer in the range OSHWM to HIMEM, and should be on a word boundary. By changing PAGE, several BASIC programs may reside in the machine at once.

Note

If you attempt to set PAGE to an invalid address, a warning message is given and PAGE is not altered.

Example

```
PAGE = HIMEM - &4000
```

PI

Function returning the value of π .

Syntax

```
PI
```

Result

The constant 3.141592653

Examples

```
DEF FNcircum(r)=2*PI*r
```


PLOT

Statement performing an operating system PLOT function.

Syntax

```
PLOT <expression1>,<expression2>,<expression3>
```

Arguments

<expression1> is the plot number in the range from zero to 255. For example, 85 is the plot number for an absolute triangle plot in the foreground colour. The second and third expressions are the 'x' and 'y' coordinates respectively, in the range -32768 to +32767. See APPENDIX F for a full list of PLOT codes.

Examples

```
PLOT 85,100,100 : REM Draw a triangle
```

```
PLOT 69,x,y : REM Plot a single point
```

POINT

Statement to plot a single point or move the on-screen pointer.

Syntax

- (1) POINT <expression>,<expression>
- (2) POINTBY <expression>,<expression>
- (3) POINTTO <expression>,<expression>

Arguments (1)

The <expression>s are integer numerics in the range -32768 to +32767. They are the coordinates at which the point will be placed. The point is plotted in the current graphics foreground colour, and the graphics cursor is updated to these coordinates.

Arguments (2)

The <expression>s are integer numerics in the range -32768 to +32767. They are offsets from the graphics cursor at which the point will be placed. The point is plotted in the current graphics foreground colour, and the graphics cursor is updated to these coordinates.

Arguments (3)

The <expression>s are integer numerics in the range -32768 to +32767. They are the coordinates at which the on-screen pointer will be placed if it is not linked to the mouse position. If the pointer is linked to the mouse this command is ignored.

Examples

```
POINT 320,600
```

```
POINT X%+4, Y%+4
```

```
POINT BY 100,0
```

```
POINT TO 640,512
```

POINT(

Finds the logical colour of a graphics pixel.

Syntax

```
POINT(<expression1>,<expression2>)
```

Arguments

<expression1> is the 'x' coordinate of the pixel and <expression2> is the 'y' coordinate. These are integers in the range -32768 to +32767.

Result

This is an integer in the range -1 to 'n', where 'n' is less than the number of logical colours in the current mode. For example, 'n' is 15 in a 16-colour mode. If the point specified lies outside the current graphics window, -1 is returned. Otherwise, it is the logical colour of the point. Note that the value returned is in the range 0 to 63.

The function `TINT (x, y)` will read the tint of the given co-ordinate, returning a value in the range 0 to 255.

Example

```
REPEAT Y%=Y%+4:UNTIL POINT(640,y%)<>0
```

POS

Function returning the x-coordinate of the text cursor.

Syntax

```
POS
```

Result

An integer between zero and 'n', where 'n' is the width of the current text window minus one. This is the position of the text cursor which is normally given relative to the left-hand edge of the text window. If the cursor direction has been altered using `VDU 23, 16, ...` then it is given relative to the negative 'X' edge of the screen which may be top, bottom, left or right.

Examples

```
old_x%=POS
```

```
IF POS<>0 THEN PRINT
```

PRINT

Print information on the output stream(s).

Syntax

The items following PRINT may be string expressions, numeric expressions, and print formatters. By default, numerics are printed in decimal, right justified in the print field given by '@%' (see below). Strings are printed left justified in the print field. The print formatters have the following effects when printing numbers:

- ; Do not right justify (print leading spaces before) numbers in the print field. Set numeric printing to decimal. Semi-colon stays in effect until a comma is encountered. Do not print a new line at the end if this is the last character of the PRINT statement.
- , (comma) Right justify numbers in the print field. Set numeric printing to decimal. This is the default print mode. Comma stays in effect until a semi-colon is encountered. If the cursor is not at the start of the print field, print spaces to reach the next one.
- ~ Print numbers as hexadecimal integers, using the current left/right-justify mode. Tilde stays in effect until a comma or semi-colon is encountered.
- ' (apostrophe) Print a new line. Retain current left/right justify and hexadecimal/decimal modes.
- TAB (If there is one argument, for example, TAB (n) , print (n-COUNT) spaces. If the cursor is initially past position 'n' (ie COUNT >n), print a new line first. If there are two arguments, for example, TAB (10, 20) , move directly to that tab position. Left/right-justify and hexadecimal/decimal modes are retained.

SPC (Print the given number of spaces. For example SPC (5) outputs five spaces. Right-justify and hexadecimal/decimal modes are retained.
space	Print the next item, retaining left/right-justify and hexadecimal/decimal modes.

When strings are printed the descriptions above apply, except that hexadecimal mode does not affect the string. Also no trailing spaces are printed after a string unless it is followed by a comma. This prints enough spaces to move to the start of the next print field.

The format in which numbers are printed, and the width of print fields are determined by the value of the special system integer variable, @%. Each byte in the variable has a special meaning. These are described below.

Byte 4

This determines whether the STR\$ function uses the print format determined by @%, when converting its argument to a string, or whether it will use a default general format. If the byte is zero (the default), STR\$ uses a general format. If it is non-zero, STR\$ uses the format determined by @%.

Byte 3

This selects the format to be used. The legal values are:

- 0- General format: Numbers have the form nnn.nnn, the maximum number of digits printed being given in byte 2. This is the default format.
- 1- Exponent format: Numbers have the form n.nnnEnn, the number of digits printed being given in byte 2.
- 2- Fixed format: Numbers have the form nnn.nnn, the number of digits after the decimal point being given in byte 2.

Byte 2

This determines the number of digits printed. In General format, this is the number of digits which may be printed before reverting to Exponent format (1 to 10); in Exponent format it gives the number of significant figures to be printed after the decimal point (1 to 10). In fixed format it gives the number of digits (exactly) that follow the decimal point.

Byte 1

This gives the print field width for tabulating using commas, and is in the range zero to 255.

Examples of @%

@%=&0102020A uses Fixed format with two decimal places in a tab field width of ten. In addition, STR\$ uses this format instead of its default (which is &0A0A). Numbers are printed out in the form 1.23, 923.10, etc.

@%=&00010408 uses Exponent format. Four significant digits are printed, in a field of eight characters. These numbers look like 1.234E0, 1.100E-3, etc.

@%=&0000090A uses General format with up to nine significant digits in a field width of ten characters. Note that General format reverts to Exponent format when the number is less than 0.1. This is the default setting of @%.

Notes

Setting byte two to 10, eg &0A0A, shows the inaccuracies which arise when trying to store certain numbers in binary. For example:

```
PRINT 7.7
```

prints 7.699999999 when @%=&0A0A.

The print formatters ' ', TAB (and SPC may also be used in INPUT statements.

Examples

```
PRINT"Hello there";
```

```
PRINTa, SIN (RAD (a)) , x, y' 'p, q
```

```
PRINT TAB (10, 3) "Profits" SPC (10); profits;
```

PRINT#

Print information to an open file.

Syntax

```
PRINT#<factor>[, <expression>], etc
```

Arguments

<factor> is the channel number of a file opened for output or update. The expressions, if present, are any BASIC integer, real or string expressions. They are evaluated and sent to the file specified with the corresponding type information.

Integers are written as &40 followed by the twos complement representation of the integer in four bytes, least significant byte first.

Reals are written as &FF followed by four bytes of mantissa and one byte exponent. 31 bits of the mantissa represent its magnitude and one bit its sign. It is sent with its least significant byte first. The exponent is in twos complement excess 128 form.

Strings are written as &00 followed by a one byte count of the length of the string, followed by the characters in the string in reverse order.

Example

```
PRINT#file, name$+"":", INT (100*price+.5) , qnty*
```

PROC

Word introducing or calling a user-defined procedure.

Syntax

- (1) DEF PROC<proc part>
- (2) PROC<proc part>

Arguments (1)

<proc part> has the form <identifier> [(<parameter list>)]. It gives the name of the procedure (the <identifier>) and the names and types of the optional parameters, which must be enclosed in brackets and separated by commas.

Arguments (2)

The second form is used when the procedure is actually invoked, and this time the parameter list comprises expressions of types corresponding to the parameters declared in the DEF PROC statement. The expressions are evaluated and assigned (locally) to the parameter variables. Control returns to the calling program when an ENDPROC is executed.

Examples

```
DEF PROCdelay(n)
TIME=0:REPEAT UNTIL TIME=n*100:ENDPROC

IF ?flag=0 THEN REPEAT PROCdelay(0.1): UNTIL ?flag
```


PTR#

Pseudo-variable accessing the pointer of a file.

Syntax

- (1) PTR#<factor>
- (2) PTR#<factor>=<expression>

Argument (1)

<factor> is a channel number, as returned from an OPENIN, OPENOUT, or OPENUP function.

Result

An integer giving the position of the next byte to be read or written in relation to the start. The minimum value is zero and the maximum value depends on the filing system in use.

Argument (2)

<factor> is as (1). The <expression> is an integer giving the desired position of the sequential pointer in the file.

Examples

```
PRINT PTR#file;"bytes processed"
```

```
PTR#chan%=rec_len%
```

QUIT

Command to leave BASIC.

Syntax

QUIT

Purpose

QUIT leaves the BASIC interpreter.

RAD

Function returning the radian value of its argument.

Syntax

RAD<factor>

Argument

A number representing an angle in degrees.

Result

A real giving the corresponding value in radians: <argument>*PI/180.

Examples

```
sin%?i%=SIN(RAD(i%))
```

```
PRINT RAD(theta)-PI/2
```

READ

Statement reading information from a DATA statement.

Syntax

```
READ [<variable>], [<variable>], etc
```

Arguments

The zero or more variables should correspond in type to the items in the DATA statement being read. In fact, a string READ item is able to read any type of DATA and interpret it as a string constant after stripping leading spaces. A numeric READ item tries to evaluate its DATA; so in the latter case, the DATA expression should yield a suitable number.

Examples

```
READ n%
```

```
READ a$, fred%, float
```

RECTANGLE

Statement to draw a rectangle or copy/move a rectangular area of the screen or set the mouse bounding box.

Syntax

- (1) RECTANGLE <exp1>, <exp2>, <exp3>, <exp4>
- (2) RECTANGLE FILL <exp1>, <exp2>, <exp3>, <exp4>
- (3) RECTANGLE <exp1>, <exp2>, <exp3>, <exp4> TO <exp5>, <exp6>
- (4) RECTANGLE FILL <exp1>, <exp2>, <exp3>, <exp4> TO <exp5>, <exp6>
- (5) MOUSE RECTANGLE <exp1>, <exp2>, <exp3>, <exp4>

Arguments (1) and (2)

<exp1> and <exp2> are expressions evaluating to integer numerics in the range -32768 to +32767. They are the coordinates of one of the corners of the rectangle. <exp3> and <exp4> are similar expressions giving the offsets of the opposite corner: the width and height of the rectangle. If <exp4> is not present, then it is assumed to be the same as <exp3>.

Purpose

RECTANGLE draws the outline of a rectangle which is aligned with the 'x' and 'y' axes. RECTANGLE FILL plots a solid axes-aligned rectangle. The rectangles are drawn in the current graphics foreground colour.

RECTANGLE leaves the graphics cursor at the starting position. However, with RECTANGLE FILL, the graphics cursor is updated to the position of the opposite corner specified.

Arguments (3) and (4)

<exp1> to <exp6> are all expressions evaluating to integer numerics in the range -32768 to +32767. <exp1> and <exp2> are the coordinates of a corner of the source rectangle being defined. <exp3> and <exp4> are as in (1) and (2). <exp5> and <exp6> give the position to which the lower left corner of the source rectangle is copied or moved.

Purpose

RECTANGLE ... TO copies the original rectangular area defined to the new position, hence making a second copy of a rectangular screen area.

RECTANGLE FILL ... TO moves the original rectangular area defined to the new position, replacing the old area with the current graphics background colour. The new position can overlap with the rectangular area.

Purpose (5)

To set a bounding box for the mouse pointer. See MOUSE.

Examples

```
RECTANGLE 500,500,-200,-100
```

```
RECTANGLE FILL bottomleft%(1),bottomleft%(2),width%,height%
```

```
RECTANGLE 400,400,60,60 TO 460,400
```

```
RECTANGLE FILL x,y,size,size TO xnew,ynew
```

REM

Statement indicating a remark.

Syntax

```
REM<string>
```

Argument

<string> can be absolutely anything; it is ignored by BASIC. The purpose of a REM is to provide comments to make the program clear to any reader.

Example

```
REM find the next prime
```

RENUMBER

Command to renumber the program lines.

Syntax

```
RENUMBER [<integer>][,<step>]
```

Arguments

See AUTO for a description.

Purpose

RENUMBER resequences the lines in the program so that the first line is <integer> and the line numbers increase in steps of <step>. It also changes line numbers within the program, such as after GOTOs, so that they match the new line numbers. If the line used in a GOTO cannot be found, the message

```
Failed with nnnn on line 1111
```

is given, where nnnn is the line number which was referenced but which does not appear in the program, and 1111 is the line on which the reference was made.

RENUMBER needs some workspace, and if there is not enough room to change the line numbers successfully, a RENUMBER space error is generated.

Examples

```
RENUMBER
```

```
RENUMBER 1000,20
```

REPEAT

Statement marking start of a REPEAT ... UNTIL loop.

Syntax

```
REPEAT
```

Purpose

The statements following REPEAT are repeatedly executed until the condition following the matching UNTIL evaluates to FALSE. The statements may occur over several program lines, or may all be on the same line separated by colons. The second approach is useful in immediate statements. The statements are executed at least once.

Examples

```
REPEAT UNTIL NOT INKEY-99 : REM wait for SPACE to be released
```

```
REPEAT  
a%+=1:c%=c% >> 1  
UNTIL c%=0
```

REPORT

Statement printing the message of the last error encountered.

Syntax

```
REPORT
```

Examples

```
REPORT:PRINT "at line",ERL;END
```

```
REPORT:PRINT "error!!":END
```

REPORT\$

Function returning the message of the last error encountered as a string.

Syntax

```
REPORT$
```

Examples

```
PRINT REPORT$
```

```
ERROR ERR, REPORT$
```

RESTORE

Statement setting the DATA pointer.

Syntax

```
RESTORE [<expression>]
```

Argument

<expression> is a line number. If it is absent, the DATA pointer is reset to the first DATA statement in the program, and the next item READ comes from there. If the line number is present, the DATA pointer is set to the first item of data on or after the line specified, so that subsequent READs access that particular data item (and those which follow).

Examples

```
RESTORE
```

```
RESTORE 1000
```


RESTORE ERROR

Statement to restore saved error status.

Syntax

```
RESTORE ERROR
```

Notes

RESTORE ERROR restores the error status previously saved using LOCAL ERROR. If an error status has not been saved then a fatal error arises.

The error status is restored automatically when returning from a procedure or function.

Examples

```
LOCAL ERROR
ON ERROR PRINT"Negative value"
INPUT x
PRINT "Square root of x = ";SQR(x)
RESTORE ERROR
```

RETURN

Statement returning control from a subroutine.

Syntax

- (1) RETURN
- (2) RETURN<parameter>

Purpose (1)

RETURN returns control to the statement following the most recent GOSUB. If there are no GOSUBs currently active, a Not in a subroutine error occurs.

Purpose (2)

RETURN indicates value-and-result parameter passing (as distinct from value passing, the default) when applied to a parameter in the definition.

Example

```
DEF PROCswapIfDisordered (RETURN A, RETURN B)
    IF A>B SWAP A,B
ENDPROC
```

RIGHT\$(

Function returning or statement altering the right-most character(s) of a string.

Syntax

- (1) RIGHT\$(*<expression1>*)
- (2) RIGHT\$(*<expression1>*,*<expression2>*)
- (3) RIGHT\$(*<string variable>*) = *<expression>*
- (4) RIGHT\$(*<string variable>*,*<expression1>*) = *<expression2>*

Argument (1)

<expression1> should be a string of zero to 255 characters.

Result

A string consisting of the right-hand character is returned.

Arguments (2)

<expression1> should be a string of zero to 255 characters. *<expression2>* should be a numeric, 'n', in the range zero to 255.

Result

A string consisting of the right-most 'n' characters from the source string (<expression1>). If 'n' is greater than the length of the source string, the whole source string is returned.

Arguments (3)

<string variable> is the name of the string variable to be altered. The right-hand characters in <string variable> are replaced by the string <expression>.

Arguments (4)

<string variable> is the name of the string variable to be altered. The right-hand characters in <string variable> are replaced by the string <expression2>.

<expression1> is the maximum number of characters which will be replaced: the number of characters altered is the lesser of <expression1> and LEN <expression2>.

Examples

```
PRINT RIGHT$(any$, 4)
```

```
year$=RIGHT$(date$, 2)
```

```
RIGHT$(birthday$) = "May"
```

```
RIGHT$(name$, 4) = "Mary"
```

RND

Function returning a random number.

Syntax

- (1) RND
- (2) RND(<expression>)

Result (1)

A four-byte signed random integer between -2147483648 and +2147483647

Result (2)

- <expression> < 0 This reseeds the random number generator, and the function returns its argument as a result. Reseeding the generator with a given seed value always produces the same sequence of random numbers.
- <expression> = 0 This uses the same seed as the last RND call and returns the same random number rounded between zero and .999999999.
- <expression> = 1 This returns a random real number between zero and .999999999.

<expression> > 1 This expression, 'n', should be an integer. The result is an integer between one and 'n' inclusive.

Note that there should be no space before the opening bracket.

Examples

```
dummy=RND(-TIME) : REM reseed the generator 'randomly'  
ON RND(4) PROCone, PROTwo, PROCthree, PROCfour  
x%=RND(1280) : y%=RND(1024)
```

RUN

Statement to execute the current program.

Syntax

RUN

Purpose

RUN executes the program in memory, if one is present, after clearing all variables and resetting LOMEM.

SAVE

Command to save a program as a file.

Syntax

- (1) SAVE<expression>
- (2) SAVE

Argument (1)

<expression> should evaluate to a string which is a valid filename under the filing system in use. The current BASIC program is stored (without variables, etc) on the medium under this name.

Notes (2)

SAVE can be used without an expression, in which case the name is taken from the first line of the program which should have the format:

```
REM > <filename>
```

For example:

```
10 REM > Game1
```

Examples

```
SAVE "Version1"
```

```
SAVE FNprogName
```

```
SAVE
```

SGN

Function returning the sign of its argument.

Syntax

```
SGN<factor>
```

Argument

Any numeric.

Result

-1 for negative arguments, 0 for zero-valued arguments, and +1 for positive arguments.

Examples

```
DEF FNsquare(th)=SGN(SIN(th))  
ON SGN(arg)+2 PROCone, PROTwo, PROThree
```

Function returning the sine of its argument.

Syntax

```
SIN<factor>
```

Argument

A numeric representing an angle in radians.

Result

A real in the range -1 to 1, being the sine of the argument.

Notes

If the argument is outside the range -8388608 to +8388607 radians, an Accuracy lost in Sine/Cosine/Tangent error is displayed.

Examples

```
PRINT SIN(RAD(135))  
opp=hyp*SIN(theta)
```

SOUND

Statement generating a sound or suppressing/allowing subsequent sound generation.

