ADDCOMM

INTRODUCTION

ADDCOMM is a new advanced in upgrading the BBC Microcomputer and Acorn Electron. The software held within the EPROM has been specially designed to simulate a SUPER EXTENDED BASIC.

This new advance eliminates the untdy "star" commands necessary with other upgrades and its exceptional versatility allows the use of any real number, variable or expression within the statements.

ADDCOMM's commands have been selected to give as great a value for money as possible, being so versatile they negate the need to buy several separate chips and offer a cross-section of adaptable statements that have not been provided in BBC BASIC.

Software written by:

Richard P. D. Mallett

to whom any queries regarding the use of the ADDCOMM ROM may be addressed:

c/o Vine Micros
Marshborough,
Nr. Sandwich,
Kent. CT13 0PQ
(Tel. 0304 812276)

CONTENTS

IMPORTANT

ADDCOMM is now suitable for use on the Acorn Electron with ROM board.

Since being first released, ADDCOMM has been updated as follows:

A new FX call has been incorporated to allow ADDCOMM to be switched on and off more reliably than before.

*FX 163,0 (or just *FX 163) followed by BREAK will turn ADDCOMM on.

*FX 163,128 (or more simply the statement ADDCOMM) will turn ADDCOMM off. Note that neither a BREAK or CTRL-BREAK will turn it back on again.

*FX 163,64 will allow user definition of the memory workspace used by ADDCOMM. In use this will be 256 bytes (one 'page' of memory). Simply follow this FX call by:=

```
?(&Df0+chip)=page
```

where 'chip' is the chip number of ADDCOMM 0 to 15 (i.e. its situation on a ROM board or similar), and 'page' is the page number to be used. Then press BREAK and ADDCOMM will relocate itself. For example, to make ADDCOMM use the RS423 buffer (from &900 to &9FF) use a page number of 9 (not &900).

These facilities are *only* available on versions 1.30 and onwards.

Version 1.30 is the 'on' version (i.e. available from switch-on).

Version 1.31 is the 'off' version (i.e. it must be enabled using *FX 163,0 and then press BREAK).

A re-programming service is available to change an 'on' version to an 'off' version or vice-versa, at a charge of £1.00 (inclusive). The chip to be reprogrammed must be sent to Vine Micros, Marshborough, Nr. Sandwich, Kent CT13 0PG.

1. ENHANCED GRAPHICS

ADDCOMM greatly enhances the BBC Microcomputer's graphics ability, with the addition of such routines as circle and ellipse drawing and filling, turtle-like graphics, a recursive fill routine with the option of shading, polygon and arc drawind, rotating about any point and any angle, and perhaps most useful of all a user-definable scaled screen, allowing the user to create his or her own co-ordinate system depending on the screen.

Try the following:

```
MODE 4
SCALE 0,100,0,100
```

The screen is now scaled such that there are 100 screen units in the horizontal direction, and 100 in the vertical direction. The SCALE statement has four parameters, (try *HELP SCALE for a quick explanation), which tell the computer the exact specifications of the screen. The first is the value of the z-position that refers to the far left border of the screen, and the second refers to the far right. Thus in the example above the far left border has x-position 0, and the far right border has x-position 100. The third and fourth values refer to the y-position of the bottom of the screen, and the y-position of the top of the screen respectively.

To illustrate this using the SMOVE and SDRAW statements (which operate in a similar way to the standard MOVE and DRAW ones), try the following directly after typing the above:

```
SMOVE 0,0
SDRAW 100,100
```

The computer will now drag a line from the bottom left of the screen to the top right of the screen. To see why look at the statements you have entered. The SMOVE statement moves the graphics cursor to the point (0,0) on the screen. Note that this is according to the scale set up using the SCALE statement, so with the x-value equal to zero, and the y-value equal to zero, the point is the bottom left of the screen - which is one end of the line. The SDRAW statement draws a line from the previous point to the point (100,100), which, because of the scale in use, corresponds to the top right of the screen.

To clarify this enter the following:

```
MOVE 4
SCALE -100,200,-100,200
SMOVE 0,0
SDRAW 100,100
```

Note that although the SMOVE and SDRAW statements are the same, the line drawn is now much shorter, because the scale in use is much larger.

Try different scale values, but note than an illegal scale will be rejected with the error message 'Accuracy lost'.

To progress further it should be noted that any value can be used in the graphic statements, i.e. integer, floating point, negative integer, negative floating point, etc. So the following is allowed:

```
MODE 4
SCALE -1,1,-1,1
SMOVE 0,0
SDRAW 0.5,-0.5
```

To realise the true potential of a user-definable scale system, try the following version of a program to draw a sine curve on the screen:

```
10 MODE 4
20 SCALE -P1,PI,-1,1
30 FOR angle=-PI TO PI STEP 0.1
40 Y=COS angle
50 IF angel=-PI THEN SMOVE angle,Y ELSE SDRAW
angle,Y
60 NEXT angle
```

To do the same thing using the normal MOVE and DRAW statements would result in a program full of multiplication and addition equations, which are completely unnecessary using the scaled graphics statements provided by ADDCOMM.

In the same way that SMOVE and SDRAW replace MOVE and DRAW, SPLOT can be used instead of PLOT. For example:

```
MODE 4
SCALE 0,100,0,100
SMOVE 0,0
SMOVE 0,100
SPLOT 85,100,100
```

This will draw and fill a triangle in the usual PLOT 85 fashion.

To move on, a circle drawing statement is provided in ADDCOMM:

```
(MODE 4 : SCALE 0,100,0,100)
CIRCLE 50,50,20
```

(It should be noted that the MODE and SCALE statements do not need to be repeated before every graphic statement, but are simply included here in brackets to show what scale they use, so that the system is set up correctly.)

The above routine will draw a circle with its centre in the centre of the screen, i.e. at the point (50,50).

Type *HELP CIRCLE and you will see that the CIRCLE statement has a few more tricks up its sleeve, namely the ability to draw arcs, dotted circles and arcs, solid circles (i.e. circles filled with colour), and even sectors ('slices' of solid circles). To access these abilities the user needs to add four more parameters to the statement, which are not explained in this section, but are in the CIRCLE statement description.

Ellipses can be drawn using the ELLIPSE statement (What else?):

```
(MODE 4 : SCALE 0,100,0,100)
ELLIPSE 50,50,20,40
```

An ellipse is drawn with its vertical height twice that of its horizontal height, since the x-radius is 20 units and the y-radius is twice that at 40 units. Try different values for the two radii.

In the same way that CIRCLE can have extra parameters added to it so can ELLIPSE - see its explanation.

The next two statements are used together, being CFILL and FILL.

FILL will fill from the specified point until it meets a boundary, and CFILL defines what it fills with.

In modes 0 and 4, the colour filled with is the current foreground colour. In the other graphic modes the CFILL should be followed by either 4 or 8 colour codes, depending on what pattern/shading is required. The colour codes refer to a grid which is drawn on the screen instead of a set of points of the same colour.

By setting the colours within the grid to various values, shades of colours not normally possible on the BBC Microcomputer can be obtained. For instance, the following will produce a circle filled with an orangey colour:

```
10 MODE 1
20 SCALE -1,1,-1,1
30 CIRCLE 0,0,0.7
40 CFILL 1,2,2,1
50 FILL 0,0
```

The orange is obtained by mixing together red and yellow, which are the colours in the CFILL statement. Try CFIL 1,3,3,1 in the above routine for a pink colour, or CFILL 1,2,1,2 for a different version of the same orange colour as before. To fill in a solid colour simply set all the CFILL values to the same value, e.g. CFILL 1,1,1,1 will fill in colour 1 (which is red in Mode 1). The other 4 CFILL parameters extend the grid to a 2 by 4 one, allowing further more complicated combinations.

Note that the FILL routine is an extremely useful routine to have, and can handle screens of extraordinary complexity. For an example of its ability, go into a graphic mode, LIST a program, and then FILL around it.