Syntax

- (1) SOUND ON
- (2) SOUND OFF
- (3) SOUND <expression1>,<expression2>,<expression3>,<expression4>
- (4) SOUND <expression1>,<expression2>,<expression3>,<expression4>,<expression5>

Purpose (1) and (2)

SOUND ON is the default setting. It allows sounds to be produced by subsequent use of the SOUND (3) statements. SOUND OFF suppresses sounds and means that subsequent SOUND (3) statements have no effect.

Arguments (3) and (4)

<expression1> is the channel number, <expression2> is the amplitude, <expression3> is the pitch, <expression4> is the duration, and <expression5> is the delay.

Channel number

A two-byte integer giving the channel number to be used. It has the range 1 to 8.

Amplitude

This is an integer in one of two different ranges. The range -15 to zero is a simple volume (amplitude), -15 being the loudest and zero being the quietest (no sound). The range 256 (&100) to 511 (&1FF) is a logarithmic volume range, a difference of 16 providing a doubling or halving of the volume.

Pitch

This is treated as an integer. In the range zero to 255, the note middle C has a pitch value of 53; a difference in 'P' of 48 corresponds to a difference in pitch of one octave. In other words, there are four pitch values per semi-tone. In the range 256 (&100) to 32767 (&7FFF), the note middle C has a pitch value of &4000, and a difference in 'P' of &1000 corresponds to a difference in pitch of one octave.

Duration

The last SOUND parameter is also treated as a one-byte integer. It gives the duration of the note in twentieths of a second. A value of 255 gives a note with an infinite duration: one that does not stop unless the sound queue is flushed in some way. When an envelope is in use, 'D' gives the time for which the note sounds before the release phase is entered.

Delay

This is the number of beat counts from the last beat counter reset before the sound is produced. See BEATS and TEMPO for more details.

Examples

```
SOUND 1, -15, 255, 10
```

```
SOUND &102, &140, &2400, 200
```

```
SOUND 3, 300, 300, 100, 200
```

SPC

Print modifier to generate spaces.

Syntax

```
SPC<factor>
```

Argument

A one-byte integer between zero and 255. It gives the number of spaces to be printed.

Examples

```
PRINT a$;SPC(10);b$
```

```
INPUT SPC(7)"How many",a$
```

SQR

Function returning the square-root of its argument.

Syntax

```
SQR<factor>
```

Argument

Any positive numeric.

Result

A real which is the argument's square-root.

Examples

```
DEF FNlen(x1,y1,x2,y2)=SQR((x2-x1)^2+(y2-y1)^2)
```

```
disc=SQR(b*b-4*a*c)
```

STEP

Part of the FOR ... TO ... STEP statement or part of MOUSESTEP.

Syntax

- (1) FOR... TO... [STEP<expression>]
- (2) MOUSE STEP <expression>[,<expression>]

Argument (1)

<expression> is any numeric (preferably an integer) giving the step by which the FOR variable is to be incremented when a NEXT is encountered. If the STEP part is omitted, it is assumed to be one.

Purpose (2)

To control the speed of mouse movements. See MOUSE.

Examples

```
FOR i%=32 TO 128 STEP 32
```

```
FOR addr%=HIMEM TO HIMEM-&2000 STEP - 4
```

STEREO

Statement setting the stereo position of a sound channel.

Syntax

```
STEREO <expression1>,<expression2>
```

Arguments

<expression1> is the channel number which should be between one and the number of active channels (the maximum being eight). <expression2> is a

value giving the stereo position. It can take any value between -127 (meaning that the sound is fully to the left) and $+127$ (meaning that the sound is fully to the right). The default value of each is zero, giving central (mono) production.

If the number of physical channels is eight, only the channel specified is programmed. Otherwise, the following occurs:

No of channels	Channels programmed
1	<pos> to eight
2	<pos> and every other channel up to eight
4	<pos> and <pos>+4 if <pos>+4 is less than or equal to eight

Examples

```
STEREO 4, -60
```

```
STEREO n%, stereo%
```

STOP

Statement producing the fatal error `STOP` to terminate the program.

Syntax

```
STOP
```

Purpose

The `STOP` statement gives the fatal (untrappable) error message `Stopped`. It differs from `END`, as the latter produces no message. It may be used as a debugging aid to halt the program at a given point so that the current values of the program's variables can be determined.

Example

```
IF NOT ok THEN PRINT"Bad data":STOP
```

STR\$

Function producing the string representation of its argument.

Syntax

```
STR$[~]<factor>
```

Argument

Any numeric for decimal conversion, any integer for hexadecimal conversion. Decimal conversion is used when the tilde ('~') is absent, hex conversion when it is present.

Result

Decimal or hex string representation of the argument, depending upon the absence or presence of the tilde.

Notes

The string returned by STR\$ is usually formatted in the same way as the argument would be printed with @% set to &A0A. However, if the most significant byte of @% is non-zero, STR\$ returns the result in exactly the same format as it would be printed, taking the current value of @% into account. See also PRINT.

Examples

```
DEF FNhex4(a%)=RIGHT$("000"+STR$~(a%),4)
```

```
DEF FNdigits(a%)=LEN(STR$(a%))
```

```
dp=INSTR(STR$(any_val),".")
```

STRING\$(

Function returning multiple copies of a string.

Syntax

```
STRING$(<expression1>,<expression2>)
```

Arguments

<expression1> is an integer, 'n', in the range zero to 255. *<expression2>* should be a string of length zero to (255DIV n).

Result

A string comprising 'n' concatenated copies of the source string, of a length between zero and 255.

Examples

```
MODE 1
PRINT STRING$(40,"_"); :REM underline across the screen

pattern$=STRING$(20,"<-->")
```

SUM

Function returning the arithmetic sum or string concatenation of an array.

Syntax

```
SUM(<array identifier>)
```

Argument

<array identifier> is the name of an array.

Result

If the result is an integer or floating point array, it is an integer or floating point value of the sum of all the elements in the array.

If the result is a string array, it is the string which contains each of the elements of the array concatenated.

Examples

```
A() = 1
PRINT "There are ";SUM(A())" elements."

IF DIM(A()) = 1 AND DIM(B()) = 1 THEN
  IF DIM(A(),1) = DIM(B(),1) THEN
    DIM C(DIM(A(),1))
    C() = A()*B()
    PRINT "Scalar product is ";SUM(C())
  ENDIF
ENDIF
```

SWAP

Statement exchanging the value of two variables or arrays.

Syntax

```
SWAP <identifier1>,<identifier2>
```

Arguments

The arguments are variables or array names. Values must be of compatible type: string or numeric. Arrays must be of identical type elements (integer, floating point, string), but can be of differing sizes.

Purpose

The `SWAP` statement exchanges the contents of the two variables or arrays. In the case where arrays are swapped, the number of subscripts and their upper limits are also swapped. For example, if you have

```
DIM A(10), B(20, 20)
SWAP A(), B()
```

then after the `SWAP`, it would be as if the arrays had been `DIMed`:

```
DIM A(20, 20), B(10)
```

Examples

```
SWAP A%, B%
```

```
SWAP forename$, surname$
```

```
SWAP arr(1, 2), arr(2, 1)
```

```
SWAP array1(), array2()
```

```
SWAP a, B%
```

```
SWAP A$, $A%
```

```
SWAP matrix(), vector()
```

SYS

A statement for calling operating system routines.

Syntax

```
SYS <expl> [, <expn>] [TO <variable1>[, <variable2>]] [;<flags>]
```


Arguments

<exp1> defines which operating system routine is to be called. It may evaluate to an integer numeric giving the routine number, or to a string which is the name of a routine. The optional list of expressions following this, up to a maximum of eight, is passed to the routine via the registers. If the expression evaluates to a numeric, it is converted to an integer and placed directly in a register. If the expression evaluates to a string, the string is placed on BASIC's stack, beginning at a word boundary and terminating with a null character. A pointer to it is put in the register. Any expressions not given default to zero.

The optional TO is followed by a variable list. Each variable is assigned any value returned by the routine in the registers R0 to R7 respectively. If the variable to assign to is numeric, the integer in the register is converted to an appropriate format and stored in it. If the variable to assign to is a string, the register is treated as a pointer to a string terminated by zero, 10 or 13 and this string is assigned to the variable. The strings given on input can be overwritten, but should not be extended.

<flags> is an optional variable, to which the processor flag bits are returned. The value stored in the <flags> value is a binary number of the form %NZCV, where the letters stand for the result flags of the ARM status register.

Purpose

SYS provides access to the routines supplied by the operating system for entering and outputting characters, error handling, sprite manipulation, and so on. Details of these operating system routines is beyond the scope of this book.

Examples

```
SYS 0,ASC"A" : REM Write the character A to the screen
```

```
SYS 5,"CAT" : REM Equivalent to OSCLI"CAT"
```

```
SYS 4 TO char% : REM Read the ASCII value of a character input
```

TAB(

Print modifier to position text cursor.

Syntax

- (1) TAB (<expression>)
- (2) TAB (<expression1>, <expression2>)

Argument (1)

A numeric in the range zero to 255. It expresses the desired x-coordinate of the cursor. This position is obtained by printing spaces. A new line is generated first if the current position is at or to the right of the required one. COUNT is updated correctly.

Arguments (2)

<expression1> is the desired 'x'-coordinate; <expression2> is the desired 'y'-coordinate. The position is reached using the VDU 31 command. Both coordinates must lie within the current text window, otherwise, no cursor movement will take place. COUNT is no longer correct.

Examples

```
PRINT TAB(10) "Product";TAB(20) "Price"
```

```
INPUT TAB(0,10) "How many eggs",eggs*
```

TAN

Function giving the tangent of its argument.

Syntax

```
TAN<factor>
```

Argument

A real number interpreted as an angle in radians.

Result

A real giving the tangent of the angle, in the range $-1E38$ to $+1E38$.

Notes

If the argument is outside the range -8388608 to $+8388607$ radians, an *Accuracy lost in Sine/Cosine/Tangent* error message is displayed.

Example

```
opp=adj*TAN(RAD(theta))
```

TEMPO

Function returning or statement altering the beat counter rate.

Syntax

- (1) TEMPO <expression>
- (2) TEMPO

Argument (1)

<expression> is a hexadecimal fractional number, in which the three least-significant digits are the fractional part. A value of $\&1000$ corresponds to a tempo of one tempo beat per centi-second; doubling the value causes the tempo to double (two tempo beats per centi-second), halving the value halves the tempo (to half a beat per centi-second).

The tempo affects the rate at which the beat counter increases.

Result (2)

A number giving the current tempo.

Examples

```
TEMPO &2000
```

```
PRINT TEMPO
```

THEN

Part of the IF ... THEN ... ELSE and IF ... THEN ... ELSE ... ENDIF statements.

Syntax

```
(1) IF <expression> [THEN] [<statements>] [ELSE [<statements>]]
```

```
(2) IF <expression> THEN
```

```
    [<statements>]
```

```
    [ELSE]
```

```
    [<statements>]
```

```
ENDIF
```

Notes (1)

THEN is optional except in the case where a pseudo-variable is being assigned. It must also be included in statements of the form

IF <expression> THEN <expression>, where the second expression evaluates to an integer between zero and 64279; that is, a line number followed by a GOTO is implied.

Notes (2)

THEN is necessary in the block structured IF construct and must be the last object on the line.

Examples

```
IF a>3 THEN PRINT "Too large"      : REM THEN optional

IF mem THEN HIMEM = HIMEM - &2000

IF A$="Y" THEN 1200 ELSE GOTO 1400

MODE 1
IF colour$ = "red" THEN
COLOUR 1
CLS
ELSE
COLOUR 0 : CLS
ENDIF
```

TIME

Pseudo-variable reading or altering the value of the centi-second clock.

Syntax

- (1) TIME
- (2) TIME=<expression>

Result (1)

An integer giving the number of centi-seconds that have elapsed since the last time the clock was set to zero.

Arguments (2)

<expression> is an integer value used to set the clock. TIME is initially set to the lowest four bytes of the five byte clock value maintained by the operating system. Assigning to the TIME pseudo-variable alters the system centi-second timer: the one which is read and written by OS_Words &01 and &02 respectively. There is, however, an additional system clock which is monotonic: it

always increases in value with time, and cannot be reset by software. TIME does not affect this timer.

Example

```
DEF PROCdelay(n) TIME=0:REPEAT UNTIL TIME>=n*100
```

TIMES

Pseudo-variable accessing the real-time clock.

Syntax

- (1) TIMES
- (2) TIMES=<expression>

Result (1)

TIMES returns a 24-character string of the format:

```
Fri, 24 May 1984.17:40:59
```

The date and time part are separated by a full stop '.'

(2)

The expression should be a string specifying the date, the time, or both. Punctuation and spacing are crucial and should be as shown in the examples below.

Examples

```
PRINT TIMES$
```

```
TIMES$="Tue, 1 Jan 1972"
```

```
TIMES$="21:12:45"
```

```
TIMES$="Tue, 1 Jan 1972.21:12:06"
```

TINT

Part of the COLOUR or GCOL statements for use in 256-colour modes, or a statement on its own, or a function.

Syntax

- (1) COLOUR <num-expr> [TINT <num-expr>]
- (2) GCOL <num-expr>[, <num-expr>] [TINT <num-expr>]
- (3) TINT<num-expr>
- (4) TINT<x,y>

Arguments

The numeric expression following TINT is used to select the lowest two significant bits of the colour in 256-colour modes. Values between zero and 255 are permitted but only the lowest two bits of the number are significant.

Examples

```
COLOUR 1+J% TINT N%
```

```
GCOL 128+63 TINT 255 : REM Set the graphics background colour to solid white
```

The TINT keyword may also be used as a function. In the 256-colour modes it returns the tint value with which a pixel was written onto the screen. Examples are

```
GCOL 3 TINT TINT(x,y)
t=TINT(0,0)
```

TO

Part of the FOR ... TO ... STEP statement, part of MOUSE TO or part of POINT TO.

Syntax

See FOR, MOUSE or POINT

Examples

```
FOR i%=1 TO 100
```

```
MOUSE TO x%,y%
```

```
POINT TO x%,y%
```

TOP

Function returning the address of the end of the program.

Syntax

```
TOP
```

Result

TOP gives the address of the first byte after the BASIC program. The length of the program is equal to TOP-PAGE. LOMEM is usually set to TOP, so this is where the variables start.

Example

```
PRINT TOP
```


TRACE

Statement to initiate line tracing.

Syntax

- (1) TRACE<expression>
- (2) TRACE ON
- (3) TRACE PROC
- (4) TRACE STEP<expression>
- (5) TRACE STEP ON
- (6) TRACE STEP PROC
- (7) TRACE OFF

Arguments (1) and (2)

<expression> is a line number. All line numbers below this line number are printed out when they are encountered during the execution of the program.

TRACE ON is the same as TRACE 65279 ie all line numbers are printed as they are met.

Purpose (3)

TRACE PROC prints out the name of each procedure called.

Purposes (4), (5) and (6)

These are the same as TRACE<expression>, TRACE ON and TRACE PROC respectively, except that when each line number or procedure name has been printed, execution of the program stops and does not continue until a key is pressed.

Purpose (7)

TRACE OFF disables tracing.

Examples

```
IF debug THEN TRACE 9000
```

```
TRACE STEP PROC
```

```
IF debug THEN TRACE OFF
```

TRUE

Function returning the constant -1.

Syntax

```
TRUE
```

Result

TRUE always returns -1, which is the number yielded by the relational operators when the condition is true. For example, $1+1 < 3$ gives TRUE as its result.

Example

```
debug=TRUE  
IF debug PRINT"debug in operation"
```

UNTIL

Statement to terminate a REPEAT loop.

Syntax

```
UNTIL <expression>
```

Argument

<expression> can be any numeric expression which can be evaluated to give a truth value. If it is zero (FALSE), control passes back to the statement immediately after the corresponding REPEAT. If the expression is non-zero, control continues to the statement after the UNTIL.

Examples

```
DEF PROCirritate
REPEAT VDU 7:UNTIL FALSE:ENDPROC

REPEAT PROCmove:UNTIL gameOver
```

USR

Function returning the value of the first register after executing a machine code routine.

Syntax

```
USR<factor>
```

Argument

The address of the machine code to be called. Calls to the 6502-based BBC Microcomputer operating systems are handled by USR for compatibility.

USR is similar to CALL except that it returns the integer value of the first register as a result and cannot be passed any parameters (other than the address).

Result

An integer.

Example

```
DEF FNmachinecode :=USR(start_of_code)
```

VAL

Function returning the numeric value of a decimal string.

Syntax

```
VAL<factor>
```

Argument

A string of zero to 255 characters.

Result

The number that would have been read if the string had been typed in response to a numeric INPUT statement. The string is interpreted up to the first character that is not a legal numeric one (0 to 9, 'E' and '.').

Example

```
date=VAL(date$)
```

VDU

Statement sending bytes to the VDU drivers.

Syntax

```
VDU [<expression>] [, or ; or | or <expression>] etc [; or |]
```

Arguments

The zero or more <expression>s may be followed by a comma, a semi-colon, a vertical bar, or nothing in the case of the <expression> at the end of the line.

Expressions followed by a semi-colon are sent as two bytes (low byte first) to the operating system VDU drivers.

Expressions followed by a comma (or nothing in the case of the last expression) are sent to the VDU drivers as one byte, taken from the least significant byte of the expression.

The vertical bar means , 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, and so sends the <expression> before it as a byte followed by nine zero bytes. Since the maximum number of parameters required by any of the VDU statements is nine, the vertical bar ensures that sufficient parameters have been sent for any particular call. Any surplus ones are irrelevant, since VDU 0 does nothing.

Notes

For the meanings of the VDU codes, see the chapter: VDU CODES.

Examples

```
VDU 24,400;300;1000;740; : REM set up a graphics window
```

```
VDU 7,7 : Emit 2 beeps
```

```
VDU 23,9,200|23,10,200| : Slow down the flash rate of both alternating colours
```

VOICES

Statement specifying the number of sound channels to be used.

Syntax

```
VOICES <expression>
```

Arguments

<expression> is the number of channels to be used. The maximum number allowed is eight. Any number between one and eight can be specified, but the number which the computer is to handle must be a power of two and so the computer rounds up the number you give to either one, two, four or eight.

Note

The sound system uses up a lot of the computer's processing power, and so it is good practice to minimise the number of active channels. Otherwise, the computer will take longer to perform other tasks such as drawing to the screen.

Examples

```
VOICE 4
```

```
VOICE n%2
```

VPOS

Function returning the 'Y' coordinate of the text cursor.

Syntax

```
VPOS
```

Result

The vertical position of the text cursor relative to the top of the text window.

Examples

```
DEF FNmyTab(x%)
PRINT TAB(x%,VPOS);: ""

IF VPOS>10 THEN PRINT TAB(0,10);
```

WAIT

Statement to wait for end of the current display frame. Waiting until the end of the frame maximises the amount of time available in which to draw objects while the electron beam is 'blanked'.

Syntax

```
WAIT
```

Purpose

To enable a program to synchronise animation effects with the scanning of the display hardware.

WHEN

Part of the CASE ... OF ... WHEN ... OTHERWISE ... ENDCASE statement.

Syntax

```
WHEN <expression1> <,expressionn> : <statements>
```

Arguments

WHEN is followed by a list of expressions separated by commas. These expressions should evaluate to the same type as that of the expression following the corresponding CASE statement. If the value of the expression following the CASE statement matches that of any of the list following the WHEN, <statements> are executed and control is then passed to the statement following the ENDCASE.

Notes

WHEN must be the first non-space object on a line. A CASE statement can contain any number of WHEN statements, but only the statements of the first one which contains a matching value will be executed. To match any value, an OTHERWISE should be used.

Examples

```
WHEN 1 : PROCload
```

```
WHEN 2,4,6,8 : PRINT "Even" : remainder= 0
```

```
WHEN "Y","y","YES","Yes","yes" : PROCgame
```


WHILE

Statement marking the start of a WHILE ... ENDWHILE loop.

Syntax

```
WHILE <expression>
```

Arguments

<expression> can be any numeric or string condition which can be evaluated to give a truth value. If it is zero (FALSE), control passes forward to the statement immediately after the corresponding ENDWHILE. If it is non-zero, control continues until the ENDWHILE statement is reached, then loops back to the WHILE statement, and <expression> is re-evaluated.

Notes

The statements making up the WHILE ... ENDWHILE loop are never executed if the initial value of <expression> is FALSE (zero).

Examples

```
WHILE TIME < 1000  
PROCdraw  
ENDWHILE
```

```
WHILE flag : PROCmainloop : ENDWHILE
```

WIDTH

Statement setting the line width in BASIC.

Syntax

```
WIDTH<expression>
```

Arguments

<expression> should be a positive integer. Expressions in the range 1 to 2147483627 cause BASIC to print a new line and reset COUNT to zero every time COUNT exceeds that number. If the expression is zero-valued, BASIC stops generating auto-newlines, which is the default. The WIDTH keyword may also be used as a function. It returns the current value of the print width, or 0 if none has been set.

Examples

```
WIDTH 0: REM 'infinite width'
```

```
WIDTH 40: REM newline every 40 characters horizontally
```


VDU COMMANDS

The VDU (Visual Display Unit) driver is a part of the operating system which provides a set of routines used to display all text and graphical output. Any bytes sent to the VDU driver are treated either as characters to be displayed or as VDU commands: instructions which tell the driver to perform a specific function. Their interpretation depends on their ASCII values as follows:

ASCII value	Interpretation
0-31	VDU commands
32-126	Characters to be displayed
127	Delete
128-159	Characters to be displayed / Teletext control codes
160-255	Characters to be displayed

The statement `VDU X` is equivalent to `PRINT CHR$(X)`; except that VDU ignores the value of `WIDTH` and does not affect `COUNT`.

In addition, the VDU commands can be given from the keyboard by holding down **Ctrl** and one further key as shown in the table below. For example, to give the command `VDU 0`, you would press **Ctrl** and `@`. Some VDU commands require extra data to be sent. The number of bytes extra is also given in the table.

VDU Code	CTRL	Extra bytes	Meaning
0	@(2)	0	Do nothing
1	A	1	Send next character to printer only
2	B	0	Enable printer
3	C	0	Disable printer
4	D	0	Write text at text cursor
5	E	0	Write text at graphics cursor
6	F	0	Enable VDU driver
7	G	0	Generate bell sound
8	H	0	Move cursor back one character
9	I	0	Move cursor on one space
10	J	0	Move cursor down one line
11	K	0	Move cursor up one line
12	L	0	Clear text window
13	M	0	Move cursor to start of current line
14	N	0	Turn on page mode
15	O	0	Turn off page mode
16	P	0	Clear graphics window
17	Q	1	Define text colour
18	R	2	Define graphics colour
19	S	5	Define logical colour
20	T	0	Restore default logical colours
21	U	0	Disable VDU drivers
22	V	1	Select screen mode
23	W	9	Multi-purpose command
24	X	8	Define graphics window
25	Y	5	PLOT
26	Z	0	Restore default windows
27	{	0	Does nothing
28	\	4	Define text window
29	}	4	Define graphics origin
30	^(6)	0	Home text cursor
31	_(-)	2	Move text cursor

The VDU commands are described below.

VDU 0

VDU 0 does nothing.

VDU 1

VDU 1 sends the next character to the printer only, if the printer has been enabled (with VDU 2 for example).

VDU 2

VDU 2 causes all subsequent printable characters to be sent to the printer as well as to the screen.

VDU 3

VDU 3 reverses the effects of VDU 2 so that all subsequent printable characters are sent to the screen only.

VDU 4

VDU 4 causes all subsequent printable characters to be printed at the current text cursor position using the current text foreground colour.

VDU 5

VDU 5 links the text and graphics cursors and causes all subsequent printable characters to be printed at the current graphics cursor position using the current graphics foreground colour and action.

VDU 6

VDU 6 restores the functions of the VDU driver after it has been disabled (using VDU 21). Hence, this command causes all subsequent printable characters to be sent to the screen.

VDU 7

VDU 7 generates the bell sound.

VDU 8

VDU 8 causes either the text cursor (by default or after a VDU 4 command) or the graphics cursor (after a VDU 5 command) to be moved back one character position. It does not cause the last character to be deleted.

VDU 9

VDU 9 causes either the text cursor (by default or after a VDU 4 command) or the graphics cursor (after a VDU 5 command) to be moved on one character position.

VDU 10

VDU 10 causes either the text cursor (by default or after a VDU 4 command) or the graphics cursor (after a VDU 5 command) to be moved on one line.

VDU 11

VDU 11 causes either the text cursor (by default or after a VDU 4 command) or the graphics cursor (after a VDU 5 command) to be moved back one line.

VDU 12

VDU 12 clears either the current text window (by default or after a VDU 4 command) or the current graphics window (after a VDU 5 command) to the current text or graphics background colour respectively. In addition the text or graphics cursor is moved to its home position (see VDU 30).

VDU 13

VDU 13 causes the text cursor (by default or after a VDU 4 command) or the graphics cursor (after a VDU 5 command) to be moved to the start of the current line.

VDU 14

VDU 14 enters paged mode, and so makes the screen display wait for **Shift** to be pressed before displaying the next page.

VDU 15

VDU 15 cancels the effect of VDU 14 so that scrolling is unrestricted.

VDU 16

VDU 16 clears the current graphics window to the current graphics background colour using the graphics and action. It does not affect the position of the graphics cursor.

VDU 17,n

VDU 17 sets either the text foreground ('n' < 128) or background ('n' >= 128) colours to the value 'n'. It is equivalent to COLOUR n.

VDU 18,k,c

VDU 18 is used to define either the graphics foreground or background colour and the way in which it is to be applied to the screen. The BASIC equivalent is GCOL k, c.

VDU 19,l,p,r,g,b

VDU 19 is used to define the physical colours associated with the logical colour l.

If $0 \leq p \leq 15$, r, g and b are ignored, and one of the standard colour settings is used. This is equivalent to COLOUR l, p.

If $p = 16$, the palette is set up to contain the levels of red, green and blue dictated by r, g and b. This is equivalent to COLOUR l, r, g, b.

If $p = 24$, the border is given colour components according to r, g and b.

If $p = 25$, the mouse logical colour 1 is given colour components according to r, g and b . This is equivalent to `MOUSE COLOUR 1, r, g, b`.

VDU 20

VDU 20 restores the default palette for the current mode and so cancels the effect of all VDU 19 commands or their BASIC keyword counterparts. It also sets the default text and graphics foreground and background colours.

VDU 21

VDU 21 stops all further text and graphics output to the screen until a VDU 6 command is received.

VDU 22,n

VDU 22 is used to change mode. It is equivalent to `MODE n`.

See APPENDIX E for full details of the modes available.

VDU 23,p1,p2,p3,p4,p5,p6,p7,p8,p9

VDU 23 is a multi-purpose command taking nine parameters, of which the first identifies a particular function. Each of the available functions is described below. Eight additional parameters are required in each case.

VDU 23,0,n,m,0,0,0,0,0

If 'n' = 8, this sets the interlace as follows:

Value	Effect
$m = 0$	sets the screen interlace state to the opposite of the current *TV setting
$m = 1$	sets the screen interlace state to the current *TV setting
$m = \&80$	turns the screen interlace off
$m = \&81$	turns the screen interlace on

If 'n' = 10 or 11, this controls the height of the cursor on the screen and its appearance.

If 'n' = 10, then 'm' defines the start line for the cursor and its appearance. Thus:

Bits	Effect															
0-4	define the start line															
5-6	define its appearance:															
	<table border="0"> <thead> <tr> <th>Bit 6</th> <th>Bit 5</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>steady</td> </tr> <tr> <td>0</td> <td>1</td> <td>off</td> </tr> <tr> <td>1</td> <td>0</td> <td>fast flash</td> </tr> <tr> <td>1</td> <td>1</td> <td>slow flash</td> </tr> </tbody> </table>	Bit 6	Bit 5	Meaning	0	0	steady	0	1	off	1	0	fast flash	1	1	slow flash
Bit 6	Bit 5	Meaning														
0	0	steady														
0	1	off														
1	0	fast flash														
1	1	slow flash														

If 'n' = 11, then 'm' defines the end line for the cursor.

VDU 23,1,n,0,0,0,0,0,0

This controls the appearance of the cursor on the screen depending on the value of 'n'. Thus:

Value	Effect
n = 0	stops the cursor appearing
n = 1	makes the cursor reappear
n = 2	makes the cursor steady
n = 3	makes the cursor flash

VDU 23,2 to 5,n1,n2,n3,n4,n5,n6,n7,n8

These define the four colour patterns. Each of the parameters n1 to n8 defines one row of the pattern, n1 being the top row and n8 the bottom row. See the chapter: GRAPHICS for more details.

VDU 23,6,n1,n2,n3,n4,n5,n6,n7,n8

This sets the dot-dash line style used by dotted line PLOT commands. Each of the parameters n1 to n8 defines eight elements of the line style, n1 controlling the start and n8 the end. The bits in each are read from the most significant to the least significant, zero representing a space and one representing a dot. See the chapter: GRAPHICS for more details.

VDU 23,7,m,d,z,0,0,0,0

This scrolls the current text screen. The values of 'm', 'd' and 'z' determine the area to be scrolled, the direction of scrolling and the amount of scrolling respectively. Thus:

Value	Effect
m = 0	scroll the current text window
m = 1	scroll the entire screen
d = 0	scroll right
d = 1	scroll left
d = 2	scroll down
d = 3	scroll up
d = 4	scroll in the positive X direction
d = 5	scroll in the negative X direction
d = 6	scroll in the positive Y direction
d = 7	scroll in the negative Y direction
z = 0	scroll by 1 character cell
z = 1	scroll by 1 character cell vertically or 1 byte horizontally

VDU 23,8,t1,t2,x1,y1,x2,y2,0,0

This clears a block of the current text window to the text background colour. The parameters t1 and t2 indicate the base positions relating to the start and end of the block to be cleared respectively. The positions to which the values of 't' refer are shown below:

Value	Position
t = 0	top left of window
t = 1	top of cursor column
t = 2	off top right of window
t = 4	left end of cursor line
t = 5	cursor position
t = 6	off right of cursor line
t = 8	bottom left of window
t = 9	bottom of cursor column
t = 10	off bottom right of window

The parameters x_1 , y_1 and x_2 , y_2 are the 'x' and 'y' displacements from the positions specified by t_1 and t_2 respectively. They determine the start and end of the block.

VDU 23,9 to 10,n,0,0,0,0,0,0

These set the flash time for the first and second flashing colours respectively. The time is set to 'n' frame periods (approximately 1/50th of a second).

VDU 23,11,0,0,0,0,0,0,0

This sets the four-colour patterns to their default values. See the chapter: **GRAPHICS** for more details.

VDU 23,12 to 15,n1,n2,n3,n4,n5,n6,n7,n8

These set up the simple colour patterns. A block of two-by-four pixels is defined using the eight parameters. Each pair of parameters corresponds to the colours of the pixels on a given row, n_1 and n_2 being the top row and n_7 and n_8 the bottom row. See the chapter: **GRAPHICS** for more details.

VDU 23,16,n,0,0,0,0,0,0

This alters the direction of printing.

Normally when a character has been printed, the cursor moves to the right by one place, and then to the start of the row below when a character is entered in the right-hand column. This movement, however, can be altered so that, for example, the cursor moves down one row after each character, and moves to the top of the next column to the right when the bottom of the screen has been reached. This effect can be produced by typing

```
VDU 23,16,8,0,0,0,0,0,0,0
```

The effect on cursor movement depends on the value 'n' as shown below:

Value	Effect
0	Positive X direction is right, positive Y direction is down
2	Positive X direction is left, positive Y direction is down
4	Positive X direction is right, positive Y direction is up
6	Positive X direction is left, positive Y direction is up
8	Positive X direction is down, positive Y direction is right
10	Positive X direction is down, positive Y direction is left
12	Positive X direction is up, positive Y direction is right
14	Positive X direction is up, positive Y direction is left

Altering the direction of cursor movement also affects the way in which the screen scrolls; so in the example above, when a character has been entered at the bottom right-hand corner, the screen scrolls to the left by one column rather than scrolling up by one row as it usually does.

The following is the complete list of VDU commands for moving the cursor:

Command	Movement
VDU 8	moves the cursor one place in the negative X direction
VDU 9	moves the cursor one place in the positive X direction
VDU 10	moves the cursor one place in the negative Y direction
VDU 11	moves the cursor one place in the positive Y direction

VDU 23,17,n,m,0,0,0,0,0

If 'n' = 0 to 3, this command sets the tint to the value 'm' for the text foreground, text background, graphics foreground and graphics background colours respectively. It is equivalent to TINT n, m. See the chapter: SCREEN MODES for more details.

If 'n' = 4, this command chooses which set of default ECF patterns is used. 'm' = 0 gives the Master 128-compatible set; 'm' = 1 gives the native set. See the chapter: GRAPHICS for more details.

If 'n' = 5, this command swaps the text foreground and background colours.

VDU 23,18 to 24,n1,n2,n3,n4,n5,n6,n7,n8

These are reserved for future expansion.

VDU 23,25,n1,n2,n3,n4,n5,n6,n7,n8

VDU 23, 25 is used for aliased fonts.

VDU 23, 25, 1-4 sets the transfer function which allows the initial 16 grey levels used for anti-aliased fonts to be translated into eight, four or two levels. n1 specifies the number of bits per pixel used for anti-aliasing. The threshold values, n2 to n7 are used to decide which output bits are produced from the original values (0 to 15). There are either one, three or seven of them depending on the number of bits per pixel.

VDU 23, 25, 128-143 sets up the relevant palette registers for anti-aliasing in two, four, or 16 colour modes. n2 specifies the foreground logical colour to be used for anti-aliasing, and n1 to n128 specify the background logical colour. The start and end physical colours are specified by n3,n4,n5 and n6,n7,n8 respectively; in each case the three colours give the amount of red, green, and blue to be used in the colour. The background logical colour is set to the specified start physical colour, and subsequent logical colours up to n2 are set to intermediate values up to the end physical colour.

VDU 23, 25, &FF defines the setting of one of the 16 logical colours, n2, in the pseudo palette maintained for anti-aliasing in 256-colour modes. n3,n4,n5 give the amounts of red, green and blue for the start physical colour. n5,n6,n7 give them for the end physical colour.

VDU 23,26,h,s,p1,p2,s1,s2,0,0

VDU 23, 26 selects the font whose name is given with point size 's' and gives it the handle 'h'.

p1 and p2 give the number of pixels per inch horizontally and vertically in the screen mode in which the font will be painted. If zero, the values are chosen depending on the current screen mode.

s1 and s2 are the 'x' scale and 'y' scale respectively. They allow non-integer point sizes to be used, and also allow the horizontal and vertical point sizes of a font to be different. The size of the font is calculated in 16ths of a point, by multiplying the point size by the 'x' and 'y' scales respectively.

VDU 23,27,m,n,0,0,0,0,0

If 'm' = 0, this command selects the sprite whose name is 'n'. It is equivalent to *SCHOOSE n.

If 'm' = 1, this command defines sprite 'n' to contain the contents of the previously marked rectangle. It is equivalent to *SGET n.

See the chapter: SPRITES for more details.

VDU 23,28 to 31,n1,n2,n3,n4,n5,n6,n7,n8

These are reserved for use by applications programs.

VDU 23,32 to 255,n1,n2,n3,n4,n5,n6,n7,n8

These redefine the printable ASCII characters. The bit pattern of each of the parameters n1 to n8 corresponds to a row in the eight-by-eight grid of the character. See the chapter **OUTPUTTING TEXT** for more details.

VDU 24,x1;y1;x2;y2;

VDU 24 defines a graphics window. The four parameters define the left, bottom, right and top boundaries respectively, relative to the current graphics origin.

The parameters may be sent as shown, with semicolons after them. This indicates that the values are each two bytes long. Alternatively, they can be sent as eight one-byte values separated as usual by commas. The first of each pair contains the low byte for the boundary; the second contains the high byte. For example,

```
VDU 24,160;300;360;800;
```

is equivalent to

```
VDU 24,160,0,44,1,104,1,32,3.
```

See the chapter: **WINDOWS** for more details.

VDU 25,k,x;y;

VDU 25 is a multi-purpose graphics plotting command. It is equivalent to **PLOT** k, x, y. See the chapter: **GRAPHICS** for more details.

VDU 26

VDU 26 returns the text and graphics windows to their default states: full screen size. In addition, it resets the graphics origin to (0,0), moves the graphics cursor to (0,0), and moves the text cursor to its home position.

VDU 27

VDU 27 has no effect.

VDU 28,lx,by,rx,ty

VDU 28 defines a text window. The parameters specify the boundary of the window; the left-most column, the bottom row, the right-most column and the top row respectively. See the chapter: **WINDOWS** for more details.

VDU 29,x;y;

VDU 29 moves the graphics origin. 'x' and 'y' specify the 'x' and 'y' coordinates of the new position. Normally the origin is at the bottom left of the screen at (0,0): whenever a position is given as an absolute value, for example `MOVE 20, 80`, the coordinates are taken as being relative to the graphics origin. This command, therefore, affects all movements of the graphics cursor and all subsequent graphics window commands. The position on the screen of any existing graphics window is not affected. This command is equivalent to `ORIGIN x, y`.

VDU 30

VDU 30 moves the text cursor to its home position.

VDU 31,x,y

VDU 31 moves the text cursor to a specified position on the screen. It is equivalent to `PRINT TAB (x, y) ;`

OPERATING SYSTEM COMMANDS

Commands can be sent to the operating system either by preceding them with an asterisk or by using the BASIC OSCLI statement. For example, you can use either

```
*KEY 1 MODE 1|M
```

or its equivalent

```
OSCLI "KEY 1 MODE 1|M"
```

This chapter provides a brief description of all the operating commands which may be called.

*AUDIO

This command takes a parameter ON or OFF. It is used to enable or disable the sound system. Once disabled, the sound system will cease to process any more SOUND or *SOUND commands. The most recent command, however, will be retained and processed when the sound system is re-enabled. The *AUDIO command is equivalent to SOUND ON and SOUND OFF in BASIC.

*CHANNELVOICE

The CHANNELVOICE command is used to assign a voice name or number to one of the eight sound channels. It has the form:

```
*ChannelVoice <channel> <index>|<name>
```

The <channel> is the channel number to which you want the voice to be attached. <index> is the number of the voice required, as printed by the *VOICES*1 command. Alternatively you can specify a <name>. This is the textual name of the voice, which is also printed by *VOICES. Note that if you use a name instead of a number for the voice, it must have exactly the same spelling, which includes the case of the letters, as the name printed by *VOICES.