Very complicated pictures may give the 'No room' error, which is because the FILL routine uses a recursive algorithm, and uses up spare memory while filling in. The amount of memory used depends upon the complexity of the screen picture, and thus some may not be finished. Always allow for this extra memory as it is not possible to restart the FILL routine once it has stopped.

The next two commands do not have any immediate effect, but nevertheless allow greater flexibility with the scaled screen, and are ROTATE and TRANS (translate).

ROTATE will rotate all subsequent scaled graphics (i.e. ALL ADDCOMM routines, but not the standard MOVE, DRAW and PLOT), above a specified point and through a specified angle (in degrees for ease of use).

To illustrate this:

```
10 MODE 4
20 SCALE -10,10,-10,10
30 FOR angle=0 TO 90 STEP 10
40 ROTATE 0,0,angle
50 ELLIPSE 0,0,3,6
60 NEXT angle
```

The program will draw 10 ellipses, one at 0 degrees to the horizontal, one at 10 degrees to the horizontal, one at 20 degrees, etc. up to 90 degrees to the horizontal. This is because we are defining (in line 40) the axis of rotation as (0,0) - which is the centre of the screen and the centre of the ellipses - and the degree of rotation as increasing by 10 degrees from 0 to 90. Note that all angles in ADDCOMM are presumed to be in degrees, and are measured anticlockwise from the 3 o'clock position. This, to many, is a strange system, but is in fact accepted as the standard method of measuring angles in mathematics.

TRANS again only affects subsequent scaled graphic output, and translates the screen by a specified amount. The main use is in moving a picture without having to change all parameters in the statements SMOVE, SDRAW, CIRCLE, etc. The statement is followed by two values which give the vector shift of the scaled screen.

```
10 MODE 4
20 SCALE 0,10,0,10
30 ROTATE 0,0,0 : REM reset so no rotating
40 FOR trans=0 TO 9
50 TRANS trans,trans
60 CIRCLE 1,1,1
70 NEXT trans
```

Each subsequent circle is translated by (1,1) from the previous circle, since the TRANS parameters are increasing by 1 each time. Note that if both ROTATE and TRANS are in use, then graphic output is first rotated and then translated.

The next and final graphic statement in this section is mainly for use with the POINT(x,y) function in the standard BASIC.

UNSCALE will convert scaled co-ordinates to non-scaled co-ordinates, to provide greater compatibility between the standard graphic scale and the user-defined one. To use it, simply follow the statement with the scaled co-ordinates and follow them with the two variables that you require the result to be in:

```
SCALE 0,100,0,100
UNSCALE 50,50,X,Y
PRINT X,Y
```

This will display the value of the unscaled co-ordinates corresponding to the scaled co-ordinates (50,50), which will be (if ROTATE and TRANS are not used) 639,511 (half of 1279 is 639.5, and half of 1023 is 511.5).

That concludes this section on enhanced graphics. For further details on the graphics statements, either use *HELP to access the syntax, or refer to the appropriate section of this manual.

2. LOGO/TURTLE GRAPHICS

Logo/Turtle graphics provide an interesting alternative to the normal cartesian system (x,y co-ordinates), and bear a slight relationship to polar co-ordinates, where the angle and distance of a point are given relative to the origin.

The Logo/Turtle graphic system (hereafter referred to as just Logo) is such that an imaginary cursor (usually called a turtle) is guided around the screen leaving a trail. Its direction is changed using TURN, and is moved in that direction using ADVANCE. Its initial position and direction is set using the LMOVE and ANGLE statements respectively, and the particular type of trail is governed by PEN. As an example, try the following short program:

```
 10 SCALE 0,10,0,10
 20 ROTATE 0,0,0
 30 TRANS 0,0
 40 MODE 4
 50 ANGLE 90
 60 LMOVE 5,5
 70 PEN 13
 80 FOR loop=1 TO 6
 90 ADVANCE 1
100 TURN 60
110 NEXT
```

Lines 10, 20 and 30 simply set up the screen scale (See 'Enhanced Graphics').

ANGLE 90 in line 50 will point the Logo cursor upwards (remember that angles are measured in degrees anticlockwise from the 3 o'clock position).

LMOVE 5,5 in line 60 will move the Logo cursor to the centre of the screen, since the scale is set for values from 0 to 10, and 5 is halfway between these two.

PEN 13 defines the plotting mode for Logo graphics as a line with the last point omitted. (See page 319 in the User Guide.)

The loop from lines 80 to 110 is repeated six times, and during each of these iterations the Logo cursor is advanced one unit, and turned by 60 degrees (anticlockwise). Thus a hexagon is produced.

By changing the TURN value and the iterations performed, other polygons can be created.

Other Logo-related statements are LCIRCLE and LELLIPSE.

LCIRCLE is exactly the same as CIRCLE except that the centre of the circle is the Logo-cursor position, and the radius is specified by the value after the LCIRCLE statement:

(Using the scale in the above program.)

```
LMOVE 5,5
LCIRCLE 3
```

LELLIPSE is, as one might expect, similar to ELLIPSE, but again the centre of the ellipse is now the Logo-cursor position, and the x-radius and y-radius are specified by the values following the LELLIPSE statement:-

```
LMOVE 5,5
LELLIPSE 4,2
```

The full range of graphics that can be created by using Logo is beyond the scope of this manual, but various weird and wonderful results can be created using a mixture of the graphics statements available with ADDCOMM.

Don't forget that Logo graphics *are* affected by SCALE, ROTATE and TRANS in exactly the same manner as the other graphics statements.

3. TOOLKIT COMMANDS

Toolkit commands are those that help not with the running of programs, but with the actual writing of them, and thus should not be used within a program.

The first one we shall look at is called MEM, and simply displays a breakdown of the memory available for the BASIC program. For instance, type in 'MEM'. (Note that the ' mark should not be included.)

```
MEM
```

The display will be something like:

```
Program :0002     2
Reserved:0000     0
Spare   :6CFE 27902
```

This indicates that 2 bytes are used up for a program, none have been reserved (i.e. there are no variables in memory, and no memory has been reserved using the DIM x statement), and 27902 bytes are spare for any type of use. Note that MEM gives the values in hexadecimal first, and then decimal.

FKEYS will display the contents of the function keys, so try defining them with a string of characters (e.g. *KEY3 CLS|M) and then type 'FKEYS'. ADDCOMM will then display the function key contents in the same way that they were entered. i.e. the function key 3 (if defined as above) will be printed as:

```
*KEY 3 CLS|M
```

As many will know, defining characters on the BBC Microcomputer is made rather complicated by the fact that each character needs to be first drawn on an eight by eight grid, and then have the code for each row worked out. Using a routine within the chip called via the 'CHAR' command this process is greatly simplified.

To illustrate how easy it is to use, type 'CHAR 224'. The screen will clear and an eight by eight array of dots will be printed, with eight numbers (in decimal) down the righthand side. These numbers are the numbers that refer to the value for each row of the character to be used in the VDU 23 statement. You will also notice that the cursor is at the top left of the array. Press the 'COPY' key. You will see a white block appear above the cursor, and you will probably have guessed that this corresponds to a lit pixel in the character 224, which is the one which you are defining.

If you are in Mode 7 then you will also see the '-' character below the array. Otherwise, if you are in any other mode, the character that you are defining will be displayed actual-size. You will also notice that one of the numbers changed when you pressed the 'COPY' key. This is simply because the values to be used in the VDU 23 statement have been updated. Pressing the 'DELETE' key will clear the pixel that the cursor is positioned under, and will again change the values for the VDU 23 statement.

To move the cursor around to set and reset various pixels is (as you may have guessed) done using the cursor keys. To exit, press the 'ESCAPE' key.

A further point worth mentioning is that the character displayed on entering this routine is in fact the current definition, and is not necessarily blank.