CONFIGURE

*CONFIGURE defines the CMOS RAM configurations as shown below:

Command	Effect																				
BAUD n	sets the RS423 transmit and receive rate as follows:																				
	<table><thead><tr><th>'n'</th><th>baud rate</th></tr></thead><tbody><tr><td>0</td><td>9600</td></tr><tr><td>1</td><td>75</td></tr><tr><td>2</td><td>150</td></tr><tr><td>3</td><td>300</td></tr><tr><td>4</td><td>1200</td></tr><tr><td>5</td><td>2400</td></tr><tr><td>6</td><td>4800</td></tr><tr><td>7</td><td>9600</td></tr><tr><td>8</td><td>19200</td></tr></tbody></table>	'n'	baud rate	0	9600	1	75	2	150	3	300	4	1200	5	2400	6	4800	7	9600	8	19200
'n'	baud rate																				
0	9600																				
1	75																				
2	150																				
3	300																				
4	1200																				
5	2400																				
6	4800																				
7	9600																				
8	19200																				
BOOT	reverses the actions of Break and Shift Break .																				
CAPS	sets the keyboard caps lock mode to on.																				

DATA n specifies the data format used by the RS423 interface as follows:

'n'	word length	parity bits	stop
0	7	even	2
1	7	odd	2
2	7	even	1
3	7	odd	1
4	8	none	2
5	8	none	1
6	8	even	1
7	8	odd	1

DELAY n sets the keyboard auto-repeat delay to 'n'/100 seconds.

DIR causes ADFS to initialise with a directory selected.

DRIVE n causes the machine to initialise with drive n selected. On a machine with both Winchester and floppy drives, n has the following meaning:

n = 4-7	Select Winchester drive 4-7
n = 0-3	Select floppy drive 0-3

DUMPFORMAT n selects the format to be used by *DUMP, *LIST, *LIST commands and the VDU: output device. 'n' is a four-bit number. The bottom two bits define how control characters are displayed, as follows:

value	meaning
0	GSREAD read format is used (eg A for ASCII 1)
1	period (.) is used
2	<n> is used, where 'n' is in decimal
3	<&n> is used, where 'n' is in hex

If bit 2 is cleared, top-bit set characters are treated as control codes, otherwise they are treated as printable characters.

If bit 3 is set, characters are ANDed with &7F before being processed, otherwise they are left as they are.

FILE n defines the default filing system to file 'n'

FLOPPIES n causes the machine to initialise as though 'n' floppy drives were attached

FONTSIZE n reserves n * 4K pages for the font cache. If n = 0 then no space is reserved for fonts. The maximum space allowed is 255 * 4K.

FS [nnn.]sss selects the number of the network fileserver. sss is the station number. nnn is the net number, which is optional.

HARDDISCS n causes the machine to initialise believing that 'n' Winchester discs are attached.

IGNORE n	sets the printer ignore character to CHR\$(n) . If 'n' is missing, all characters are sent.								
LOUD	sets full volume for the bell sound.								
MODE n	defines the default screen mode to be mode 'n'.								
MONITORTYPE n	The parameter is the type of monitor fitted to the machine, as follows:								
	<table border="0" style="margin-left: 40px;"> <thead> <tr> <th style="text-align: left;">parameter</th> <th style="text-align: left;">type</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>15-17 KHz (TV standard)</td> </tr> <tr> <td>1</td> <td>25-32 KHz Multisync</td> </tr> <tr> <td>2</td> <td>64 KHz High resolution monitors (400 series only)</td> </tr> </tbody> </table>	parameter	type	0	15-17 KHz (TV standard)	1	25-32 KHz Multisync	2	64 KHz High resolution monitors (400 series only)
parameter	type								
0	15-17 KHz (TV standard)								
1	25-32 KHz Multisync								
2	64 KHz High resolution monitors (400 series only)								
NOBOOT	assigns the normal functions to Break and Shift Break .								
NOCAPS	resets the keyboard caps lock (ie caps lock is off).								
NODIR	causes ADFS to initialise with no directory selected.								
NOSCROLL	enables the scroll protect option.								

PRINT n defines the printer driver type as follows:

'n' printer driver type

0 sink (ie no output printed)

1 printer port (ie parallel, Centronics type)

2 RS423 port (ie serial)

3 user printer driver

4 network printer

PS [nnn.]sss selects the number of the network printer server. *sss* is the station number. *nnn* is the net number, which is optional.

QUIET sets half volume for the bell sound.

RAMFSIZE n sets the number of 32K pages for the RAM filing system (when that module is present).

RMASIZE n reserves *n* * 32K pages in the relocatable module area (RMA) for modules. If *n* = 0 then the default number is reserved. The maximum allowed is 255 * 32K.

REPEAT n sets the keyboard auto-repeat rate to '*n*'/100 seconds.

SCREENSIZE n reserves '*n*' pages for the screen. The page size and default setting (when *n* = 0) depend on the machine's RAM size. Thus:

RAM	Page	Default
0.5M	8K	80K (ten pages)
1M	8K	160K (20 pages)
4M	32K	160K (5 pages)

SCROLL	disables the scroll protect option
SHCAPS	sets the keyboard SHIFT caps lock option so that alphabetic keys produce upper-case characters and SHIFTed alphabetic keys produce lower-case characters. See NOCAPS and CAPS.
SOUNDDEFAULT sp v1 vo	sets the default sound characteristics. The sp parameter is 0 to disable the internal speaker on reset, 1 to enable it. The v1 parameter sets the overall sound volume, and is in the range 0 (min) to 7 (max). The vo parameter sets the voice which will be assigned to channel 1, and is in the range 1 to 16.
SPRITESIZE n	reserves n * 8K pages for the sprite system. If n = 0 then no sprite space is reserved. The maximum allowed is 255 * 8K.
STEP n	sets the floppy disc drive step rate to <n>. Values are:
SYNC n	selects either separate (n = 0) or composite (n = 1) sync on the monitor.
SYSTEMSIZE n	reserves 'n' pages for the system heap. The page size for a given RAM configuration is as for ScreenSize. If n = 0 then the default number (one page) is reserved. The maximum allowed is 255.

TV n [[,]m]

selects the default setting for the *TV command parameters. See the *TV command details of 'n' and 'm'.

*ECHO

*ECHO outputs characters to the screen.

Given a string, it simply reflects it back. For example,

```
*ECHO text
```

prints the string `text` on the screen.

Certain characters are interpreted as requests for special treatment. For example the angle-bracket characters:

```
*ECHO <19><0><6><0><0><0>
```

This is equivalent to `VDU 19, 0, 6, 0, 0, 0` from BASIC. It can therefore be used to issue any VDU command sequence from the operating system command prompt, or from any language which can issue commands to the OSCLI. It can also have variable references. For example:

```
*ECHO <Sys$Time>
```

*FX

*FX accesses the operating system `OSBYTE` routine. The actions are described in the chapter: ***FX COMMANDS**.

*GO

This command is followed by the address of machine code to call. If the address is omitted, it defaults to `&8000`, which is where application programs (such as RAM BASIC) are loaded.

*GOS

GOS calls the operating system 'supervisor'. This allows you to type '' commands. To return from the supervisor, use the *QUIT command.

*HELP

*HELP provides help information about commands.

*HELP . provides help information on all commands.

*HELP name prints help information on command name. This can include wildcard characters in order to allow commands to be specified precisely.

*IF

The IF command allows you to execute '*' commands conditionally. Its syntax is:

```
*IF <expression> THEN <command> [ELSE <command>]
```

<expression> can be any legal BASIC integer expression, including variable names enclosed in angled brackets. The expression is evaluated by the operating system's expression evaluation (not by BASIC). For example:

```
*IF <Sys$Year>=1987 THEN RUN Calendar
```

*IGNORE

*IGNORE sets the printer ignore character.

*IGNORE n sets the printer ignore character to CHR\$(n).

*KEY

*KEY assigns a string to a function key.

*KEY n text assigns the string text to function key 'n'.

*POINTER

This command is provided by the WIMP manager. If the command is used without any parameters (or with the value 1), mouse pointer one is set to a blue arrow and enabled. *POINTER 0 disables the pointer.

*QSOUND

This command is used to add an item to the queue of sounds to be produced at some later time. The parameters are the same as the *SOUND command described below, with the addition of one parameter giving the number of tempo beats at which to make the sound. The number of beats is relative to the start of the current bar, or relative to the present moment if no BEATS command has been issued. See the BASIC SOUND statement for more details.

*SET

*SET assigns a string to a variable.

*SET varname text assigns string text to variable varname.

Special variables exist which can be assigned values but not deleted. These are:

SYS\$TIME
SYS\$DATE
SYS\$ABORT
SYS\$RETURNCODE

*SET ALIAS\$name cname sets name as the alternative name for the command cname.

Further information is contained in the Reference guide.

***SETEVAL**

*SETEVAL assigns a value to a variable.

*SETEVAL varname exp assigns expression exp to variable varname.

***SETMACRO**

*SETMACRO assigns an expression to a variable.

*SETMACRO varname exp assigns the expression exp to the variable varname. This expression is evaluated each time the variable is used.

***SHADOW**

*SHADOW enables and disables automatic use of shadow memory after a mode change. The command *SHADOW causes the shadow bank (bank 2) to be used; *SHADOW 1 causes the non-shadow bank (bank 1) to be used on the next mode change. To allow the shadow bank to be used, there must be enough memory for two of the screen mode displays. For example, to use a shadow bank in mode 8 (a 40K mode), at least 80K of screen memory must be configured.

***SHOW**

*SHOW lists all variables defined.

*SHOW name lists all variables matching name, which may include wildcards.

***SOUND**

This command is directly equivalent to the BASIC SOUND statement. It takes four parameters: channel, volume, pitch, and duration of the sound. Note that the parameters must be separated by spaces, not commas. Note also that the volume parameter must be unsigned. This means that -15 (loudest) is written as &FFF1, -6 as &FFFA and so on.

*SPEAKER

*SPEAKER enables and disables the internal speaker. When disabled, the speaker is prevented from making any sound, although the 3.5mm stereo jack socket may still be used to play the sound through an amplifier. The parameter to the *SPEAKER command is ON or OFF.

*STATUS

*STATUS displays default values held in battery-backed RAM.

*STATUS displays all values.

*STATUS name displays the value of the *CONFIGURE option name.

*STEREO

This command takes two parameters:

*STEREO <channel> <position>

The <channel> parameter indicates which channel's stereo position is to be set; the <position> parameter sets the stereo position. The ranges are 1 to 8 for the channel, and -127 (full left) to 127 (full right) for the position. See the BASIC STEREO statement for the interpretation of the position parameter.

*TEMPO

The *TEMPO command sets the sound system tempo: the rate at which the beat counter increments. Its parameter is a number which has the same meaning as BASIC's TEMPO parameter. See the BASIC TEMPO statement for more details.

*TIME

*TIME prints the day, date and time.

***TUNING**

This is followed by a number in the range 1 to 32767 to set the overall tuning for the sound system. The default value is 27312. You should avoid the use of this command if possible; a better way of controlling the sound system pitch will be introduced later.

***TV**

*TV specifies the vertical screen alignment and interlace options.

*TV n, 0 adjusts vertically by 'n' lines and turns interlace on.

*TV n, 1 adjusts vertically by 'n' lines and turns interlace off.

***VOICES**

*VOICES displays a list of the installed sound voices by name and number, and shows which voice is attached to each of the eight sound channels. A typical output from the command is:

	Voice	Name
12	1	WaveSynth-Beep
34	2	Percussion-Soft
56	3	Percussion-Medium
78	4	Percussion-Snare
	5	Percussion-Noise

^^^^^^ Channel Allocation Map

***VOLUME**

*VOLUME sets the overall volume for the sound system. It takes a parameter in the range 1 to 127 (softest to loudest). Sound voices use this figure in order to determine the amplitude of the sounds they make. The default setting can be set using the *CONFIGURE SoundDefault command.

*UNSET

*UNSET deletes variables set by *SET, etc.

*UNSET varname deletes all variables matching name varname, which may include wildcards.

In addition to the '*' commands listed above, there are several which relate to the Module manager. These are outside of the scope of the User guide.

*FX COMMANDS

Each *FX command is used to control a particular operating system effect, such as the rate at which flashing colours flash or output to a particular printer. This chapter provides a brief description of the *FX commands which can be used from within BASIC.

*FX 0

*FX 0 displays the operating system title and version number.

Command	Effect
*FX 0,0	Displays the information.

*FX 1

*FX 1 writes to the location left free for the user.

Command	Effect
*FX 1,n	Writes 'n' to the location.

*FX 2

*FX 2 specifies the stream for all subsequent character input.

Command	Effect
*FX 2,0	Selects keyboard input and disables the RS423 port.
*FX 2,1	Selects and enables the RS423 port.
*FX 2,2	Selects keyboard input and enables the RS423 port.

*FX 3






*FX 3 specifies the streams for all subsequent character output.

Command	Effect																		
*FX 3, n	Specifies the stream depending on the bits of 'n' as shown below:																		
	<table><thead><tr><th>Bit</th><th>Effect if set</th></tr></thead><tbody><tr><td>0</td><td>Enables RS423 driver.</td></tr><tr><td>1</td><td>Disables VDU driver.</td></tr><tr><td>2</td><td>Disables printer driver.</td></tr><tr><td>3</td><td>Enables printer (independently of CtrlB and CtrlC).</td></tr><tr><td>4</td><td>Disables spooled output.</td></tr><tr><td>5</td><td>Calls VDUXV instead of VDU driver.</td></tr><tr><td>6</td><td>Disables printer apart from VDU 1, 'n'.</td></tr><tr><td>7</td><td>Not used.</td></tr></tbody></table>	Bit	Effect if set	0	Enables RS423 driver.	1	Disables VDU driver.	2	Disables printer driver.	3	Enables printer (independently of Ctrl B and Ctrl C).	4	Disables spooled output.	5	Calls VDUXV instead of VDU driver.	6	Disables printer apart from VDU 1, 'n'.	7	Not used.
Bit	Effect if set																		
0	Enables RS423 driver.																		
1	Disables VDU driver.																		
2	Disables printer driver.																		
3	Enables printer (independently of Ctrl B and Ctrl C).																		
4	Disables spooled output.																		
5	Calls VDUXV instead of VDU driver.																		
6	Disables printer apart from VDU 1, 'n'.																		
7	Not used.																		






*FX 4

*FX 4 controls the cursor key status.

Command	Effect
*FX 4,0	Enables cursor editing.
*FX 4,1	Disables cursor editing. The cursor keys return ASCII values as shown below:

Key	ASCII code
	135
	136
	137
	138
	139

Command	Effect
*FX 4,2	Disables cursor editing. The cursor keys act as function keys as shown below:

Key	Function key number
	11
	12
	13
	14
	15

*FX 5

*FX 5 selects where subsequent printer output is sent.

Command	Effect
*FX 5,0	Selects printer sink.
*FX 5,1	Selects parallel (Centronics) printer driver.
*FX 5,2	Selects RS423 output.
*FX 5,3	Selects user printer driver.
*FX 5,4	Selects network printer driver.

*FX 6

*FX 6 selects the printer ignore character.

Command	Effect
*FX 6,n	Selects the character with ASCII code 'n'.

***FX 7**

*FX 7 selects the RS423 baud rate for receiving data.

Command	Effect
*FX 7,0	Selects 9600 baud.
*FX 7,1	Selects 75 baud.
*FX 7,2	Selects 150 baud.
*FX 7,3	Selects 300 baud.
*FX 7,4	Selects 1200 baud.
*FX 7,5	Selects 2400 baud.
*FX 7,6	Selects 4800 baud.
*FX 7,7	Selects 9600 baud.
*FX 7,8	Selects 19200 baud.

***FX 8**

*FX 8 selects RS423 baud rate for transmitting data.

Command	Effect
*FX 8,0	Selects 9600 baud.
*FX 8,1	Selects 75 baud.
*FX 8,2	Selects 150 baud.
*FX 8,3	Selects 300 baud.
*FX 8,4	Selects 1200 baud.
*FX 8,5	Selects 2400 baud.
*FX 8,6	Selects 4800 baud.
*FX 8,7	Selects 9600 baud.
*FX 8,8	Selects 19200 baud.

*FX 9

*FX 9 selects the flash rate for first flashing colour.

Command	Effect
*FX 9, 0	Sets constant display of first colour.
*FX 9, n	Sets the duration to approximately 'n'/50 seconds.

*FX 10

*FX 10 selects the flash rate for second flashing colour.

Command	Effect
*FX 10, 0	Sets constant display of second colour.
*FX 10, n	Sets the duration to approximately 'n'/50 seconds.

*FX 11

*FX 11 selects the keyboard auto-repeat delay.

Command	Effect
*FX 11, 0	Disables auto-repeat.
*FX 11, n	Sets the auto-repeat delay to 'n'/100 seconds.

*FX 12

*FX 12 selects the keyboard auto-repeat rate.

Command	Effect
*FX 12, 0	Sets rate and delay to their configured settings.
*FX 12, n	Sets the auto-repeat rate to 'n'/100 seconds.

***FX 15**

*FX 15 flushes one or more buffers.

Command	Effect
*FX 15,0	Flushes all buffers.
*FX 15,1	Flushes the current input buffer.

***FX 18**

*FX 18 resets the function keys.

***FX 19**

*FX 19 waits for vertical sync (vsync).

***FX 20**

*FX 20 resets the font definitions for chars 32 to 127.

***FX 21**

*FX 21 flushes a selected buffer.

Command	Effect
*FX 21,0	Flushes keyboard buffer.
*FX 21,1	Flushes RS423 input buffer.
*FX 21,2	Flushes RS423 output buffer.
*FX 21,3	Flushes printer buffer.
*FX 21,4	Flushes sound channel 0.
*FX 21,5	Flushes sound channel 1.
*FX 21,6	Flushes sound channel 2.
*FX 21,7	Flushes sound channel 3.
*FX 21,8	Flushes speech buffer.
*FX 21,9	Flushes mouse buffer.

*FX 25

*FX 25 resets a group of font definitions.

Command	Effect
*FX 25, 0	Restores characters 32 to 255.
*FX 25, 1	Restores characters 32 to 63.
*FX 25, 2	Restores characters 64 to 95.
*FX 25, 3	Restores characters 96 to 127.
*FX 25, 4	Restores characters 128 to 159.
*FX 25, 5	Restores characters 160 to 191.
*FX 25, 6	Restores characters 192 to 223.
*FX 25, 7	Restores characters 224 to 255.

*FX 106

*FX 106 selects the pointer or activates the mouse.

Command	Effect
*FX 106, 0	Turns off current pointer.
*FX 106, n	Selects pointer 'n' and links it to the mouse.
*FX 106, n+128	Selects pointer 'n' and unlinks it from the mouse.

*FX 112

Write screen bank number addressed by VDU driver.

Command	Effect
*FX 112, 0	Selects default for current mode.
*FX 112, n	Selects bank 'n'.

***FX 113**

Write screen bank number addressed by display hardware.

Command	Effect
*FX 113, 0	Selects default for current mode.
*FX 113, n	Selects bank 'n'.

***FX 114**

Sets up automatic shadow mode.

Command	Effect
*FX 114, 0	Subsequent MODE n commands to be interpreted as MODE n+128.
*FX 114, 1	Cancel the above command.

***FX 118**

*FX 118 reflects the keyboard status in the LEDs.

***FX 120**

*FX 120, x, y writes the internal key numbers of the two most recently pressed keys. These are 'x' (the most recent key) and 'y' (the original key).

***FX 124**

*FX 124 clears an escape condition.

***FX 125**

*FX 125 sets an escape condition.

*FX 126

*FX 126 acknowledges an escape condition.

*FX 138

*FX 138 inserts a character code into a buffer.

Command	Effect
*FX 138, 0, n	Inserts CHR\$(n) into keyboard buffer.
*FX 138, 1, n	Inserts CHR\$(n) into RS423 input buffer.
*FX 138, 2, n	Inserts CHR\$(n) into RS423 output buffer.
*FX 138, 3, n	Inserts CHR\$(n) into printer buffer.
*FX 138, 4, n	Inserts CHR\$(n) into sound channel 0.
*FX 138, 5, n	Inserts CHR\$(n) into sound channel 1.
*FX 138, 6, n	Inserts CHR\$(n) into sound channel 2.
*FX 138, 7, n	Inserts CHR\$(n) into sound channel 3.
*FX 138, 8, n	Inserts CHR\$(n) into speech channel.
*FX 138, 9, n	Inserts CHR\$(n) into mouse buffer.

*FX 139

*FX 139, x, y sets the filing system options. It is exactly the same as *OPT x, y. You should, therefore, refer to that command for details (in the chapter: **FILING SYSTEMS**).

*FX 143

This call issues a module service call. See the section on **Modules** in the **Reference guide** for details.

*FX 144

*FX 144, x, y is exactly equivalent to *TV x, y.

***FX 153**

*FX 153 inserts a character into an input buffer, checking for escape.

Command	Effect
*FX 153, 0, n	Inserts CHR\$(n) into keyboard buffer.
*FX 153, 1, n	Inserts CHR\$(n) into RS423 input buffer.

***FX 156**

This command controls the RS423 port. In the command *FX 156, x, you may regard 'x' as an eight-bit number. The meanings of the bits are as follows:

Any command with bits 0 and 1 both set cause the RS423 port to be reset. Any other combination of bits 0 and 1 is ignored.

Bits 2 to 4 form a three bit number which controls the data format used by the RS423 port. See the *CONFIGURE Data command for the eight possible values that these bits may take on.

Bits 5 and 6 control the RS423 transmitter:

bit 6	bit 5	
0	0	RTS low, transmit interrupt disabled
0	1	RTS low, transmit interrupt enabled
1	0	RTS high, transmit interrupt disabled
1	1	RTS high, transmit interrupt disabled

Bit 7 controls the receiver interrupt; if it is set, the interrupt is enabled.

***FX 162**

*FX 162 writes a value to the battery-backed CMOS RAM. The call

*FX 162, x, y

writes the value 'y' into location 'x' of the CMOS RAM. See the Reference guide for details of the locations available for use.

*FX 163

*FX 163 sets the dot-dash line pattern length.

Command	Effect
*FX 163,242,0	default pattern and length.
*FX 163,242,n	line repeat length set to 'n'.

*FX 178

This may be used to enable or disable the keyboard, as shown below:

Command	Effect
*FX 178	Disables the keyboard.
*FX 178,255	Enables the keyboard.

*FX 181

*FX 181 sets the way in which the RS423 input port is handled. It normally differs from the keyboard in that the escape character does not have any effect, function key codes are not 'expanded' into strings, and input events are not generated. Using *FX 181 you can make the RS423 port act as the keyboard, so that all of these things do happen. Thus:

Command	Effect
*FX 181	Makes the RS423 act like the keyboard.
*FX 181,1	Treats the RS423 in the normal way.

*FX 196

This is a synonym for *FX 11.

***FX 197**

This is a synonym for *FX 12.

***FX 200**

*FX 200 selects the **Break** and **Escape** effects.

Command	Effect
*FX 200, 0	Normal Escape and Break action.
*FX 200, 1	Escape disabled, normal Break action.
*FX 200, 2	Normal Escape action, Break clears memory.
*FX 200, 3	Escape disabled, Break clears memory.

***FX 201**

*FX 201 sets the keyboard status.

Command	Effect
*FX 201, 0	Enables keyboard input.
*FX 201, 1	Disables keyboard input.

***FX 202**

*FX 202 alters the keyboard status byte.

Command	Effect
*FX 202, n	Indicates the status depending on the bits of 'n' as shown below:

Bit Indication when set

0	Alt pending
1	Scroll Lock engaged
2	Num Lock disengaged
3	Shift depressed
4	Caps Lock disengaged
5	reserved
6	Ctrl depressed
7	Shift enabled

*FX 203

*FX 203, n sets the number of free spaces which must be left in the RS423 input buffer before the Archimedes tells the remote machine to stop sending. It defaults to 9, but may be set to any value between 1 and 255.

*FX 204

This determines whether characters received by the RS423 port will be placed into the buffer or discarded:

Command	Effect
*FX 204	Insert received characters into buffer.
*FX 204, 1	Ignore received characters.

Note that RS423 events may still occur, even when characters are not being buffered.

***FX 211**

*FX 211 selects the bell channel number.

Command	Effect
---------	--------

*FX 211, n	Sets the bell channel number to 'n'.
------------	--------------------------------------

***FX 212**

*FX 212 selects the bell amplitude.

Command	Effect
---------	--------

*FX 212, n	Sets the bell amplitude to 'n'.
------------	---------------------------------

***FX 213**

*FX 213 selects the bell frequency.

Command	Effect
---------	--------

*FX 213, n	Sets the bell frequency to 'n'.
------------	---------------------------------

***FX 214**

*FX 214 selects the bell duration.

Command	Effect
---------	--------

*FX 214, n	Sets the bell duration to 'n'/20 seconds.
------------	---

***FX 216**

*FX 216 may be used to cancel the reading of a function key string. If a function key has been pressed thereby causing subsequent keyboard input be read from

that key's string definition, this command will cause the string to be ignored, and input to come from the next key to be pressed.

*FX 217

This command sets the 'paged mode line count'. This count determines how many more lines the screen may scroll before the **Shift** key must be pressed. Using this command to set the count to zero ensures that a program does not 'freeze' while waiting for **Shift** to be pressed. Note that it is not necessary to use this command before a BASIC INPUT statement since this is done automatically for you.

*FX 218

*FX 218 cancels the current VDU command sequence, just as *FX 216 cancels the current function key input string. Its main use is in machine code programs' error handling routines, in order to abort any VDU sequence which is partially completed when an error occurred. You should never have to use it in BASIC.

*FX 219

*FX 219 selects the **Tab** key code.

Command	Effect
---------	--------

*FX 219, n	Sets the ASCII code generated by Tab to 'n'.
------------	---

*FX 220

*FX 220 selects the **Escape** character.

Command	Effect
---------	--------

*FX 220, n	Sets the ASCII code of the Escape key to 'n'.
------------	--

*FX 221

*FX 221 selects the interpretation of input values 192 to 207.

Command	Effect
*FX 221, 0	Ignores the code
*FX 221, 1	Code generates the corresponding function key string.
*FX 221, 2	Code generates a zero followed by $128 + (\text{code MOD } 16)$.
*FX 221, n	Code generates the value $'n'(3-225) + (\text{code MOD } 16)$.

*FX 222

*FX 222 selects the interpretation of input values 208 to 223.

Command	Effect
*FX 222, 0	Ignores the code.
*FX 222, 1	Code generates the corresponding function key string.
*FX 222, 2	Code generates a zero followed by $128 + (\text{code MOD } 16)$.
*FX 222, n	Code generates the value $'n'(3-225) + (\text{code MOD } 16)$.

*FX 223

*FX 223 selects the interpretation of input values 224 to 239.

Command	Effect
*FX 223, 0	Ignores the code.
*FX 223, 1	Code generates the corresponding function key string.
*FX 223, 2	Code generates a zero followed by $128 + (\text{code MOD } 16)$.
*FX 223, n	Code generates the value $'n'(3-225) + (\text{code MOD } 16)$.

*FX 224

*FX 224 selects the interpretation of input values 240 to 255.

Command	Effect
*FX 224, 0	Ignores the code.
*FX 224, 1	Code generates the corresponding function key string.
*FX 224, 2	Code generates a zero followed by 128 + (code MOD 16).
*FX 224, n	Code generates the value 'n'(3-225) + (code MOD 16).

*FX 225

*FX 225 selects the function key interpretation.

Command	Effect
*FX 225, 0	Ignores the key depression.
*FX 225, 1	Key generates the corresponding function key string.
*FX 225, 2	Key generates a zero followed by 128 + (code MOD 16).
*FX 225, n	Key generates the value 'n'(3-225) + (code MOD 16).

*FX 226

*FX 226 selects the Shift plus the function key interpretation.

Command	Effect
*FX 226, 0	Ignores the key depression.
*FX 226, 1	Key generates the corresponding function key string.
*FX 226, 2	Key generates a zero followed by 128 + (code MOD 16).
*FX 226, n	Key generates the value 'n'(3-225) + (code MOD 16).

*FX 227

*FX 227 selects the **Ctrl** plus the function key interpretation.

Command	Effect
*FX 227, 0	Ignores the key depression.
*FX 227, 1	Key generates the corresponding function key string.
*FX 227, 2	Key generates a zero followed by 128 + (code MOD 16).
*FX 227, n	Key generates the value 'n'(3-225) + (code MOD 16).

*FX 228

*FX 228 selects the **Shift Ctrl** plus the function key interpretation.

Command	Effect
*FX 228, 0	Ignores the key depression.
*FX 228, 1	Key generates the corresponding function key string.
*FX 228, 2	Key generates a zero followed by 128 + (code MOD 16).
*FX 228, n	Key generates the value 'n'(3-225) + (code MOD 16).

*FX 229

*FX 229 selects the **Escape** key status.

Command	Effect
*FX 229, 0	Makes escape character generate an escape condition.
*FX 229, n	Makes escape character generate its ASCII code.

*FX 230

*FX 230 selects the `Escape` effects.

Command	Effect
*FX 230,0	Enables side effects.
*FX 230,n	Disables side effects.

These effects are:

- All active buffers are flushed
- Any currently open *EXEC file is closed
- The VDU queue is cleared
- The VDU line count used in paged mode is cleared
- Any sound being produced is terminated.

The `Escape` effects mentioned only take place when the escape condition is acknowledged (using *FX 126), not when the escape condition first occurs. If you use *FX 124 to clear the escape condition, the effects will never occur.

*FX 238

*FX 238 selects the numeric keypad interpretation.

Command	Effect
*FX 238,n	Resets base value for keypad codes to 'n'.

*FX 247

*FX 247, x controls the effects of pressing the `Break` key on the operation of the machine. The parameter 'x' consists of 4 two-bit numbers. Bits 0 and 1 control `Break`; bits 2 and 3 control `Shift Break`; bits 4 and 5 control `Ctrl Break`, and bits 6 and 7 control `Ctrl Shift Break`.

Each two-bit number may take on one of these values:

Value	Effect
0	Perform reset.
1	Perform Escape.
2	No effect.
3	Undefined.

The default value of 'x' is 1, so **Break** causes an Escape condition, and all other combinations cause a reset.

*FX 254

*FX 254 selects the effect of **Shift** on the numeric keypad.

Command	Effect
*FX 254, 0	Enables the effect of Shift
*FX 254, n	Disables the effect of Shift

*FX 255

*FX 255 selects the **Break** and **Shift Break** startup options.

Command	Effect
*FX 255, 0	Interchanges the effects
*FX 255, 8	Restores normal functions

In addition to the *FX calls mentioned in this chapter, there are several more which may be used to read information instead of writing it. To access these, and other operating system routines, you must use the **SYS** statement in BASIC. Detailed descriptions of the OS routines can be found in the Reference guide.


THE BASIC SCREEN EDITOR

The screen editor allows you to move around and change any part of a program currently loaded in the Archimedes.

ENTERING THE EDITOR

To enter the screen editor from BASIC type

```
EDIT
```

and press .

This command enters the editor with the current program.

The editor tries to re-enter the program at the point at which you left it. If you have changed the program from within BASIC, it may not be possible to maintain the context, in which case editing starts from the top of the program.

If you wish to enter the editor at a particular point, such as line 100, type

```
EDIT 100
```

BASIC enters the editor with line 100 displayed at the top of the screen. If line 100 does not exist, the editor chooses either the next line or the end of the program.

You may wish to enter the editor with the first occurrence of a particular piece of text at the top of the screen. For example:

```
EDIT three
```

The editor displays the program starting with the first occurrence of the word `three` at the top of the screen. If the string cannot be found, the computer 'beeps' and editing starts at the top of the program.

THE EDIT SCREEN

Once in the editor, your program is displayed with the line numbers at the left hand side. If you enter the editor with no program loaded the screen is nearly blank, with just the number 10 at the top left.

The cursor is at the beginning of the top line on the screen, just to the right of the line number. Note that the editor automatically puts a line number on the beginning of each line: there is no need for you to type them in.

The status line

The status line is at the bottom of the screen, displayed in reversed colours in order to make it stand out from your program text. It contains various useful pieces of information such as the size of your program, its name, and whether it has been modified since you entered the editor.

The status line displays the following information (if it will fit):

- Program size
- Original/Modified indicator
- Program name
- Copy if in cursor copy mode.

In addition, the status line is used for prompts such as `Replace? (Y/N)` which appear in the **SELECTIVE REPLACE** facility. See the section: **Searching and replacing** below for details.

Moving the cursor

The cursor can be moved around using the four arrow keys. Note, however, that you cannot move the cursor into that area of the screen containing the line numbers. This is because in general you need never be concerned with providing line numbers for your BASIC statements. As a result, cursor movement is restricted to the area of the screen which contains program text.

Changing a line

To change a line, use the cursor keys to position the cursor on the correct line. You can then delete part or all of the line and type new text in place of the old.

Now, assume that the program looks like this:

```
10 FOR X = 2 TO 30
20 PRINT X+X
30 NEXT X
```

and that it needs to be changed to look like this:

```
10 FOR X = 2 TO 20
11 PRINT X*X
20 PRINT X+X
30 NEXT X
```

To achieve this you must change line 10 and add a new line: line 11.

Position the cursor on the '0' of 30 on line 10, press **Delete** and type '2'. The 30 is replaced by 20.

Adding a line


To create a new line in the middle of the program move the cursor to the line above the place where you want the new line and press **Enter**.

In the example above, move the cursor to line 10 and press **Enter**.



Line 11 is now created.

To complete the above program type


```
PRINT X*X
```







The program should now be complete. You may like to experiment with the  and cursor keys to create a larger program.


Inserting lines at the top or bottom of a program.

There are two function keys which, no matter where you are in the program, create a new line at the top or end of the program and move you there directly. These keys are  (INSERT AT START) and  (INSERT AT END).

Deleting text

There are two ways to delete single characters. The  key removes the character to the left of the cursor and moves the characters to the right of the cursor back one space.

To delete the character immediately above the cursor, hold the  key down and press the  key.  and  both move the following text back a space, but  leaves the cursor in the same position.


To delete all the characters from the cursor position to the end of the line, press the  key.

Long lines

If a statement is too long to fit on one line of the screen, it wraps around to the next line. To see this, try typing more text after one of the lines in the program. As in a BASIC program, the length of a line is limited by the BASIC editor to 251 characters.

SAVING AND LOADING PROGRAMS

Saving a program

To save a program which you have created or changed press  (SAVE).

A window appears into which you should type the name of the program. Once you are sure that you have typed the correct name for the program press **↵** or **f12** (EXECUTE) to perform the save operation.

The program name need not be enclosed within quotation marks.

If you wish to save only a portion of a program you may do this by setting limits. See the section on **Line Commands** below for details of how to do this.

Loading a program

You may now wish to load in one of your own programs to experiment with before moving on to the next chapter. To do this press **f2** (LOAD).

A window appears ready to accept the filename.

Type in the name of the program and press **↵** or **f12** (EXECUTE).

If the current program has been modified but not saved a warning message is given.

Appending a program

You can also join one program onto the end of the current one.

To do this press **Shift f2** (APPEND) and then proceed in the same manner as for loading.

SEEING OTHER PARTS OF YOUR PROGRAM

Several commands are provided to help you move quickly around when you are editing a large program, such as one which is too large to be displayed on the screen at one time.

Moving vertically

If you move the cursor to the top screen line and keep pressing the **↑** key, previous statements are brought onto the screen one at a time until you reach the beginning of the program. Similarly, pressing **↓** from the bottom screen line brings the following statements onto the screen one at a time until you reach the end of the program.

To move directly to the top of your program, press **Ctrl ↑** which moves the cursor to the first line of the program. Pressing **Ctrl ↓** moves to the last line.

If you press **Shift ↓** and the **↓** key, the next screenful of your program is displayed. In this way, you can move quickly around your program from beginning to end. Similarly, if you press **Shift ↑**, you can see the previous screenful. These functions are duplicated by the **Page Up** and **Page Down** keys.

Moving horizontally

Pressing the **Shift** key together with the **→** and **←** keys enables you to move sideways across the screen at twice the normal speed.

Pressing **Ctrl ←** takes you to the beginning of the previous statement and **Ctrl →** takes you to the end of the current line.

RENUMBERING THE PROGRAM

If new lines are created in the middle of a program, the editor automatically adjusts the numbering where necessary. If this happens in a program containing a GOTO or a GOSUB to a line number as yet non-existent, then that line number is replaced by the characters @@@.

You may at any time renumber the program yourself by pressing **rfB** (RENUMBER). This renumbers the program starting at line 10 with an increment of 10.

Further editing functions

Swapping case

If you have typed in some text in either upper or lower case and you want to change it to the opposite case, move to the area to be changed and press **[F10]** (SWAP). This converts one alphabetic character at a time from lower case to upper case and vice versa.

Undoing changes to a line

If you want to abandon any changes you have made to a statement before you have left it, press **[Shift][F10]** (UNDO). This restores the statement to the way it was before you made the changes.

Splitting and joining lines

Occasionally, you may want to split one statement into two or more. You can do this by positioning the cursor on the character which is to be at the start of the new statement and pressing **[Shift][F1]** (SPLIT). You can only split a statement from somewhere in the middle. As you are creating a new statement, this may cause renumbering to take place.

There may also be occasions when you want to join two statements together. To do this, move the cursor to the first of the two statements and press **[Ctrl][F1]** (JOIN). The editor automatically puts a colon between the two statements. If the combined length of the two statements would exceed the maximum space available, the join is not carried out and an error message is displayed.

Repeating a line

To create an exact copy of any statement immediately after it, move to the statement you wish to copy and press **[Shift][F8]** (REPEAT). As in the case of SPLIT, this may cause renumbering to be carried out.

MARKING A LINE

Placing the marker line

As you move about your program, there may be a statement which you wish to come back to later on. The editor provides a way of marking a statement so that you can go back to it with a single key-stroke. To mark a statement, first move to it and press **f6** (TOGGLE MARK). Pressing the same key again removes the marker. A full stop appears on the screen between the line number and the start of the text, indicating that this statement has been marked. Up to four marks may be set at any time.

Finding a marker

Wherever you are in the program, pressing **Shift f6** (GOTO MARK) brings the marked statement to the top of the screen and positions the cursor there. If there is no marked line, pressing GOTO MARK displays an error; pressing **Escape** then allows you to continue.