Progressing further takes us to the KILLREM command, which, as its name implies, kills (or deletes) all REM statements in a program. To try it out simply enter a short program and type 'KILLREM' on its own. Re-list the program and you will see that any REM statements will have been deleted (as well as any preceding spaces or colons). At times it may be necessary to delete REMs from only part of the program, and this is done using 'KILLREM|' - see KILLREM's description, or type *HELP KILLREM for further details.

'Bad program' is a sight not appreciated after having worked on a program for a while, and 'GOODPROG' will in most circumstances cure this, and thus allow a program to be listed.

LVAR will list variables except the system integers (A%-Z%). Simply entering 'LVAR' will list *all* variables. Entering 'LVARS' (note no space between the 'R' and the 'S') will list variables whose name starts with 'S' onwords in roughly alphabetical order. Similarly, 'LVARSg' will list variables whose names begin with 'S' onwards to those beginning with 'g'. Note lower case follows upper case in the lists of variables. To list all variables beginning with, say the ltter 't' enter 'LVARtt'.

'VERIFY' will, as its name suggests, verify that a program stored using cassette or disc is exactly the same as the program in memory. Load a program, then try 'VERIFY "<filename>" '. If you verify the program that you have just loaded, then no error message will be given. If however they are not the same, then the address

of the error will be given, followed by a listing of the line that contains the error, followed by the 'Not verified' error message. Machine code programs can also be verified - see VERIFY's description.

Perhaps the most useful and one of the more complicated toolkit routines is the 'FIND' command, which is used to search a program for a set of characters. Load a program, then type 'FIND|PRINT'. Any line with the PRINT statement in it will be listed. Note that the "|" character is vital, as it allows statements followed it to be tokenised. Try the following:

```
FIND|A
FIND|A|"
FIND|A|100,1000
FIND|A|"100,1000
```

Assuming your program is an average-length BASIC program, then you may see that if you follow the FIND command with ' |" ' then the characters will be sought for within quotation marks. Also, following with |100,1000 will search lines 100 to 1000 inclusive. Following with "|100,1000 will search lines 100 to 1000 inclusive within quotation marks.

ADDCOMM provides two replacing commands, which are GREPL and SREPL, short for global replace and selective replace. Replacing commands are used to change, say one variable name to another, or one text word to another, or one of many uses that the user will soon find.

Both have the same format, i.e. first the search characters, then the replace characters, and then any special data - the latter being optional and detailed not here but in the SREPL and GREPL sections.

For example type in the following short example:

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

Then type in exactly:

```
GREPL|A$|name$
```

This will replace all sets of characters 'A$' with 'name$'. Note that the lines in which anything is changed are listen before changing them and the lines listed will of course be 10 and 20. Now list the program. Then type:

```
GREPL|name|A
```

List the program. You will be surprised to see that only the variable 'name$' has been changed back to 'A$'. Type in:

```
GREPL|A|name
GREPL|name|address|"
```

Then list the program. The first command simply restores the program with the variable 'name$'. The second command is identical to a previous one, except that there are the two characters - |" - which tell ADDCOMM only to change those sets of characters that are within quotation marks, e.g. in PRINT statements etc. As with FIND, line numbers can also be added onto the end of the command with the result that only sections of a program will be changed.

The above examples are for the GREPL command, although they would all work with the SREPL command (the format of the command being identical), with the difference that SREPL will ask the user whether the line that has been listed should be altered or not (simply key in either 'Y' or 'N').

The last toolkit command is intended to be used only when a program is completely debugged and finished. The command is COMPACT, and is an intelligent line compacter. COMPACT will add lines together whereever it can, making sure not to add, say something onto the end of an IF...THEN statement, nor to delete a line that is jumped to elsewhere in the program, etc. The idea of COMPACT is to provide a way of greatly decreasing the size of a program, without affecting its running. The maximum length of lines created plus start and end lines can also be added - see COMPACT's description.

4. MISCELLANEOUS STATEMENTS

Jumping out of FOR...NEXT loops, and REPEAT...UNTIL loops is not taken as good programming, and jumping out of subroutines is certainly not recommended, but ADDCOMM comes to the rescue with three statements, that delete the appropriate loop of subroutine that is being jumped out of, being:

```
POPFOR
POPGOS
POPREP
```

For instance, type in the following short program:

```
10 GOSUB 20
20 PRINT "ADDCOMM rules !"
30 GOTO 10
```

RUN the program, and of course, the program will stop with the error 'Too many GOSUBs'. Now add line 25:-

```
25 POPGOS
```

Re-RUN the program and you will see that it will not stop this time.

LLIST is simply a statement that lists a line of a program. But isn't this the same as LIST you might ask? The difference is that LIST cannot be used within a program, but LLIST can, and more usefully that LLIST can be used to split up lines into their component statements.

For example:

```
OPT 1,0
LLIST 10
```

This will list line 10 in the split-up mode.

Changing 'OPT 1,0' to 'OPT 1,1' will disable this option and list the line normally.

SETWIN and WIN are used to create multiple windows, where each window has its own cursor. For instance, type in:

```
MODE 1
SETWIN 1,0,31,19,0
SETWIN 2,20,31,39,0
WIN 1
CLS
PRINT SQR(2)
```

```
WIN 2
PRINT PI
WIN 1
```

SETWIN will define the windows (up to a maximum of 7), and WIN will select them. Note that the most important feature is that the position of the cursor is 'remembered' when a window is re-selected.

LGOTO provides a way of jumping to a line using its name: For instance:

```
10 loop
20 PRINT "Hi there!"
30 LGOTO loop
```

This program will of course print out 'Hi there!' ad infinitum, but it demonstrates how labels can make programs very much easier to write - since there are no line numbers to remember. Note that spaces are ignored completely in labels, so 'LGOTO abc' is exactly the same as 'LGOTO a b c'. Labelled lines (e.g. line 10 in the program above) must have the label at the beginning of the line, otherwise they will be ignored. Labels are not allowed in immediate mode.

OPT provides a simple way of controlling some of ADDCOMM's routines, for instance, whether the FILL routine should fill using the shade set by CFILL, or the current foreground colour. A list of options is given in OPT's description.

SORT is a useful utility used to sort a single dimension string array:

```
DIM A$(4)
A$(1)="SHEILA"
A$(2)="FRED"
A$(3)="JIM"
A$(4)="BOB"
SORT A$(1),A$(4)
PRINT A$(1),'A$(2)'A$(3)'A$(4)
```

Surprise, surprise! The following will be printed:

```
BOB
FRED
JIM
SHEILA
```

See SORT's description for extra details.

ADDCOMM (the command) is sued to switch off ADDCOMM, so that games and the like can be played.

Use of ADDCOMM's statements

Before the user starts to use ADDCOMM extensively he or she should take note of the following:

1. That keywords placed *directly* after ADDCOMM's statements will result in the 'No such variable' error, beause the keyword will not have been tokenised.

   e.g. ANGLEDEG1

2. That a variable array placed *directly* after one of ADDCOMM's statements will cause the 'Array' error, because BASIC tries to interpret ADDCOMM's statement as an array.

   e.g. SMOVEX(1),Y(1)

3. That a left-parenthesis (bracket) placed *directly* after one of ADDCOMM's statements will again cause the 'Array' error, because BASIC is trying to interpret the statement as an array.

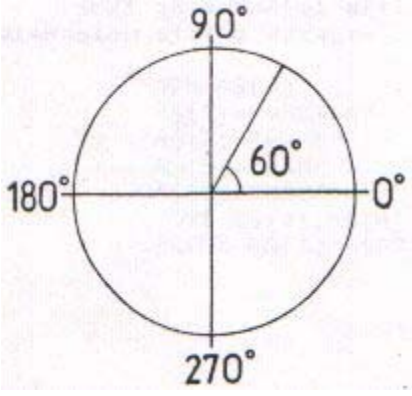
   e.g. ANGLE(PI+1)*2

All of these problems are cured simply by inserting a space *after* the statement word.

i.e. ANGLEDEG1 becomes ANGLE DEG1
SMOVEX(1),Y(1) becomes SMOVE X(1),Y(1)
ANGLE(PI+1)*2 becomes ANGLE (PI+1)*2

This will rarely cause any problems, as it is good programming practice to insert spaces anyway.