LINE COMMANDS

These are commands which allow you to delete, move and copy either a single line or a block of lines. They can be inserted into the left-hand margin and are not executed until **f12** (EXECUTE) is pressed.

For example, to delete a single line, move the cursor onto that statement, hold down the **Ctrl** key and press 'D'. The line number is removed and replaced by the letter 'D'. To delete the line from your program, press **f12** (EXECUTE). The line is removed from the screen and the cursor positioned on the previous line.

Deleting lines

If there is a block of lines which you want to delete, move to the first line in the block and press **Ctrl D** twice. The line number disappears and is replaced by the letters DD. Now move to the last line in the block and press **Ctrl D** twice more. Finally, press **f12** (EXECUTE) to remove this block of lines from your program.

You may wish to delete from the current line to the end of the program. In this case, press **Ctrl**D twice on the current line and then press **Ctrl**E. The line number is replaced by DDE and the block from there to the end of the program can be removed by pressing **F12** (EXECUTE).

In a similar way, you can delete from the current line to the top of the program by using **Ctrl**T instead of **Ctrl**E and then pressing **F12** (EXECUTE).

CtrlE and **Ctrl**T are examples of destinations and we shall encounter more of these later.

Moving a block

To move a single statement from its current position to the end of the program, move to it and press **Ctrl**M followed by **Ctrl**E. The line number is replaced by ME and pressing **F12** (EXECUTE) moves that line to the end of the program.

CtrlT can be used likewise to move a statement to the top of a program.

Instead of using **Ctrl**T or **Ctrl**E to specify the destination as the top or the end of the program you can specify that the destination is before or after a certain line.

To move text to a position after a particular line, move to the destination and press **Ctrl**A.

Alternatively you can use **Ctrl**B to move text to a position before a particular line.

Blocks of lines can be moved as easily as a single line by putting MM around the block to be moved, choosing your destination, and pressing **F12** (EXECUTE).

Copying lines

Whereas moving text removes it from its original position, copying text leaves the original unchanged and duplicates it elsewhere. The command to copy text is **Ctrl**C instead of **Ctrl**M, but otherwise the move and copy commands are the same.

Naturally, for both the move and copy commands the destination must not be within the block being moved or copied.

Denoting limits

You can limit the effect of certain operations either to one line or to a block of lines. These operations are:

- SAVE: Part of a program can be saved.
- RENUMBER: Part of the program is renumbered.
- SEARCH, SEARCH & EDIT: The search is limited to the line or block.
- SELECTIVE REPLACE, GLOBAL REPLACE: The replacement is limited to the line or block.

To limit the operation to a single line, move the cursor to that line and press **Ctrl**L. To delimit an entire block of lines press **Ctrl**L twice each on the first and last line of the block in question.

When a limit is set up, the functions which take account of it display the limit in their window.

Justifying text

The editor can indent all or part of a program automatically. To reformat a part of the program, move to the first line of the block you want to justify and press **Ctrl**J twice. Then move to the last line of the block and press **Ctrl**J twice more. Pressing **F12** (EXECUTE) justifies the block so that the indentation of each line is identical to that of the first line.

Removing line commands

To remove a line command, move to the line in question and press **Ctrl**R. This deletes the line command from the screen and replaces the line number. Pressing **Ctrl**R on a line which does not contain any line commands removes all line

commands no matter where they are. You do not, however, have to remove a line command in order to change it: to replace the old command simply overtype it with a new one.

CtrlR can also be used to remove the line marker set by **f6** (TOGGLE MARK); but unlike the line commands, the marker can only be removed when you are on the marked statement.

Things to notice about line commands

Line commands are not stored as part of your program text but are only held internally in the editor. There is no need, therefore, to remove line commands or the marker before saving your program.

Note that copying or moving statements causes renumbering to take place automatically.

SEARCHING AND REPLACING

SEARCH & EDIT

To search for the first occurrence of a particular piece of text, press **f4** (SEARCH & EDIT). A window appears where you should enter the text to be found. When you have done this press **f12** (EXECUTE) and the search is carried out. The cursor reappears on the first match within the program.

SEARCH

As an alternative to SEARCH & EDIT you can find all occurrences of a given string and have them displayed. To do this press **f7** (SEARCH) and enter the string which is to be located. Then press **f12** (EXECUTE) to perform the search. Any line on which a match is found is displayed. You may then move up and down the list, choose one to look at and press **Home**. This line is then placed at the top of the full edit screen and you can edit it.

GLOBAL REPLACE

To change one string for another throughout your entire program press **f5** (GLOBAL REPLACE) and enter the text to be changed. You must then enter the new text, and when you are happy with it press **f12** (EXECUTE) to carry out the change.

SELECTIVE REPLACE

It is possible to perform a replace operation selectively. To do so press **Shift f5** (SELECTIVE REPLACE). You must then enter both the text to be changed and the new text. Press **f12** (EXECUTE) to start the search. Each match is displayed and you are prompted for either 'Y' or 'N' to indicate whether the replacement is to be performed or not.

NEXT MATCH and PREVIOUS MATCH

It is possible to move on to the next occurrence of the text searched for in the last search operation or back to the previous one. To do this press either **Shift f7** (NEXT MATCH) or **Ctrl f7** (PREVIOUS MATCH).

SETTING VARIOUS OPTIONS

Pressing **Shift f3** brings up a window which allows you to select various options. This is called the Options Window. The options are displayed in three groups described below. Pressing **Enter** allows you to cycle through the groups.

Keyboard options

The **Tab** key: This enables you to move more quickly across the screen. It moves the cursor to every fifth character position. At the end of a line, it takes the cursor to the beginning of the next line.

Pressing **Shift Tab** moves the cursor in the opposite direction.

The options can be used to set the width of the tab movement to any value (numbers of characters) in the range 0 to 63.

Auto indentation: The editor can automatically line up text in a program so that each line starts beneath the first position of the line above which is not blank. This is known as auto-indentation. It can be turned on or off using the Options Window:

Auto-indent (on/off)

Insert mode vs overtype mode: There will be times when you want to insert new text into a line rather than overtype what is already there. To do this, press `Insert` and you will see that the cursor has changed to a block. This indicates that you are in insert mode, and that text which you type in will move any following text to the right. To return to overtype mode, press `Insert` again, and you will be able to overtype text as before. In overtype mode, an underline cursor is used. In insert mode, a block cursor is used.

When you enter the editor, the default setting (insert or overtype) is used. You can change this default using the Options Window. Your choice is retained in non-volatile memory.

Wildcard options

There are four wildcards, each of which may be customised using the options available.

- Single character (default is '.').
- Multiple characters (default is '|').
- Start case insensitivity: this will match both `PRINT` and `print` (default is '{').
- End case insensitivity: this will match exactly what is entered. This is the normal method of searching (default is '}').

Wildcards can be changed to any punctuation character, or can be disabled by using the Space Bar. Different wildcards must not use the same character.

Mode and colours

The editor works in 40-, 80- or 132-column modes. You can choose the default mode using the Options Window. The value is held between sessions in non-volatile memory.

Note that 256-colour modes and modes with 20-column text are not allowed.

You can also set up your default choice of foreground and background colours.

USER DEFINED KEYS

The editor makes extensive use of the normal function keys, but you can still program your own in the usual way via the *KEY command. To access them you must press **Ctrl****Shift** together with the function key, and not just the function key on its own.

FULL USE OF WINDOWS

Windows are displayed whenever user input is required or information is displayed.

Input windows

Valid keys and their actions are:

Tab / ↵ / ↵	Moves the cursor to the next field
Shift Tab / ↑	Moves the cursor to the previous field
Escape	Cancel the window and returns to editing
EXECUTE f12	Validates the input and executes the command
Insert	Toggles insert/overtyping for this window only

Delete	Deletes the character to the left of the cursor
Shift>Delete	Deletes the character above the cursor
DELETE TO END OF LINE f11	Deletes characters from the cursor to the end of the field
DELETE TO START OF LINE Shift f1 f1	Deletes all characters before the cursor
DELETE LINE Ctrl f11	Deletes all text in this field
←/Shift ←	Moves the cursor left 1 or 2 positions
→/Shift →	Moves the cursor right 1 or 2 positions
Ctrl ←	Moves the cursor to the beginning of the field
Ctrl →	Moves the cursor to the end of the field

Information windows

Escape	Removes the window and returns to editing.
---------------	--

Entering data

Data can be entered in one of three ways:














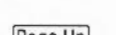
- Typing in text (eg program name)
- Selecting a prompted action (eg Y/N)
- Pressing the Space Bar to cycle through a list of valid choices (eg foreground colour)

Pressing another function key whilst a window is present usually executes its function. The exceptions are those functions which manipulate the program text (eg SPLIT and JOIN).

KEYBOARD SUMMARY

The following actions are performed directly via key presses:

Editing keys

	Moves right
	Moves left
	Moves up
	Moves down
	Moves right two characters
	Moves left two characters
	Moves cursor up a screenful
	Moves cursor down a screenful
	Moves to the end of the statement
	Moves to the beginning of the statement
	Moves to the beginning of the program
	Moves to the end of the program
	Moves cursor up a screenful
	Moves cursor down a screenful

Tab	Moves right to next tab position
Shift Tab	Moves left to previous tab position
Home	Brings statement to top of screen
Copy	Enters copy mode
Escape	Ends copy mode
Insert	Toggles insert/overtyping mode.
Delete	Deletes character to left of cursor
Shift Delete	Deletes character at cursor position
Enter	Creates a new statement after the current one

Function keys

f1 (* COMMAND)	Perform OS command
f2 (LOAD)	Load a program
f3 (SAVE)	Save a program
f4 (SEARCH & EDIT)	Find string and edit from it
f5 (GLOBAL REPLACE)	Global search and replace
f6 (TOGGLE MARK)	Set or remove a marker. Up to four markers are allowed
f7 (SEARCH)	Find all occurrences of a string.
f8 (RENUMBER)	Renumber the entire program

f9 (OLD)	Same as BASIC OLD
f10 (SWAP)	Swap case of alphabetic characters
f11 (DELETE TO END OF LINE)	Delete from cursor to end of line
f12 (EXECUTE)	Execute line commands
Function keys with Shift	
Shift f1 (SPLIT)	Split statement at the cursor
Shift f2 (APPEND)	Append a program
Shift f3 (OPTIONS)	Present the Options Window
Shift f4 (EXIT)	Return to BASIC. Any variables will be lost if changes were made
Shift f5 (SELECTIVE REPLACE)	Selective replace. When prompted, only 'Y', 'N', Escape and Home are valid
Shift f6 (GOTO MARK)	Go to next marker, with program wraparound
Shift f7 (NEXT MATCH)	Go to next occurrence of search string
Shift f8 (REPEAT)	Copy current statement
Shift f9 (NEW)	Same as BASIC NEW. Prompts if program has been modified
Shift f10 (UNDO)	Undo changes to current statement
Shift f11 (DELETE TO START OF LINE)	Delete all characters before the cursor

Shift f12 (GOTO LINE COMMAND)	Go to next line command, with program wraparound
Function keys with Ctrl	
Ctrl f1 (JOIN)	Join two statements, with a colon separator
Ctrl f2	Reserved
Ctrl f3	Reserved
Ctrl f4	Reserved
Ctrl f5	Reserved
Ctrl f6	Reserved
Ctrl f7 (PREVIOUS MATCH)	Go to previous occurrence of search string
Ctrl f8 (EXTEND)	Add a line to current statement
Ctrl f9 (INSERT AT START)	Add a statement at beginning of program
Ctrl f10 (INSERT AT END)	Add a statement at end of program
Ctrl f11 (DELETE LINE)	Delete all text from current statement
Ctrl f12 (GO TO LINE)	Go to selected line number

Function keys are used with **Ctrl** and **Shift** for user-defined strings.

ERROR MESSAGES

The editor displays the following messages. In each case, an explanation is given below the message.

Limit is xxxx to xxxx/Limit is xxxx only

A range has been set using the L or LL line commands, and this function will only operate within the range.

Line xxxx is too long to be edited

The program already contains a line which is too long.

Not enough room in RMA for The BASIC Editor

RMA initialisation failed to acquire workspace.

Replace? (Y/N)

Displayed on the status line when prompting during the SELECTIVE REPLACE operation.

Tab must be between 0 and 63

Displayed by OPTIONS.

The combined length of these statements would be too big

The two statements cannot be joined.

The destination must be outside the block being moved or copied

Raised by EXECUTE.

The first statement in the block to be justified must not be blank

Raised by EXECUTE.

The maximum line is 65279

Raised by GOTO LINE.

The name has been truncated

On saving, the program name following REM > in the first line of the program is longer than can be displayed in the window.

The named program is invalid

The user appended a program which was invalid. The editor restored the original.

The named program is too big

The user tried to load or append a program for which there was not enough room in memory

The renumber has failed. Unmatched line numbers have been replaced by @@@@

When trying to renumber the program one or more line number references could not be resolved.

The search string has no text

The search string must not be blank, and must not contain only wildcards.

The string could not be found

The search string could not be found.

There is not enough memory to update the program

All available memory has been used up.

This is not a valid mode

An invalid screen mode was specified in OPTIONS.

This is not a valid program

OLD was pressed with no valid BASIC program in memory, or the user tried to load an invalid program.

This program could not be found

The named program on a load or append was not in the directory.

This program has not been saved

The user is warned on a load if the program has been modified and not saved.

This program has not been saved

Press NEW again to confirm

Press ESCAPE to cancel

The user pressed NEW but the program had been modified and not saved.

This statement is too long

The statement is too long, and needs to be shortened.

This statement is too long to be changed

Replacing or justifying would make the statement too long.

This statement is too long to be split

Even after splitting, both parts of the statement would still be too long.

Wildcards must not be the same

Raised by OPTIONS.

You cannot load a directory

The filename specified in LOAD or APPEND is a directory.

You do not need to enter a destination for this command

Raised by EXECUTE.

You do not need to enter a repetition factor for this command

Raised by EXECUTE.

You have entered a destination but no command

Raised by EXECUTE.

You have entered too many commands

Raised by EXECUTE.

You have not entered any line commands

Raised by GOTO LINE COMMAND when there are no line commands.

You have not entered any markers

Raised by GOTO MARKER when no markers are set.

You have not yet entered a search string

Raised by NEXT MATCH or PREVIOUS MATCH when no find string has been entered.

You have used the maximum number of statements. No more can be added

The program already contains the maximum number of statements allowed by BASIC (65279) and the user tried to add another.

You must enter a destination for this command

Raised by EXECUTE.

You must enter a mode

No screen mode was specified within OPTIONS.

You must enter a program name

The program name was not entered for LOAD, APPEND or SAVE.

You must enter a search string

The search string was not entered.

You must enter a tab value

No tab value was specified in OPTIONS.

You need to specify both ends of the range for this command

Raised by EXECUTE.

You should not enter two different commands

Raised by EXECUTE.

*ARMBE is only valid from BASIC

The user invoked the editor from outside BASIC.

APPENDIX A - MINIMUM ABBREVIATIONS

Command	Abbreviation
ABS	ABS
ACS	ACS
ADVAL	AD.
AND	A.
APPEND	AP.
ASC	ASC
ASN	ASN
ATN	ATN
AUTO	AU.
BEAT	BEAT
BEATS	BEA.
BGET	B.
BPUT	BP.
CALL	CA.
CASE	CASE
CHAIN	CH.
CHR\$	CHR\$
CIRCLE	CI.
CLEAR	CL.
CLG	CLG
CLOSE	CLO.
CLS	CLS
COLOR	C.
COLOUR	C.
COS	COS
COUNT	COU.
DATA	D.
DEF	DEF
DEG	DEG
DELETE	DEL.
DIM	DIM
DIV	DIV
DRAW	DR.
EDIT	EDIT
EDITO	ED.
ELLIPSE	ELL.

Command	Abbreviation
ELSE	EL.
END	END
ENDCASE	ENDC.
ENDIF	ENDIF
ENDPROC	E.
ENDWHILE	ENDW.
EOF	EOF
EOR	EOR
ERL	ERL
ERR	ERR
ERROR	ERR.
EVAL	EV.
EXP	EXP
EXT	EXT
FALSE	FA.
FILL	FI.
FN	FN
FOR	F.
GCOL	GC.
GET	GET
GET\$	GE.
GOSUB	GOS.
GOTO	G.
HELP	HE.
HIMEM	H.
IF	IF
INKEY	INKEY
INKEY\$	INK.
INPUT	I.
INSTR(INS.
INT	INT
LEFT\$(LE.
LEN	LEN
LET	LET
LINE	LINE
LIST	L.

Command	Abbreviation
LISTO	LISTO
LN	LN
LOAD	LO.
LOCAL	LOC.
LOG	LOG
LOMEM	LOM.
LVAR	LV.
MID\$(M.
MOD	MOD
MODE	MO.
MOUSE	MOU.
MOVE	MOVE
NEW	NEW
NEXT	N.
NOT	NOT
OF	OF
OFF	OFF
OLD	O.
ON	ON
OPENIN	OP.
OPENOUT	OPENO.
OPENUP	OPENUP
OR	OR
ORIGIN	OR.
OSCLI	OS.
OTHERWISE	OT.
PAGE	PA.
PI	PI
PLOT	PL.
POINT	POINT
POINT(PO.
POS	POS
PRINT	P.
PROC	PROC
PTR	PTR
QUIT	Q.

Command	Abbreviation
RAD	RAD
READ	READ
RECTANGLE	REC.
REM	REM
RENUMBER	REN.
REPEAT	REP.
REPORT	REPO.
RESTORE	RES.
RETURN	R.
RIGHT\$ (RI.
RND	RND
RUN	RUN
SAVE	SA.
SGN	SGN
SIN	SIN
SOUND	SO.
SPC	SPC
SQR	SQR
STEP	S.
STEREO	STER.
STOP	STOP
STR\$	STR\$
STRING\$ (STRI.
SUM	SUM
SWAP	SW.
SYS	SYS
TAB (TAB (
TAN	T.
TEMPO	TE.
THEN	TH.
TIME	TI.
TO	TO
TOP	TOP
TRACE	TR.
TRUE	TRUE
UNTIL	U.

Command	Abbreviation
USR	USR
VAL	VAL
VDU	V.
VOICES	VO.
VPOS	VP.
WAIT	WA.
WHEN	WHEN
WHILE	W.
WIDTH	WI.

APPENDIX B – BASIC ERRORS

Error number	Error message
0	Silly!
0	No room to do this renumber
0	Line numbers larger than 65279 would be generated by this renumber
0	No room
0	Stopped
0	Invalid LISTO option
0	Invalid EDITO option
0	Corruption of stack
0	Error control status not found on stack for RESTORE ERROR
0	Missing incore name
0	LIST/EDIT found line number reference
0	HELP has no information on this keyword
1	No such mnemonic
1	No such suffix on EQU
2	Bad immediate constant
2	Bad address offset
2	Bad shift
3	Bad register
4	Missing =
4	Missing in FOR statement
4	Mistake
5	Missing .
6	Type mismatch: number needed
6	Type mismatch: numeric variable needed
6	Type mismatch: string needed
6	Type mismatch: string variable needed
6	Type mismatch: array needed
6	Type mismatch between arrays
6	Can't assign to array of this size
6	Array type mismatch as parameter
6	Can't SWAP arrays of different types
7	Not in a function

Error number	Error message
8	Too low a value for \$<number>
9	Missing "
10	DIM() function needs an array
10	No room for this dimension
10	Impossible dimension
10	No end of dimension list)
10	Bad DIM statement
10	Can't DIM negative amount
10	No room for this DIM
10	Arrays cannot be redimensioned
12	Items can only be made local in a function or procedure
13	Not in a procedure
14	Reference array incorrect
14	Unknown array
14	Unknown array in DIM() function
14	Undimensioned array
15	Subscript out of range
15	Incorrect number of subscripts
16	Syntax error
17	Escape
18	Division by zero
19	String too long
20	Number too big
20	Number too big for arc Sine or arc Cosine
21	Negative root
22	Logarithm range
23	Accuracy lost in Sine/Cosine/Tangent
24	Exponent range
26	Unknown or missing variable
26	Can't use array reference here
27	Missing)
27	Missing]
27	Missing {

Error number	Error message
27	Missing }
28	Bad Hex
28	Hex number too large
28	Bad Binary
29	No such function/procedure
30	Bad call of function/procedure
31	Arguments of function/procedure incorrect
31	Invalid RETURN actual parameter
31	Invalid array actual parameter
32	Not in a FOR loop
33	Can't match FOR
34	Bad FOR control variable
35	The step cannot be zero
36	Missing TO
38	Not in a subroutine
39	ON syntax
40	ON range
41	No such line
42	Out of data
43	Not in a REPEAT loop
45	Missing #
46	Not in a WHILE loop
47	Missing ENDCASE
48	CASE statement must be the last thing on a line
49	Missing ENDIF
50	Bad MOUSE variable
51	Too many input expressions for SYS
51	Too many output variables for SYS

APPENDIX C - CHARACTER CODES

LATIN ALPHABET 1

b. 0				0	0	0	0	0	0	0	0	1	1	1	1	1	1	1		
b. 0				0	0	0	0	1	1	1	1	0	0	0	0	1	1	1		
b. 0				0	0	1	1	0	0	1	1	0	0	1	1	0	0	1		
b. 0				1	0	1	0	1	0	1	0	1	0	1	0	1	0	1		
				00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	
b.	b.	b.	b.																	
0	0	0	0	00			SP	0	a	P	'	p			NBSP	°	À	Ð	à	ð
0	0	0	1	01			!	1	A	Q	a	q			ı	±	Á	Ñ	á	ñ
0	0	1	0	02			"	2	B	R	b	r			¢	²	Â	Ò	â	ò
0	0	1	1	03			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó
0	1	0	0	04			\$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô
0	1	0	1	05			%	5	E	U	e	u			¥	µ	Å	Ö	å	ö
0	1	1	0	06			&	6	F	V	f	v			¦	¶	Æ	Ø	æ	ø
0	1	1	1	07			'	7	G	W	g	w			§	·	Ç	×	ç	÷
1	0	0	0	08			(8	H	X	h	x			"	,	È	Ø	è	ø
1	0	0	1	09)	9	I	Y	i	y			©	¹	É	Ù	é	ù
1	0	1	0	10			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú
1	0	1	1	11			+	;	K	Ç	k	ç			«	»	Ë	Û	ë	û
1	1	0	0	12			,	<	L	\	l				¬	¼	Ì	Ü	ì	ü
1	1	0	1	13			-	=	M]	m	}			SHY	½	Í	Ý	í	ý
1	1	1	0	14			.	>	N	^	n	~			®	¾	Î	Ë	î	ë
1	1	1	1	15			/	?	O	_	o				™	¿	Ï	Ë	ï	ÿ

LATIN ALPHABET 2

b.	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
b.	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
b.	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
b.	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
b.	b.	b.	b.													
0	0	0	0	00												
0	0	0	1	01												
0	0	1	0	02												
0	0	1	1	03												
0	1	0	0	04												
0	1	0	1	05												
0	1	1	0	06												
0	1	1	1	07												
1	0	0	0	08												
1	0	0	1	09												
1	0	1	0	10												
1	0	1	1	11												
1	1	0	0	12												
1	1	0	1	13												
1	1	1	0	14												
1	1	1	1	15												

LATIN ALPHABET 3

				b.	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	
				b.	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	
				b.	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	
				b.	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	
				00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15		
b.	b.	b.	b.	00			SP	0	á	P	·	p			NBSP	°	À	⊗	à	⊗	
0	0	0	1	01			!	1	A	Q	a	q			Ɔ	h	Á	Ñ	á	ñ	
0	0	1	0	02			"	2	B	R	b	r			˘	²	Â	Ò	â	ò	
0	0	1	1	03			#	3	C	S	c	s			£	³	⊗	Ó	⊗	ó	
0	1	0	0	04			\$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô	
0	1	0	1	05			%	5	E	U	e	u			⊗	μ	Č	Ĝ	č	ĝ	
0	1	1	0	06			&	6	F	V	f	v			Ɔ	h	Č	Ö	č	ö	
0	1	1	1	07			'	7	G	W	g	w			Š	·	Ç	×	ç	÷	
1	0	0	0	08			(8	H	X	h	x			"	,	È	Ĝ	è	ĝ	
1	0	0	1	09)	9	I	Y	i	y			ı	ı	É	Ù	é	ù	
1	0	1	0	10			*	:	J	Z	j	z			Ş	ş	Ê	Ú	ê	ú	
1	0	1	1	11			+	;	K	Ł	k	ł			Ğ	ğ	Ë	Û	ë	û	
1	1	0	0	12			,	<	L	\	l				Ĵ	ĵ	Ì	Ü	ì	ü	
1	1	0	1	13			-	=	M	Ÿ	m	ÿ			SHY	½	Í	Û	í	ü	
1	1	1	0	14			.	>	N	ˆ	n	˜			⊗	⊗	Î	Ŝ	î	ŝ	
1	1	1	1	15			/	?	O	_	o						Ž	ž	İ	ß	

GREEK ALPHABET

				b.	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	
				b.	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
				b.	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
				b.	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
				00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	
b.	b.	b.	b.	00			SP	0	α	ρ	`	p			NBSP	°	ι	Π	υ	π
0	0	0	1	01			!	1	A	Q	a	q			'	±	A	P	α	ρ
0	0	1	0	02			"	2	B	R	b	r			,	2	B	⊗	β	s
0	0	1	1	03			#	3	C	S	c	s			£	3	Γ	Σ	γ	σ
0	1	0	0	04			\$	4	D	T	d	t			⊗	'	Δ	T	δ	τ
0	1	0	1	05			%	5	E	U	e	u			⊗	'	E	T	ε	υ
0	1	1	0	06			&	6	F	V	f	v				'A	Z	Φ	ζ	φ
0	1	1	1	07			'	7	G	W	g	w			§	·	H	X	η	χ
1	0	0	0	08			(8	H	X	h	x			"	'E	Θ	Ψ	θ	ψ
1	0	0	1	09)	9	I	Y	i	y			©	'H	I	Ω	ι	ω
1	0	1	0	10			*	:	J	Z	j	z			⊗	'I	K	İ	κ	ï
1	0	1	1	11			+	;	K	Ç	k	ç			«	»	Λ	Ï	λ	ü
1	1	0	0	12			/	<	L	\	l		→		-	'O	M	α	μ	ó
1	1	0	1	13			-	=	M]	m	}			SHY	½	N	é	ν	ù
1	1	1	0	14			.	>	N	ˆ	n	˜			⊗	'T	Z	η	ξ	ώ
1	1	1	1	15			/	?	0	-	o				-	'Ω	O	ι	o	⊗

BFONT CHARACTER CODES

	0	10	20	30	40	50	60	70	80	90	100
0	Nothing	Down	Default logical colours	Move text cursor to 00	(2	<	F	P	Z	d
1	Next to printer	Up	Disable VDU	Move text cursor)	3	=	G	Q	L	e
2	Start printer	Clear text	Select mode	█	*	4	>	H	R	\	f
3	Stop printer	Start of line	Reprogram characters	!	+	5	?	I	S	J	g
4	Separate cursors	Paged mode	Define graphics area	"	.	6	@	J	T	^	h
5	Join cursors	Scroll mode	Plot	#	-	7	A	K	U	_	i
6	Enable VDU	Clear graphics	Default text/graphics areas	\$.	8	B	L	V	E	j
7	Beep	Define text colour	Nothing	%	/	9	C	M	W	a	k
8	Back	Define graphics colour	Define text area	&	0	:	D	N	X	b	l
9	Forward	Define logical colours	Define graphics origin	'	1	;	E	O	Y	c	m

110	120	130	140	150	160	170	180	190	200	210	220	230	240	250
η	κ	Β	Α	Ω	°	Λ	Ζ	ν	Θ	Σ	±	Ζ	π	Ω
ο	υ	Ϣ	ε	ι	ι	Η	Ι	⊞	Ι	Τ	Π	η	ρ	≡
ρ	ζ	ε	ε	ε	—	—	⊞	⊞	κ	υ	θ	σ	ζ	
q	{	0	ε	δ	Γ	Η	Α	Λ	ε	δ	ι	τ	≠	
Γ	:	υ	α	δ	—	—	τ	Β	Μ	κ	ρ	κ	υ	ζ
ς	}	0	α	δ	Γ	Η	Α	Λ	ε	δ	ι	τ	≠	
t	~	+	α	δ	—	—	α	ε	υ	β	υ	κ		
υ	Back space and delete	+	α	δ	—	—	α	ε	υ	β	υ	κ		
υ	Α	↓	ε	α	'	'	υ	Ζ	Μ	Ι	δ	ε	ω	
ω	Α	↑	ο	ς	ι	'	'	Η	Ρ	±	ε	ο	δ	

APPENDIX D – TELETEXT CHARACTER CODES

TELETEXT ALPHANUMERIC CHARACTER CODES

	0	10	20	30	40	50	60	70	80	90	100	110	120
0	Nothing	Down	Nothing	Move cursor to 00									
1	Next to printer	Up	Disable VDU	Move cursor									
2	Start printer	Clear screen	Select mode										
3	Stop printer	Start of line	Reprogram characters										
4	Nothing	Paged mode	Nothing										
5	Nothing	Scroll mode	Nothing										
6	Enable VDU	Nothing	Nothing										
7	Beep	Nothing	Nothing										Back space and delete
8	Back	Nothing	Nothing										Nothing
9	Forward	Nothing	Nothing										Alpha red

	130	140	150	160	170	180	190	200	210	220	230	240	250
Alpha green	Normal * height	Graphic cyan											
Alpha yellow	Double height	Graphic white											
Alpha blue	Nothing	Conceal display											
Alpha magenta	Nothing	Contiguous graphics *											
Alpha cyan	Nothing	Separated graphics											
Alpha * white	Graphic red	Nothing											
Flash	Graphic green	Black * background											
Steady *	Graphic yellow	New background											
Nothing	Graphic blue	Hold graphics											
Nothing	Graphic magenta	Release * graphics											

* every line starts with these options selected .

TELETEXT GRAPHICS CHARACTER CODES

	0	10	20	30	40	50	60	70	80	90	100	110	120
0	Nothing	Down	Nothing	Move cursor to 00									
1	Next to printer	Up	Disable VDU	Move cursor									
2	Start printer	Clear screen	Select mode										
3	Stop printer	Start of line	Reprogram characters										
4	Nothing	Paged mode	Nothing										
5	Nothing	Scroll mode	Nothing										
6	Enable VDU	Nothing	Nothing										
7	Beep	Nothing	Nothing										Back space and delete
8	Back	Nothing	Nothing										Nothing
9	Forward	Nothing	Nothing										Alpha red

	13 ϕ	14 ϕ	15 ϕ	16 ϕ	17 ϕ	18 ϕ	19 ϕ	20 ϕ	21 ϕ	22 ϕ	23 ϕ	24 ϕ	25 ϕ
Alpha green	Normal * height	Graphic cyan											
Alpha yellow	Double height	Graphic white											
Alpha blue	Nothing	Conceal display											
Alpha magenta	Nothing	Contiguous graphics *											
Alpha cyan	Nothing	Separated graphics											
Alpha * white	Graphic red	Nothing											
Flash	Graphic green	Black * background											
Steady *	Graphic yellow	New background											
Nothing	Graphic blue	Hold graphics											
Nothing	Graphic magenta	Release graphics *											

* every line starts with these options

APPENDIX E – SCREEN MODES

Mode	Text col x row	Resolution hor x ver	Log.Cols	Memory used
0	80 x 32	640 x 256	2	20K
1	40 x 32	320 x 256	4	20K
2	20 x 32	160 x 256	16	40K
3	80 x 25	Text only	2	40K
4	40 x 32	320 x 256	2	20K
5	20 x 32	160 x 256	4	20K
6	40 x 25	Text only	2	20K
7	40 x 25	TELETEXT	16	80K
8	80 x 32	640 x 256	4	40K
9	40 x 32	320 x 256	16	40K
10	20 x 32	160 x 256	256*	80K
11	80 x 25	Text only	4	40K
12	80 x 32	640 x 256	16	80K
13	40 x 32	320 x 256	256*	80K
14	80 x 25	Text only	16	80K
15	80 x 32	640 x 256	256*	160K
16	132 x 32	Text only	16	132K
17	132 x 25	Text only	16	132K
18†	80 x 64	640 x 512	2	40K
19†	80 x 64	640 x 512	4	80K
20†	80 x 64	640 x 512	16	160K


















* In the 256 colour modes, there are some restrictions on the control of the colours. In particular, the COLOUR and GCOL commands can only be used to select from 64 base colours. The full 256 can be obtained via the TINT option. Also, the selection from the colour palette of 4096 shades is only possible in groups of 16.

† Modes 18 to 20 should only be used on multiple scan-rate monitors (for example, NEC Multisync). On ordinary monitors they do not produce a usable picture.

APPENDIX F – INKEY VALUES

Key	INKEY number
f0	-33
f1	-114
f2	-115
f3	-116
f4	-21
f5	-117
f6	-118
f7	-23
f8	-119
f9	-120
f10	-31
f11	-29
f12	-30
A	-66
B	-101
C	-83
D	-51
E	-35
F	-68
G	-84
H	-85
I	-38
J	-70
K	-71
L	-87
M	-102
N	-86
O	-55
P	-56
Q	-17
R	-52
S	-82
T	-36
U	-54
V	-100

Key	INKEY number
W	-34
X	-67
Y	-69
Z	-98
0	-40
1	-49
2	-50
3	-18
4	-19
5	-20
6	-53
7	-37
8	-22
9	-39
,	-103
-	-24
.	-104
/	-105
[-57
\	-121
]	-89
;	-88
Escape	-113
Tab	-97
Caps Lock	-65
Scroll Lock	-32
Num Lock	-78
Break	-45
Back Tick/~	-46
£/Currency	-47
Back Space	-48
Insert	-62
Home	-63
Page Up	-64
Page Down	-79

Key	INKEY number
'/"	-80
 (either/both)	-1
 (either/both)	-2
 (either/both)	-3
 (left-hand)	-4
 (left-hand)	-5
 (left-hand)	-6
 (right-hand)	-7
 (right-hand)	-8
 (right-hand)	-9
Space Bar	-99
	-90
	-74
	-106
	-58
	-26
	-122
	-42
Keypad 0	-107
Keypad 1	-108
Keypad 2	-125
Keypad 3	-109
Keypad 4	-123
Keypad 5	-124
Keypad 6	-27
Keypad 7	-28
Keypad 8	-43
Keypad 9	-44
Keypad +	-59
Keypad -	-60
Keypad .	-77
Keypad /	-75
Keypad #	-91
Keypad *	-92
Keypad 	-61

Key	INKEY number
Select	-10
Menu	-11
Adjust	-12

APPENDIX G – PLOT CODES

The groups of PLOT codes are as follows:

0 – 7	(&00 – &07)	Solid line including both end points
8 – 15	(&08 – &0F)	Solid line excluding final points
16 – 23	(&10 – &17)	Dotted line including both end points
24 – 31	(&18 – &1F)	Dotted line excluding final points
32 – 39	(&20 – &27)	Solid line excluding initial point
40 – 47	(&28 – &2F)	Solid line excluding both end points
48 – 55	(&30 – &37)	Dotted line excluding initial point
56 – 63	(&38 – &3F)	Dotted line excluding both end points
64 – 71	(&40 – &47)	Point plot
72 – 79	(&48 – &4F)	Horizontal line fill (left & right) to non-background
80 – 87	(&50 – &57)	Triangle fill
88 – 95	(&58 – &5F)	Horizontal line fill (right only) to background
96 – 103	(&60 – &67)	Rectangle fill
104 – 111	(&68 – &6F)	Horizontal line fill (left & right) to foreground
112 – 119	(&70 – &77)	Parallelogram fill
120 – 127	(&78 – &7F)	Horizontal line fill (right only) to non-foreground
128 – 135	(&80 – &87)	Flood to background
136 – 143	(&88 – &8F)	Flood to foreground
144 – 151	(&90 – &97)	Circle outline
152 – 159	(&98 – &9F)	Circle fill
160 – 167	(&A0 – &A7)	Circular arc
168 – 175	(&A8 – &AF)	Segment
176 – 183	(&B0 – &B7)	Sector
184 – 191	(&B8 – &BF)	Block copy/move
192 – 199	(&C0 – &C7)	Ellipse outline
200 – 207	(&C8 – &CF)	Ellipse fill
208 – 215	(&D0 – &D7)	Graphics characters
216 – 223	(&D8 – &DF)	Reserved for Acorn expansion
224 – 231	(&E0 – &E7)	Reserved for Acorn expansion
232 – 239	(&E8 – &EF)	Sprite plot
240 – 247	(&F0 – &F7)	Reserved for user programs
248 – 255	(&F8 – &FF)	Reserved for user programs

Within each block of eight the offset from the base number has the following meaning:

- 0 Move cursor relative (to last graphics point visited)
- 1 Draw relative using current foreground colour
- 2 Draw relative using logical inverse colour
- 3 Draw relative using current background colour
- 4 Move cursor absolute (ie move to actual co-ordinate given)
- 5 Draw absolute using current foreground colour
- 6 Draw absolute using logical inverse colour
- 7 Draw absolute using current background colour

The above applies except for COPY and MOVE where the codes are as follows:

- 184 (&B8) Move only, relative
- 185 (&B9) Move rectangle relative
- 186 (&BA) Copy rectangle relative
- 187 (&BB) Copy rectangle relative
- 188 (&BC) Move only, absolute
- 189 (&BD) Move rectangle absolute
- 190 (&BE) Copy rectangle absolute
- 191 (&BF) Copy rectangle absolute

APPENDIX H – VDU COMMANDS

VDU Code	Ctrl	Extra bytes	Meaning
0	@	0	Does nothing
1	A	1	Sends next character to printer only
2	B	0	Enables printer
3	C	0	Disables printer
4	D	0	Writes text at text cursor
5	E	0	Writes text at graphics cursor
6	F	0	Enables VDU driver
7	G	0	Generates bell sound
8	H	0	Moves cursor back one character
9	I	0	Moves cursor on one space
10	J	0	Moves cursor down one line
11	K	0	Moves cursor up one line
12	L	0	Clears text area
13	M	0	Moves cursor to start of current line
14	N	0	Turns on page mode
15	O	0	Turns off page mode
16	P	0	Clears graphics area
17	Q	1	Defines text colour
18	R	2	Defines graphics colour
19	S	5	Defines logical colour
20	T	0	Restores default logical colours
21	U	0	Disables VDU drivers or deletes current line
22	V	1	Selects screen mode
23	W	9	Multi-purpose command
24	X	8	Defines graphics window
25	Y	5	PLOT
26	Z	0	Restores default windows
27	[0	Does nothing
28	\	4	Defines text window
29]	4	Defines graphics origin
30	^	0	Homes text cursor
31	_	2	Moves text cursor