All angles on ADDCOMM are measured in degrees, and according to the diagram below:-



i.e. They are measured *anti-clockwise* from the 3 o'clock position.

The example in the diagram being 60 degrees.

This is not a bug in ADDCOMM, but is the standard mathematical way of measuring angles.

ADDCOMM

Description

Many programmers currently on the market use the full memory capacity of the BBC Microcomputer, especially games. It may therefore be necessary to regain the 256 bytes reserved by ADDCOMM in order for these programs to run correctly. The command 'ADDCOMM' will turn ADDCOMM off, so that it will not respond to any statement or command, and thus will not use any memory. The error message "Off !" (number 50) is generated by this command.

Syntax

ADDCOMM

Comment

It is advisable to press BREAK after using this command so that PAGE will be set correctly.

To turn ADDCOMM back on, use *FX 163 (See opposite page 1).

CTRL-BREAK will *not* turn it back on.

ADVANCE

Description

ADVANCE is used in LOGO graphics to move the logo cursor in the direction it is pointing. Depending on the value used in the PEN statement, ADVANCE can be made to leave a trail of dots (PEN 69); lines (PEN 13); dotted lines (PEN 29); and even solid triangles - see PEN. The resulting new position is relative to the start position.

Syntax

ADVANCE d

where: d is the distance according to the screen scale that the cursor turtle is to be advanced.

Comment

The value of 'd' can be negative, in which case the logo cursor will move backwards.

ANGLE

Description

ANGLE is used to set the direction of the logo cursor, and is normally used prior to use of the other logo statements.

Syntax

ANGLE n

where: n is the angle in degrees measured anticlockwise from the 3 o'clock position.

Comment

The ANGLE statement is absolute, i.e. that the previous direction of the logo cursor does not affect the new direction.

Example

ANGLE 90 will point the logo cursor upwards.
ANGLE 180 will point the logo cursor westwards.

CFILL

Description

CFILL is used to set up the colour grid to be used in the FILL statement. The grid can either be 2 by 2, or 2 by 4 (i.e. 2 pixels across and 4 pixels down). Each pixel can be any colour other than the current background colour.

Syntax

```
    CFILL c1,c2,c3,c4
or  CFILL c1,c2,c3,c4,c5,c6,c7,c8
```

where: c1 to c8 are the colours to be used, and none of which is the background colour (See diagram).

Comment

To fill solely using the current foreground colour, use OPT 6,1

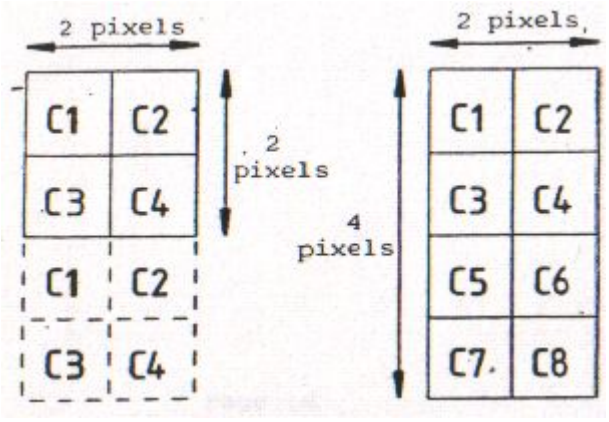
MODEs 0 and 4 have only one possible combination, being the current foreground colour, which is automatically selected during filling.

See FILL.

Example

```
CFILL 1,2,1,1
```

will (in a 4 colour mode) set up a shade of orange (red(1) + yellow(2) equals orange).



CHAR

Description

CHAR provides a simple way of designing user-definable characters. In use, an 8 by 8 grid will be displayed, with eight numbers down the righthand side - these numbers are the values to be used in the VDU 23 routine.

Controls

Cursor keys move the cursor around the grid.
DELETE key clears a dot.
COPY key sets a dot.
C key clears the whole grid.
ESCAPE exits the routine.

Syntax

CHAR c

where: c is character code to be defined.

Comment

A value of c less than 32 results in 'Syntax error'.

Character code 255 is used within this routine for the solid white block, so make sure that character 255 is user-definable if this command is used, e.g. don't set the definable characters to 32 to 63.

Example

```
CHAR 224
```

CIRCLE

Description

CIRCLE is a general purpose statement that allows a multitude of shapes to be drawn using a single statement, including: any regular polygons, circles, arcs, solid sectors, triangles, squares... all can be drawn utilising any of the standard plot codes available on the BBC Microcomputer.

Syntax

CIRCLE x,y,r (,sa,ea,ia,p)

where: x is the x-co-ordinate of the centre of the circle
y is the y co-ordinate of the centre of the circle
r is the radius of the circle
sa is the start angle for drawing the circle
ea is the end angle for drawing the circle
ia is the increment angle for drawing the circle
p is the plot code to be used

Comment

Note that the last four parameters are optional, but if one is included then they all must be. If they are omitted, they have default values of sa=0, ea=30, ia=10, p=13.

The start angle defines the angle of rotation at which drawing starts, and the end angle defines the angle of rotation at which drawing ends. The increment angle defines the angle change per section of the circle, and the plot code defines which plot routine is used to draw the circle. (See page 319 in the BBC User Guide.)

Examples

```
SCALE 0,10,0,10
CIRCLE 5,5,3  draws a full circle
CIRCLE 5,5,2,0,180,10,13  draws a half-circle
CIRCLE 5,5,2,90,360,10,13  draws a three-quarters
circle
CIRCLE 5,5,2,0,360,10,85  draws a solid circle
CIRCLE 5,5,2,0,360,60,85  draws a solid hexagon
```

COMPACT

Description

COMPACT is an intelligent line compacter that will add together lines wherever it can in a program, but in such a way that program execution is not affected. (A display of the line number currently being COMPACTed is given.)

Syntax

```
COMPACT (| s,e)

where:  s is a start line for compacting
        e is the end line for compacting
```

Comment

Note that the start and end line numbers are optional (the default being to compact the whole of the program).

The maximum length of line created is set using OPT 8,x where x is the appropriate value. On BREAK, this is set to 150.

Sometimes lines created may appear on listing to be longer than the number of characters defined above. This is because some words are tokenised so that they occupy only one byte of memory space, and not the 4 or 5 that they appear to on listing.

Note that ADDCOMM's statements and commands are *not* tokenised.

Example

```
COMPACT  compacts the whole program.
COMPACT|10,100  compacts lines 10 to 100 inclusive.
```

ELLIPSE

Description

ELLIPSE is a general purpose statement that allows a multitude of shapes to be drawn using a single statement, including: any regular polygons, circles, ellipses, arcs, solid sectors, triangles, squares, rectangles... All can be drawn utilising any of the standard plot codes available on the BBC Microcomputer.

Syntax

```
ELLIPSE x,y,rx,ry (,sa,ea,ia,p)
```

where:
- x is the x co-ordinate of the centre of the ellipse
- y is the y co-ordinate of the centre of the ellipse
- rx is the radius of the ellipse in the x direction
- ry is the radius of the ellipse in the y direction
- sa is the start angle for drawing the ellipse
- ea is the end angle for drawing the ellipse
- ia is the increment angle for drawing the ellipse
- p is the plot code to be used

Comment

Note that the last four parameters are optional, but if one is included then they all must be. If they are omitted, they have default values of sa=0, ea=360, ia=10 and p=13.

The start angle defines the angle of rotation at which drawing starts, and the end angle defines the angle of rotation at which drawing ends. The increment angle defines the angle change per section of the ellipse, and the plot code defines which plot routine is used to draw the ellipse. (See page 319 in the BBC User Guide).

Examples

```
SCALE 0,10,0,10
ELLIPSE 5,5,3,3     draws a full ellipse, which in
                    this case is a circle
ELLIPSE 5,5,2,4     draws a full ellipse, with height
                    twice that of its width
```

```
ELLIPSE 5,5,2,4,0,180,10,13
                    draws a half-ellipse
ELLIPSE 5,5,2,4,90,360,10,13
                    draws a three-quarters ellipse
ELLIPSE 5,5,2,4,0,360,10,85
                    draws a solid ellipse
ELLIPSE 5,5,2,4,0,360,60,85
                    draws a solid elongated hexagon
```

FILL

Description

FILL will fill from a defined point up to a boundary drawn on the screen

Syntax

```
FILL x,y
```

where: x is the x co-ordinate of the fill point
y is the y co-ordinate of the fill point

Comment

FILL uses a recursive algorithm, which means that spare memory space is used when filling in, thus for very complicated shapes, the error 'No room' may be given. Always allows for this memory usage, as the FILL routine cannot be restarted once it has stopped. In general, 7 bytes are used per one line drawn.

FILL can be used to produce shading on the screen if used in conjunction with CFILL. Note that if a colour defined in CFILL is recognised as the current background colour - which is not allowed - then the error message 'Bad colour' (code 49) is given.

If FILL is to be used solely with the current foreground colour, then use OPT6,1 to turn CFILL 'off'.

Example

```
MODE 1
SCALE 0,10,0,10
CFILL 1,2,2,1
FILL 5,5
```

FIND

Description

FIND is used to search all or part of a program for a specified set of characters. If found, the line is listed with the relevant characters highlighted.

Syntax

```
FIND|x(|(")s,e)

where:  x represents the search character(s)
        (") is an optional feature to search within
        quotes, otherwise not
        s is the start line for searching
        e is the end line for searching
```

Comment

A hash (#) within the search characters is used as a wild character (i.e. any character).

The optional " if missed out makes FIND search everywhere except within quotation marks, otherwise if included makes FIND search only within quotation marks.

Note that FIND displays the first match within a line and ignores the rest of that line.

Press ESCAPE to exit from FIND.

See LLIST for listing options.

Example

```
FIND|PRINT          will display all PRINT statements
FIND|A|"            will display all A's within
quotes
FIND|A|"10,100      will display all A's within
                    quotes from lines 10 to 100
                    inclusive
```

FKEYS

Description

FKEYS will display the contents of the function keys in such a manner as to ease the editing of them.

Syntax

```
FKEYS
```

Example

```
*KEY0 CLS|M
FKEYS
```

will give:

```
*KEY 0 CLS|M
```

GOODPROG

Description

GOODPROG will attempt to mend a 'bad' program, by correfting the 'index-pointers' within the lines.

Syntax

GOODPROG

Comments

If GOODPROG does not appear to work at first, type OLD, and then try.

Note that TOP, LOMEM, etc are *not* reset.

GREPL

Description

GREPL will replace globally any set of characters with any other set of characters within part or all of a program, with all replacing operations being listed and highlighted.

Syntax

```
GREPL|x|y(|(")s,e)
```

where: x represents the set of characters to be replaced,
y represents the set of characters which are replacing the 'x' character
(") is an optional feature (See below)
s is the start line
e is the end line

Comment

A hash (#) within the search characters is used as a wild character, (i.e. any character).

The optional " if missed out makes GREPL only replace outside of quotation marks. If included then replacing takes place solely within quotation marks.

Press ESCAPE to stop GREPL.

If a line becomes longer than 242 bytes then the 'No room' error is given.

Variables are cleared if a program becomes longer or shorter.

To stop GREPL listing lines, use OPT 5,1.

See LLIST for listing options.

Examples

```
GREPL|LET| will delete all LET statements
GREPL|name|address|"      will change the word name
                     to address inside quotation marks
GREPL|A$|number|10,100    will change the variable A%
                     to number from lines 10 to 100
                     inclusive
```

KILLREM

Description

KILLREM will delete REM statements within all or part of a program.

It will also delete preceding spaces and colons, and even the complete line if the REM statement is the only thing on it.

Syntax

```
KILL REM (|s,e)
```

where: s is the start line for deleting
e is the end line for deleting

Comment

Try not to write programs where a jump is made to a REM statement, as when this is deleted it will usually cause a 'No such line' error.

Variables are cleared if the program is changed.

Example

```
KILLREM                 kills all REM statements
KILLREM|10,100          kills REM statements from
                        lines 10 to 100 inclusive
```

LCIRCLE

Description

LCIRCLE draws a circle around the current logo cursor position, and can be made to draw polygons, arcs, sectors... etc as in the CIRCLE command.

Syntax

```
LCIRCLE r (,sa,ea,ia,p)
```

where: r is the radius of the circle/polygon
sa is the start angle for drawing the circle
ea is the end angle for drawing the circle
ia is the increment angle for drawing the circle
p is the plot code to be used

Comment

Note that the last four parameters are optional, but if one is included then they all must be. If they are omitted, they have default values of sa=0, ea=360, ia=10 and p=13.

Examples

```
(SCALE 0,10,0,10:LMOVE 5,5)
LCIRCLE 3
LCIRCLE 3,0,180,10,13
LCIRCLE 3,0,360,60,85
```

LELLIPSE

Description

LELLIPSE draws an ellipse around the current logo cursor position, and can also be made to draw polygons, arcs, sectors, etc...

Syntax

```
LELLIPSE rx,ry (,sa,ea,ia,p)
```

where:
- rx is the radius of the ellipse in the x direction
- ry is the radius of the ellipse in the y direction
- sa is the start angle for drawing the ellipse
- ea is the end angle for drawing the ellipse
- ia is the increment angle for drawing the ellipse
- p is the plot code to be used

Comment

Note that the last four parameters are optional, but if one is included then they all must be. If they are omitted, they have default values of sa=0, ea=360, ia=10 and p=13.

The start angle defines the angle of rotation at which drawing starts, and the end angle defines the angle of rotation at which drawing ends. The increment angle defines the angle change per section of the ellipse, and the plot code defines which plot routine is used to draw the ellipse. (See page 319 in the BBC User Guide.)

Examples

```
(SCALE 0,10,0,10:LMOVE 5,5)
LELLIPSE 3,3            draws a circle
LELLIPSE 4,2            draws an ellipse, with its
                        width twice that of its height
LELLIPSE 4,2,0,180,60,85  draws half a stretched
                        hexagon
```

LGOTO

Description

LGOTO provides a structured programming jump statement, (structured programmers never use GOTO!)

Syntax

LGOTO x

where: x represents the label

Comment

'x' like all labels must start with a lower case letter, and *cannot* have a colon within it

Any spaces within the label are ignored

Labels on their own are ignored, and if entered in immediate mode give the error message 'Label' (code 46).

A labelled line *must* have the label at the start of the line, although spaces are allowed in front of the label.

Example program

```
10 loop
20 PRINT "ADDCOMM rules!"
30 LGOTO loop
```

LLIST

Description

LLIST or 'line list' will list a single line within a program. So what, you may ask! But LLIST can be used within a program, can be programmed to split up lines into their separate statements, and with the use of a single FIND statement, all or part of a program can be listed using LLIST's options.

Syntax

```
LLIST x
```

where: x is the line number to be listed

Comment

FIND|#|s,e will list lines s to e inclusive, since # is a 'wild' character.

Various options control LLIST's features, which are detailed below. These options also control FIND, GREPL and SREPL's listing features.