APPENDIX I – OPERATING SYSTEM COMMANDS

Command	Meaning
*AUDIO	Enables/disables the sound system
*CHANNELVOICE	Assigns a voice to a channel
*CONFIGURE	Defines the CMOS RAM configurations
*ECHO	Reflects a string to the screen
*FX	Accesses a particular operating system routine
*GO	Starts execution at a given address
*GOS	Enters the Arthur supervisor
*HELP	Provides help information about commands
*IF	Conditionally executes a command
*IGNORE	Sets the printer ignore character
*KEY	Assigns a string to a soft key
*POINTER	Enables/disables the mouse pointer
*QSOUND	Queues a sound for later processing
*SET	Assigns a string to a variable
*SETEVAL	Assigns a value to a variable
*SETMACRO	Assigns an expression to a variable
*SHADOW	Enables/disables automatic shadow mode
*SHOW	Lists all variables defined
*SOUND	Makes a sound
*SPEAKER	Enables/disables the internal loudspeaker
*STATUS	Displays default values held in CMOS RAM
*STEREO	Sets the stereo position of a channel
*TEMPO	Sets the sound system tempo
*TIME	Prints the day, date and time
*TUNING	Sets the overall sound system tuning
*TV	Gives the vertical screen alignment and interlace opts
*UNSET	Deletes variables set by *SET, etc
*VOICES	Displays the available voices and channel map
*VOLUME	Sets the overall sound system volume

APPENDIX J – ADFS COMMANDS

Command	Description
*ACCESS	Sets the file attributes
*ADFS	Selects the Advanced Disc Filing System
*APPEND	Appends subsequent keyboard input to a file
*BACK	Makes the previously selected directory current
*BACKUP	Makes an exact copy of all information from one disc to another
*BUILD	Opens a new file and appends keyboard input to it
*BYE	Ends an ADFS session and 'parks' the Winchester disc
*CAT	Displays a directory catalogue
*CDIR	Creates a new directory
*CLOSE	Closes all open files in the current filing system
*COMPACT	Reorganises the free space on a disc
*COPY	Copies one or more files from one directory to another
*CREATE	Reserves space for a file
*DELETE	Deletes a file from the current file catalogue
*DIR	Changes the current directory
*DISMOUNT	Closes all currently open files on a drive
*DRIVE	Sets current drive to the drive number specified
*DUMP	Displays a HEX and ASCII dump of a file
*ENUMDIR	Sends a list of filenames to an output file
*EX	Displays file catalogue information for a directory
*EXEC	Takes subsequent input from a file
*FORMAT	Formats a floppy disc for use with the ADFS
*FREE	Displays the amount of free space on a drive
*INFO	Displays file catalogue information for one or more files
*LCAT	Displays the catalogue for the library directory
*LEX	Displays file catalogue information for the library directory
*LIB	Sets the current library directory
*LIST	Displays the contents of a file with line numbers
*LOAD	Loads a file into memory
*MAP	Shows the distribution of free space on a drive
*MOUNT	Initialises an ADFS disc
*MOVE	Moves a file between filing systems
*OPT	Sets up filing system options
*PRINT	Displays the content of a file in ASCII format
*REMOVE	Deletes a file which need not exist

Command	Description
*RENAME	Changes the name and/or directory of a file
*RUN	Loads and executes a file
*SAVE	Saves a block of memory as a file
*SETTYPE	Sets the type of a file
*SHUT	Closes all open files in all filing systems
*SHUTDOWN	Closes down all files and file servers
*SPOOL	Creates a file and sends all subsequent VDU output to it
*SPOOLON	Appends all subsequent VDU output to an existing file
*STAMP	Date-stamps a file; sets type to &FFD if untyped
*TITLE	Alters the title string of a directory
*TYPE	Displays the content of a file, without line numbers
*UP	Moves n levels up the directory structure
*WIPE	Deletes a wildcard file selection

APPENDIX K – *FX COMMANDS

The following is a summary. A complete list is included in the Reference guide.

Command	Description
*FX 0	Displays operating system title and version number
*FX 1	Writes to location left free for the user
*FX 2	Specifies stream for all subsequent data input
*FX 3	Specifies stream for all subsequent data output
*FX 4	Controls cursor key status
*FX 5	Selects where subsequent printer output will be sent
*FX 6	Selects printer ignore character
*FX 7	Selects RS423 baud rate for receiving data
*FX 8	Selects RS423 baud rate for transmitting data
*FX 9	Selects flash rate for first colour
*FX 10	Selects flash rate for second colour
*FX 11	Selects keyboard auto-repeat delay
*FX 12	Selects keyboard auto-repeat rate
*FX 15	Flushes buffer
*FX 18	Resets function keys
*FX 19	Waits for vertical sync (vsync)
*FX 20	Resets font definitions
*FX 21	Flushes a selected buffer
*FX 25	Resets a group of font definitions
*FX 106	Selects cursor / activates mouse
*FX 112	Writes screen bank number addressed by VDU driver
*FX 113	Writes screen bank number addressed by display hardware
*FX 114	Sets up automatic shadow mode
*FX 118	Reflects keyboard status in LEDs
*FX 120	Writes keys pressed information
*FX 124	Clears Escape condition
*FX 125	Sets Escape condition
*FX 126	Acknowledges Escape condition
*FX 138	Inserts character code into buffer
*FX 139	*OPT equivalent
*FX 143	Issues module service call
*FX 144	*TV equivalent
*FX 153	Inserts character into input buffer

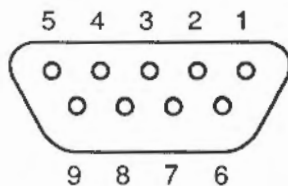
Command	Description
*FX 156	Sets RS423 attributes
*FX 162	Writes a value in CMOS RAM
*FX 163	Sets the dot-dash line pattern length
*FX 178	Enables/disables keyboard
*FX 181	Alters RS423 character actions
*FX 196	*FX 11 equivalent
*FX 197	*FX 12 equivalent
*FX 200	Selects Break and Escape effects
*FX 201	Sets keyboard status
*FX 202	Alters keyboard status byte
*FX 203	Sets RS423 'buffer full' limit
*FX 204	Enables/disables RS423 buffering
*FX 211	Selects bell channel number
*FX 212	Selects bell amplification
*FX 213	Selects bell frequency
*FX 214	Selects bell duration
*FX 216	Cancels function key expansion
*FX 217	Resets paged mode line count
*FX 218	Cancels VDU command sequence
*FX 219	Selects Tab key code
*FX 220	Selects Escape character
*FX 221	Selects interpretation of input values 192 to 207
*FX 222	Selects interpretation of input values 208 to 223
*FX 223	Selects interpretation of input values 224 to 239
*FX 224	Selects interpretation of input values 240 to 255
*FX 225	Selects soft key interpretation
*FX 226	Selects Shift plus the soft key interpretation
*FX 227	Selects Ctrl plus the soft key interpretation
*FX 228	Selects Shift Ctrl plus the soft key interpretation
*FX 229	Selects Escape key status
*FX 230	Selects Escape effects
*FX 238	Selects numeric keypad interpretation
*FX 247	Sets the Break key effects
*FX 254	Selects effect of Shift on numeric keypad
*FX 255	Selects Shift and Shift Break startup options

APPENDIX L – PIN CONNECTIONS

All plugs and sockets are viewed from the outside of the Archimedes case looking in.

VIDEO

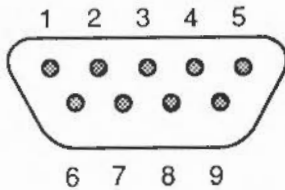
9-way D-type socket, SK2.



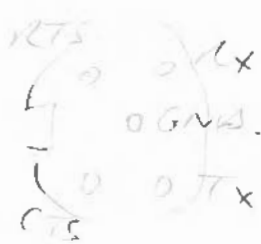
1	Red
2	Green
3	Blue
4	CSYNC
5	NC
6	0v
7	0v
8	0v
9	0v

SERIAL LINE

9-way D-type plug, PL1.

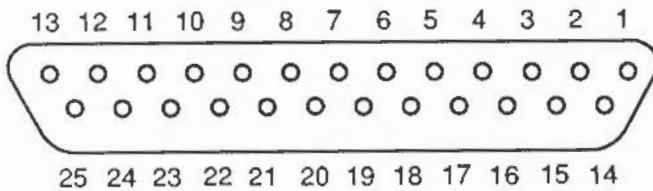


1	DCD
2	RxD
3	TxD
4	DTR
5	0v
6	DSR
7	RTS
8	CTS
9	RI



PRINTER

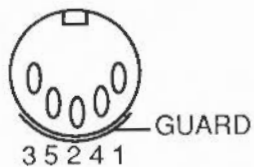
25-way D-type socket, SK3.



1	STB	13	NC
2	D0	14	NC
3	D1	15	NC
4	D2	16	NC
5	D3	17	0v
6	D4	18	0v
7	D5	19	0v
8	D6	20	0v
9	D7	21	0v
10	ACK	22	0v
11	BSY	23	0v
12	NC	24	0v
		25	0v

ECONET

5-way DIN socket, SK4.



1	Data
2	0v
3	Clock
4	Data
5	Clock

INDEX

- ABS 215
- *ACCESS 105
- ACS 216
- *ADFS 87
- ADFS commands 477
- ADVAL 216
- Advanced Disk Filing System 87
- after parameter 186
- amplitude 182
- AND 202, 217
- ANFS 87
- APPEND 218
- arc plotting 147
- arithmetic operators 26
- array operations 39
- array, size of 38
- arrays 36
- ASC 33, 219
- ASN 220
- assigning to variables 24
- ATN 220
- *AUDIO 383
- AUTO 14, 221
- auto-start options 107
- automatic numbering 14

- backing up discs 92
- bases 199
- BASIC
 - command mode 3
 - editor error messages 438
 - editor keys 434
 - errors 449
 - keywords 215
 - keywords, abbreviations 443
 - screen editor 419
- baud rate, printer 206

- BBC BASIC 1
- BBC Master 128 compatible patterns 154
- BEAT 185, 222
- BEATS 184, 222
- BFONT character codes 458
- BGET
- binary numbers 199
- BPUT
- *BUILD 112
- BY 225
- *BYE 87

- CALL 226
- CASE ... OF ... WHEN ...
OTHERWISE ... ENDCASE 69
- CASE 236
- *CAT 104
- CHAIN 237
- channel 182
- *CHANNELVOICE 383
- character codes 453
- characters
 - defining 51
 - representation of 33
- CHR\$ 34, 238
- CIRCLE 131, 238
- CIRCLE FILL 131
- circle plotting 145
- CLEAR 239
- CLG 239
- CLOSE
- CLS 241
- COLOUR (COLOR) 121, 241
- colour palette 122
- coloured text 175
- colours 120

- command files 112
- comments 16
- *COMPACT 108
- *CONFIGURE 113, 384
- conventions 1
- *COPY 102
- copying and moving graphics 158
- copying discs 92
- copying files 102
- COS 243
- COUNT 244
- CREATE 233
- cursor movements 50
- cursor, graphics 135

- DATA 55, 245
- data files 109
- debugging 209
- DEF 246
- defining characters 51
- DEG 247
- DELETE 12, 247
- deleting
 - files and directories 101
 - lines 11
 - programs 12
- DIM 36, 248
- directories 94
- directories, deleting 101
- disc directories 94
- disc options, configuring 113
- disc space 108
- discs
 - backing up 92
 - copying 92
 - formatting 90
- DIV 250

- dot-dash lines 138
- DRAW 251
- drawing lines 129
- *DRIVE 114
- drive numbers 89
- duration of sound 184

- *ECHO 390
- EDIT 252
- editing a program 8
- editing lines 10
- editor, BASIC 419
- ELLIPSE 252
- ELLIPSE FILL 132
- ellipse plotting 146
- ELSE 253
- END 254
- ENDCASE 255
- ENDIF 256
- ENDPROC 256
- ENDWHILE 257
- EOF
- EOR 202, 258
- ERL 259
- ERR 260
- ERROR 260
- error handling 209
- error message 3
- errors, BASIC 449
- errors, BASIC editor 438
- EVAL 34, 261
- *EX 106
- *EXEC 112
- EXP 262
- EXPR 234
- EXT
- FALSE 204, 264

- filename in program using REM 19
- files
 - copying and moving 102
 - deleting 101
- FILL 264
- flashing text 176
- floating-point numbers 22
- flood-fills 157
- FN 82, 265
- FOR ... NEXT 63
- FOR 266
- formatting a disc 90
- *FREE 108
- function keys 193
- functions 82
- *FX 390
- *FX commands 397, 479

- GCOL 121, 133, 267
- GCOL for pattern fills 151
- GET 54
- GET 269
- GET\$ 54, 270
- *GO 390
- *GOS 391
- GOSUB ... RETURN 72
- GOSUB 271
- GOTO 71, 273
- graphics 5, 129
 - graphics cursor 135
 - graphics resolution 119
 - graphics screen 129
 - graphics windows 163
- Greek alphabet codes 457

- halting listings 16
- HELP 215, 274

- *HELP 391
- hexadecimal numbers 199
- hierarchical directory structure 96
- HIMEM 274

- *I AM 88
- IF ... THEN ... ELSE ... ENDIF 61
- IF 275
- *IF 391
- IF...THEN ... ELSE 59
- *IGNORE 208, 391
- indirection operators 195
- *INFO 105
- INKEY 277
- INKEY values 467
- INKEY\$ 278
- INPUT 53, 279
- INPUT LINE 54, 280
- inputting data 53
- INSTALL 281
- INSTR 32, 282
- INT 283
- integer variables 25
- integers 22
- joining strings 29

- *KEY 193, 391
- key presses, reading 54
- keyboard 189
- keyboard scanning 190
- keys, in BASIC editor 434

- Latin alphabet codes 453
- LEFT\$ 30, 283
- LEN 32, 285
- LET 24, 286

libraries of procedures and functions
 83
 LIBRARY 83, 287
 LINE 129, 288
 LINE INPUT 288
 lines
 deleting 11
 editing 10
 LIST 15, 289
 listing programs 15
 listings, halting 16
 LISTO 290
 LN 291
 LOAD 19, 292
 loading a program 19
 LOCAL 76, 293
 LOCAL ERROR 294
 local error handling 210
 local variables 76
 LOG 295
 *LOGON 88
 LOMEM 295
 LVAR 296
 LVBLNK 232

 *MAP 108
 MATCH 234
 matrix multiplication 43
 memory, reserving 195
 MID\$ 30, 297
 MOD 298
 MODE 299
 modes 465
 modes, screen 117
 MOUSE 300
 MOUSE ON 192
 mouse 191

 MOVE 302
 moving files 102

 native mode patterns 153
 *NET 87
 NEW 303
 NEXT 304
 NOT 305
 numeric variables 22

 OF 306
 OFF 306
 OLD 308
 ON ... GOTO/GOSUB 72
 ON ... PROC 79
 ON 308
 OPENIN 311
 OPENOUT 312
 OPENUP 312
 operating guidelines iv
 operating system commands 383, 475
 operators
 in conditional expressions 59
 priority of 27
 *OPT 106
 OR 202, 313
 ORIGIN 314
 OSCLI 314, 383
 OTHERWISE 316

 PAGE 316
 palette, colour 122
 parallelogram plotting 141
 parameters in procedures 76
 parameters, value-result 79
 pathnames in ADFS 100
 pattern fills 149

- patterns
 - defining 151
 - giant 155
 - simple 156
- PI 317
- pin connections 481
- pitch 182
- PLOT 135, 318
- PLOT codes 471
- plotting
 - arcs 147
 - circles 145
 - ellipses 146
 - parallelograms 141
 - rectangles 140
 - sectors 148
 - segments 149
 - simple lines 138
 - sprites 171
 - triangles 139
- plug and socket connections 481
- plug connections iii
- POINT 318, 319
- *POINTER 392
- POS 320
- PRINT 45, 321
- print formatting 45
- printer ignore characters 207
- printers, connecting 205
- printing at the graphics cursor 159
- PROC 75, 325
- procedures, defining and calling 75
- procedures, recursive 80
- programs
 - deleting 12
 - editing 8
 - loading 19
 - renumbering 13
 - saving 18
- PTR
- *QSOUND 392
- QUIT 327
- RAD 327
- READ 55, 328
- READ 328
- RECTANGLE 130, 328
- RECTANGLE FILL 130
- rectangle plotting 140
- recursion 80
- REM 16, 330
- *RENAME 103
- RENUMBER 13, 331
- renumbering in BASIC editor 424
- renumbering programs 13
- REPEAT ... UNTIL 67
- REPEAT 332
- REPORT 332
- REPORT\$ 333
- RESTORE 333
- RESTORE ERROR 334
- RETURN 334
- RIGHT\$ 30, 335
- RND 337
- RUN 338
- SAVE 18, 338
- saving a program 18
- *SCHOOSE 171
- *SCOPY 170
- screen coordinate system 129
- screen modes 117, 465
- *SDELETE 170

searching and replacing, in BASIC
 editor 429
sector plotting 148
SEDIT 165
segment plotting 149
*SET 392
*SETEVAL 393
*SETMACRO 393
*SGET 173
SGN 339
*SHADOW 118, 393
shadow modes 118
*SHOW 393
SIN 340
*SLOAD 170
*SMERGE 171
SOUND 341
*SOUND 393
sound 181, 182
SPC 342
*SPEAKER 394
splitting strings 29
*SPOOL 113
sprite * commands 169
sprite editor 165
sprites 165
SQR 343
*SRENAME 169
*SSAVE 170
*STATUS 394
STEP 344
STEREO 181, 344
*STEREO 394
STOP 345
STOREA 232
STR\$ 34, 346
STRING 32
string variables 28
STRING\$ 347
strings
 copying 32
 joining 29
 length of 32
 replacing part of 31
 splitting 29
STSTORE 232
SUM 347
SWAP 348
SYS 349

TAB 49, 351
TAN 351
Teletext character codes 461
teletext mode 175
TEMPO 185, 352
*TEMPO 394
text cursor 48
text windows 161
THEN 353
TIME 354
*TIME 394
TIME\$ 355
TINT 126, 356
TO 357
TOKENADDR 235
TOP 357
TRACE 212, 358
triangle plotting 139
TRUE 204, 359
*TUNING 395
*TV 395

*UNSET 396
UNTIL 359

USR 360

VAL 34, 361
value-result parameter passing 79
variable names 21
variables 21
variables, assigning to 24
VARIND 231
VDU 50, 361
VDU commands 369, 473
VOICES 181, 363
*VOICES 395
*VOLUME 395
VPOS 363

WAIT 364
WHEN 365
WHILE ... ENDWHILE 68
WHILE 366
WIDTH 367
wildcards, in BASIC editor 431
windows 161

! operator 197
\$ operator 198
? operator 195
@% in PRINT 47
| in function key definitions 194
| operator 197

Acorn 