Options

OPT 1,0	gives split listing
OPT 1,1	gives normal listing
OPT 3,x	prints x spaces after a line number
OPT 4,x	prints x spaces before a colon

Examples

```
(10 x=4:B=3:PRINT "HELLO")
```

LLIST 10	will list line 10 (split up)
OPT 2,1	will turn off highlighting, so:-
FIND |#|10,10	will also list line 10 (split up)
OPT 1,1	will cause normal listing, so:-
FIND|#|10,10	will also list line 10 in the same manner as LIST
FIND|#|10,100	would list lines 10 to 100 inclusive using the options set

LMOVE

Description

LMOVE sets up the logo cursor position, and is usually used prior to other logo statements.

Syntax

```
LMOVE x,y
```

where: x is the x co-ordinate of the new logo-cursor position
y is the y co-ordinate of the new logo-cursor position

Comment

The position of the logo-cursor can be found by using LPOS.

Examples

```
SCALE 0,2,0,2
LMOVE 1,1
```

would move the logo-cursor to the centre of the screen.

LPOS

Description

LPOS is used to return the current position of the logo-cursor.

Syntax

```
LPOS x,y
```

where: x is the *variable* into which the x co-ordinate is to be put
y is the *variable* into which the y co-ordinate is to be put

Comment

To work out the actual logo-cursor position (in the standard scale), use a combination of LPOS and UNSCALE.

e.g.

```
LPOS X,Y
UNSCALE X,Y,X,Y
```

Example

```
LMOVE 4,2
LPOS X,Y
PRINT X,Y
```

will yield:

```
   4    2
```

LVAR

Description

LVAR will list semi-alphabetically all or some of the variables currently in memory; the ones listed being user-definable.

Syntax

LVAR(s(e))

where: s is a *single character* defining the start variable for listing
e is a *single character* defining the end variable for listing

Comment

Anything in brackets is optional.

Do *not* insert spaces after LVAR if you are going to define start and end variables.

If s is omitted, then so must e be omitted.

If e is omitted, then it will be a default of 'z'.

£ *is* a variable!

Examples

```
(A=1, B=2, c=3, d=4, j2=5, z=6)
LVAR
LVARA
LVARc
LCARcj
LVARjj
```

MEM

Description

MEM gives a breakdown of memory available for BASIC program development, namely program length, reserved memory, and spare memory, first on hex and then decimal.

Syntax

```
MEM
```

Comment

MEM will not return sensible values after GOODPROG is used.

The 'reserved' value given includes all variables (except A% to Z%), array space, and machine code DIM space.

Example

```
MEM
```

might give:

```
Program  : 0D0B  3339
Reserved : 0000     0
Spare    : 51F5 20981
```

OPT

Description

OPT is used to allow user definition of some of ADDCOMM's features.

Syntax

OPT x,y

where: x is the option number
y is the new value of the option

Comment

There are 9 options (0 to 8), 8 of which require a value to be included.

Option list

```
OPT 0     Copyright message
OPT 1,0   Split listing in LLIST and FIND and REPLace
commands
OPT 1,1   Normal listing
OPT 2,0   Highlighting in FIND, SREPL and GREPL
OPT 2,1   No highlighting
OPT 3,x   x spaces after line number in LLIST etc
OPT 4,x   x spaces before colon in split listing
OPT 5,0   List lines in GREPL
OPT 5,1   Don't list lines in GREPL
OPT 6,0   CFILL on - use CFILL colours in FILL
OPT 6,1   CFILL off - use foreground colour in FILL
OPT 7,0   Move graphics cursor before plotting (in
ADVANCE)
OPT 7,1   Don't move graphics cursor - used with PEN
85
OPT 8,x   Maximum of x character per line in COMPACT
```

All have deault of 0 on BREAK, except OPT 4,4 and OPT 8,150.

Example

OPT 8,40 defines COMPACT's maximum line length to be 40 characters (or bytes)

PEN

Description

PEN is used to define the plotting routine to be used in the logo-graphics routines (including LCIRCLE and LELLIPSE).

Syntax

PEN p

where: p is the plotting code to be used (See page 319 in the User Guide)

Comment

PEN 85 (or similar) will not appear to work at first, because with ADVANCEing ADDCOMM will first move to the present logo-cursor position, and then draw the next one, thus the three co-ordinates needed with the plot-85 code will be corrupted. To remedy this use OPT 7,1, which prevents the first 'move' from occuring.

PEN also affects LCIRCLE and LELLIPSE, in the same way as the plot code option that can be used within these two statements.

Example

```
PEN 4     nothing seen on screen.
PEN 13    draw lines.
PEN 29    draw dotted lines.
PEN 85    draw triangles (used with OPT 7,1).
```

POPFOR

Description

POPFOR is used to delete the FOR...NEXT loop in present use from the stack, which allows such loops to be jumped out of cleanly.

Syntax

POPFOR

Example

```
10 FOR X=1 TO 30
20 IF X*X>100 THEN POPFOR:GOTO 50
30 NEXT X
40 END
50 ......
```

POPGOS

Description

POPGOS will delete the current subroutine return address, so that a GOSUB subroutine can be jumped out of 'cleanly'.

Syntax

POPGOS

Comment

```
10 GOSUB 20
20 POPGOS
30 GOTO 10
```

(N.B. A silly example)

POPREP

Description

POPREP is used to jump out of a REPEAT ... UNTIL loop cleanly, by deleting the current stack entry.

Syntax

```
POPREP
```

Example

```
10 REPEAT
20 POPREP
30 GOTO 10
```

(N.B. A very silly example.)

ROTATE

Description

ROTATE is used to rotate all subsequent graphic output from ADDCOMM by any angle and about any point. ROTATE affects *all* graphic statements within ADDCOMM.

Syntax

```
ROTATE x,y,a

where:  x is the x co-ordinate of the point about
        which rotation occurs
        y is the y co-ordinate of the point about
        which rotation occurs
        a is the angle of rotation in degrees
        measured anti-clockwise
```

Comment

ROTATE is set to its default value when a new SCALE is set up. i.e.:

```
ROTATE 0,0,0
```

Example

```
SCALE 0,10,0,10
ROTATE 5,5,45
ELLIPSE 5,5,4,2
```

SCALE

Description

SCALE is used to set-up the user definable scaling of the screen, thus providing a very flexible graphics system.

Syntax

```
SCALE x1,x2,y1,y2

where:  x1 is the minimum x co-ordinate that will
        appear on the screen
        x2 is the maximum x co-ordinate that will
        appear on the screen
        y1 is the minimum y co-ordinate that will
        appear on the screen
        y2 is the maximum y co-ordinate that will
        appear on the screen
```

Comment

If x1 = x2 or y1 = y2 then the error message 'Accuracy lost' is given.

A reflection of the screen can be produced by swapping the x1 and x2 values and/or y1 and y2 values.

SCALE automatically resets ROTATE and TRANS to their default values, i.e. ROTATE 0,0,0 and TRANS 0,0.

Changing the screen scale without altering co-ordinates can be used to simulate zooming in and out.

Example

```
SCALE 0,10,0,10
```

SDRAW

Description

SDRAW is similar to DRAW, but uses the scaled screen co-ordinate system.

Syntax

```
SDRAW x,y

where:  x is the x co-ordinate
        y is the y co-ordinate
```

Example

```
SCALE 0,10,0,10
SMOVE 0,0
SDRAW 10,10
```

SETWIN

Description

SETWIN is used to set the multiple winddows available with ADDCOMM. Up to seven such windows can be defined, and are selected using the WIN statement. Note that re-selected windows move the text cursor to the correct position.

Syntax

```
SETWIN w,x1,y1,x2,y2

where :- w is the window number (1 to 7)
        x1 is the 'x' co-ordinate of the bottom left
        of the window
        y1 is the 'y' co-ordinate of the bottom left
        of the window
        x2 is the 'x' co-ordinate of the top right of
        the window
        y2 is the 'y' co-ordinate of the top right of
        the window
```

Comment

x1,y1,x2,y2 are sent directly to the OS in that order, so this format is the same as the VDU 28 routine.

Example

```
SETWIN 1,0,20,10,5
```

SMOVE

Description

SMOVE is similar to MOVE, but uses the scaled screen co-ordinate system.

Syntax

```
SMOVE x,y

where:  x is the x co-ordinate
        y is the y co-ordinate
```

Example

```
SCALE 0,10,0,10
SMOVE 0,0
SDRAW 10,10
```

SORT

Description

SORT is used to sort all of part of a string array into alphabetical order.

Syntax

SORT s,e

where: s is the start string array for sorting
e is the end string array for sorting

Comment

As mentioned elsewhere, arrays can cause problems when used in ADDCOMM's statements, to cure these just insert a space between the word SORT and the start string array element.

Don't try to sort multi-dimension arrays. The results are unpredictable.

SORT only works with *strings*.

Example

SORT A$(1),A$(5) will sort into alphabetical order elements 1 to 5 of array A$.

SPLOT

Description

SPLOT is similar to PLOT, but uses the scaled screen co-ordinate system.

Syntax

```
SPLOT p,x,y
```

where: p is the plot code (page 319 in the BBC User Guide)
x is the x co-ordinate of plotting
y is the y co-ordinate of plotting

Example

```
SCALE 0,10,0,10
SPLOT 69,5,5
```

SREPL

Description

SREPL will replace selectively any set of characters with any other set of characters within part or all of a program, with all matches being listed and highlighted, and the user specifying whether to replace or not by keying either 'Y' or 'N'.

Syntax

SREPL|x|y(|(")s,e)

where: x represents the set of characters to be replaced
y represents the set of characters which are replacing the 'x' characters
(") is an option feature (See below)
s is the start line
e is the end line

Comment

A hash (#) within the search characters is used as a wild character (i.e. any character).

The optional " if missed out makes SREPL only replace outside of quotation marks. If included then replacing takes place solely within quotation marks.

Press ESCAPE to stop SREPL.

If a line becomes longer than 242 bytes then the 'No room' error is given.

Variables are cleared if a program becomes longer or shorter.

See LLIST for listing options.

Only 'Y' or 'y' will cause SREPL to replace. Anything else is ignored.

Examples

```
SREPL|LET|           will delete all LET statements
SREPL|name|address|" will change the word name to
                     address inside quotation marks
SREPL|A%|number|10,100    will change the variable A%
                     to number from lines 10 to 100
                     inclusive
```

TRANS

Description

TRANS is used to translate the scaled screen by a vector displacement.

Syntax

```
TRANS x,y

where:  x is the x-direction displacement
        y is the y-direction displacement
```

Comment

TRANS is automatically set to its default value when changing scale.

All graphic output from ADDCOMM is rotated and *then* translated.

Example

```
SCALE 0,10,0,10
TRANS 5,5
CIRCLE 0,0,3
```

will draw a circle in the centre of the screen.

TURN

Description

TURN is used to rotate the direction in which the logo-cursor is pointing.

Syntax

TURN a

where: a is the angle in degrees by which the logo-cursor is to be rotated

Comment

A positive value of 'a' rotates the logo-cursor anti-clockwise, a negative value clockwise.

Example

TURN 180 reverses the direction of the logo-cursor.
TURN -90 rotates clockwise by 90 degrees (quarter turn).

UNSCALE

Description

UNSCALE is used to return the screen position in the standard co-ordinate system (1280 by 1024) of a scaled point. This is very useful for use with the POINT function.

Syntax

```
UNSCALE x,y,a,b
```

where: x is the x co-ordinate
y is the y co-ordinate
a is a *variable* into which the unscaled x co-ordinate is to be placed
b is a *variable* into which the unscaled y co-ordinate is to be placed

Example

```
SCALE 0,10,0,10
UNSCALE 5,5,X,Y
```

will make X=639 and Y=511, which could then be used in the POINT(X,Y) function.

VERIFY

Description

VERIFY is used to check that a BASIC or machine code program on tape or disc exactly matches that in memory. If not then the first error found is pinpointed by a memory address and (if a BASIC program) a listing of the line with the error highlighed if possible.

Syntax

```
VERIFY f(,a)

where:  f is a filename
        a is the start address of the program
        (default = PAGE if omitted)
```

Comment

An error produces the error message 'Not verified' (Code 48).

Any address displayed is first given in hexadecimal and then decimal.

An unfound filename causes the 'Channel' error message.

Example

```
VERIFY "PROGRAM"
VERIFY "M-CODE",&1234
        (where &1234 is the start address of the
code)
```

WIN

Description

WIN is used to switch between user defined windows (set using WIN)

Syntax

WIN w

where: w is the window number (1 to 7)

Comment

Don't select a window that hasn't been defined.

If you do, don't panic, but type MODE 7 (or similar).

WIN will store the current text cursor position, and move it to its old position in the re-selected window.

Example

WIN 1

will select window 1.

*HELP

*HELP on its own gives:
ADDCOMM 1.00
ADDCOMM
(:)<Statement>

*HELP ADDCOMM will give a list of the syntax of *all* of ADDCOMM's statements.

*HELP x where x is a statement name will give details of that statement's syntax.

*HELP x y .... gives details of statement x then statement y .... etc.

for example:-
*HELP SCALE

gives:-
ADDCOMM 1.00
SCALE <x-min>,<x-max>,<y-min>,<y-max>

Also:-
*HELP LMOVE TURN

gives:-
ADDCOMM 1.00
LMOVE <x-position>,<y-position>
TURN <angle>

Note:-Anything in brackets () is optional.

*HELP statements can also be abbreviated e.g. *HELP SCALE is equivalent to *HELP SC.

If one of ADDCOMM's *HELP statements is the same as another ROM's, then place a colon ":" before the statement,

e.g. *HELP :VERIFY

Compatibility and Memory Use

In use, ADDCOMM uses one page (256 bytes) of memory to store its data. This is its 'private' memory area - no shared workspace is used.

ADDCOMM solely uses zero page space from &B0 to &BF and various BASIC scratchspace when in use - none of &70 to &8F is used.

ADDCOMM uses &B0 - &BF sensibly and in such a way that Disc Filing Systems, etc, can use it as well, without crashing the system.

To turn ADDCOMM off so that no memory space is used, use the command "ADDCOMM" - See its description.

Do's and Don'ts

Do

Allow for memory used in FILL (7 bytes per line).

Remember ADDCOMM's statements/commands are *not* tokenised.

Don'ts

Don't SORT A$(0),B$(0) or similar.

Don't place keyword directly after ADDCOMM's statements.

Don't use COMPACT unless a program is completely finished.

Don't use KILLREM to kill REMs on lines jumped to.

Don't use labels such as "number=9", since this will be executed!

Don't try to FILL using the current background colour.

Don't try to use an undefined window.

Error Messages and Codes

Code	Message	Reason
46	Label	A label was entered in immediate mode or LGOTO was used with no label
47	Window	An invalid window number was used (1 to 7 only)
48	Not verfied	VERIFY produced an error
49	Bad colour	The current background colour was used on CFILL and FILL
50	Off!	ADDCOMM was turned off

Defaults on BREAK

On BREAK (or CTRL-BREAK), ADDCOMM sets itself up using:-

```
SCALE 0,1279,0,1023
ROTATE 0,0,0
TRASNS 0,0
SETWIN X,0,0,0,0    for all X (1 yo 7)
PEN 13
OPT X,0    for all X except:
              OPT 4,4
              OPT 8,150
LMOVE 0,0
ANGLE 0
CFILL 0,0,0,0
```

The only exception to this is when ADDCOMM has been turned off, in which case none of the above occurs.

Abbreviations

Most of ADDCOMM's statements can be shortened in the same way as BASIC's statements, but the difference is that ADDCOMM's statements are *not* expanded when listing.

Statement	Abbreivation	Statement	Abbreviation
ADDCOMM	ADD.	LVAR	LV.
ADVANCE	ADVAN.	MEM	ME.
ANGLE	ANG.	OPT	OPT
CFILL	CF.	PEN	PE.
CHAR	CHAR	POPFOR	POP.
CIRCLE	CI.	POPGOS	POPG.
COMPACT	COM.	POPREP	POPR.
ELLIPSE	ELL.	ROTATE	RO.
FILL	FI.	SCALE	SC.
FIND|	FI.	SDRAW	SD.
FKEYS	FK.	SETWIN	SE.
GOODPROG	GOO.	SMOVE	SM.
GREPL|	GR.	SORT	SOR.
KILLREM	K.	SPLOT	SPL.
LCIRCLE	LC.	SREPL|	SR.
LELLIPSE	LEL.	TRANS	TRAN.
LGOTO	LG.	TURN	TU.
LLIST	LL.	UNSCALE	UNS.
LMOVE	LM.	VERIFY	VE.
LPOS	LP.	WIN	WIN

Note that FIN., GR. and SR. do *not* need the | character after them.

ADDCOMM Statement Summary

ADDCOMM
    Switches ADDCOMM off.

ADVANCE <length>
    Advances logo-cursor in the set direction.

ANGLE <angle>
    Sets the direction of logo-cursor.

CFILL <colour>,<colour>,<colour>,<colour>
    (,<colour>,<colour>,
    <colour>,<colour>)
    Sets 'colour-fill' shade to be used in FILL.

CHAR <character code>
    A utility to design/alter user-definable characters.

CIRCLE <x-position>,<y-position>,<radius>(,<start
    angle>,<end angle>, <increment angle>,<plot code>)
    Draws a circle (or polygon/arc/sector.....).

COMPACT (|<start line>,<end line>)
    Intelligently compacts a program.

ELLIPSE <x-position>,<y-position>,<x-radius>,<y-raadius>
    (,<start
    angle>,<end angle>,<increment angle>,<plot code>)
    Draws an ellipse (or polygon/arc/circle/sector....).

FILL <x-position>,<y-position>
    Will fill from a defined point up to a boundary.

FIND| <search characters>(|(")<start line>,<end line>)
    Will find and highlight a set of characters.

FKEYS
    Lists the current function key definitions.

GOODPROG
    Tries to mend a 'Bad program'.

GREPL| <search characters>|<replace characters>(|(")
    <start line>, <end line>)
    Globally replaces one set of characters with another
    set of characters.

KILLREM (|<start line>,<end line>)
    Intelligently deletes REM statements.

LCIRCLE <radius>(,<start angle>,<end angle>,<increment
    angle>,<plot code>)

Draws a circle around the logo-cursor.

LELLIPSE <x-radius>,<y-radius>(,<start angle>,<end angle>,<increment angle>,<plot code>)
Draws an ellipse around the logo-cursor.

LGOTO <label>
Jumps to a labelled line.

LLIST <line>
Lists a line using various options.

LMOVE <x-position>,<y-position>
Moves the logo-cursor.

LPOS <variable>,<variable>
Returns the logo-cursor position.

LVAR (<start character>(<end character>>)
Lists the names of variables in memory.

MEM
Gives a breakdown of memory use.

OPT <option>,<code>
Sets various options.

PEN <plot code>
Sets the plotting used in logo-graphics.

POPFOR
Deletes the current FOR...NEXT loop.

POPGOS
Deletes the current subroutine return data.

POPREP
Deletes the current REPEAT...UNTIL loop.

ROTATE <x-position>,<y-position>,<angle>
Rotates *all* subsequent ADDCOMM graphic output.

SCALE <x-min>,<x-max>,<y-min>,<y-max>
Sets up the user-definable scale.

SDRAW <x-position>,<y-position>
DRAWs using the user-definable scale.

SETWIN <window>,<x-position bottom left>,<y-position bottom left>, <x-position top right>,<y-position top right>
Sets the multiple windows.

```
SMOVE <x-position>,<y-position>
    MOVEs using the user-definable state.

SORT <start element>,<end element>
    Sorts a string array.

SPLOT <plot code>,<x-position>,<y-position>
    PLOTs using the user-definable scale.

SREPL| <search characters>|<replace characters>
    (|(")<start line>,<end line>)
    Selectively replaces one set of characters with
    another set of characters.

TRANS <x-position>,<y-position>
    Translates the scaled screenm by a vector
    displacement.

TURN <angle>
    Changes the direction of the logo-cursor.

UNSCALE <x-position>,<y-position>,<variable>,variable
    Converts scaled to non-scaled co-ordinates.

VERIFY <file>,(,<start address>)
    Verifies that a saved program exactly matches that in
    memory.

WIN <window>
    Changes the window (as set by SETWIN).
```

INDEX

ADDCOMM INSTALLATION INSTRUCTIONS

For

BBC Model 'B' Microcomputer.
BASIC 1 or 2, OS 1.20.

Turn off and disconnect your BBC Computer from mains.

1.) Remove the top cover by undoing four "fix" screws as shown in Fig.1.

2.) Undo the two (or three on early BBC's) keyboard nuts and bolts, see Fig.2.

3.) Gently swing the keyboard round as in Fig.3.

4.) Insert the ADDCOMM ROM with the notch facing away from the keyboard into any spare sideways ROM socket. It will be necessary to bend the pins slightly inwards on a flat surface prior to insertion. (See Fig.4.)

5.) Secure the keyboard and cover.

6.) Switch on the Computer. If all seems to be O.K. type:-

*HELP ADDCOMM

If ADDCOMM does not respond turn off immediately and re-check all steps.

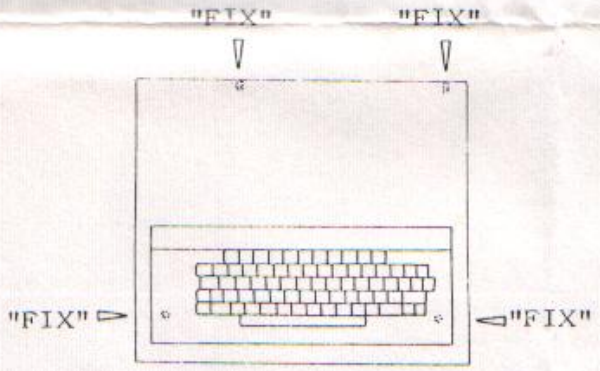


Fig.1

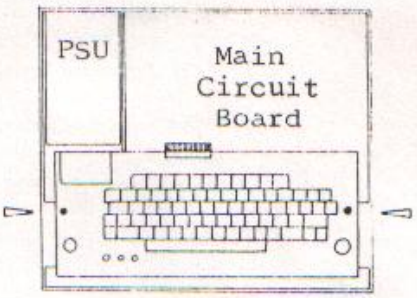


Fig.2

ADDCOMM Installation Diagrams.

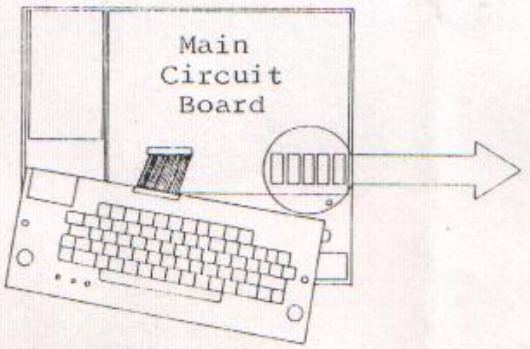


Fig.3

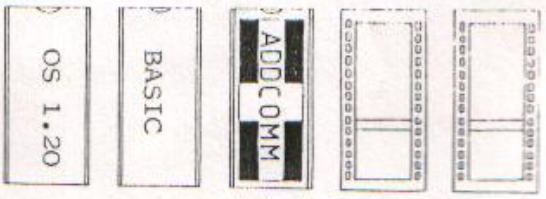


Fig.4