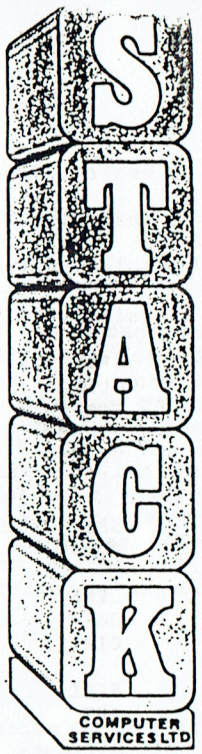
# VIC KIT 3
# USERS GUIDE




Copyright
Stack Computer Services Limited
290-298 Derby Road, Bootle, Liverpool L20 8LN,
England.

| | Page |
|---|---|
| 1 General description | 2 |
| 2 Installing, enabling and disabling VICKIT III on the VIC | 3 |
| 3 Quick reference guide to VICKIT III statements | 5 |
| 4 Using VICKIT III statements for high resolution graphics | 8 |
| GRAPHICS | 8 |
| CLEAR | 9 |
| SCREEN | 9 |
| BORDER | 9 |
| GROUND | 9 |
| TEXT | 10 |
| SET, RESET and INVERT | 10 |
| POINT | 11 |
| LINE | 12 |
| DRAW | 13 |
| CIRCLE | 17 |
| FILL | 18 |
| PUT | 19 |
| COPY | 22 |
| FONT | 23 |
| WRITE | 24 |
| OFF | 25 |
| SWITCH | 25 |
| 5 Examples of programming with VICKIT III | 26 |
| 6 Changes from VICKIT II | 33 |
| 7 Information for machine code programmers | 34 |


VIC and VIC-20 are trademarks of Commodore Business Machines





## GENERAL DESCRIPTION

VICKIT III is a 4K (4096) byte EPROM (Erasable Programmable Read Only Memory) which provides twenty new statements for the VIC which greatly reduce the programming needed to produce high resolution graphics on the television or monitor screen.

Using VICKIT III statements you select colours for the background, border and drawn lines WITHOUT having to remember any addresses for POKE statements. Once the colours you want to use have been set up you can use the statements provided by VICKIT III to produce your drawing on the television or monitor screen attached to your VIC. A finished drawing can be copied to a VIC-1515 Graphic Printer, saved in a BASIC array or reflections produced using VICKIT III's powerful PUT statement. A FILL statement lets you fill out any area (not merely triangles) on the screen by indicating one point inside the area to be filled. Finally the SWITCH statement allows you, using a STACK ROM SWITCH BOARD, to use other EPROMs or ROMs that would normally conflict with VICKIT III and switch easily between these conflicting additions to your VIC thus adding even more flexibility to your VIC.

### Using this manual

First read the section on 'Installing, enabling and disabling VICKIT III' carefully and install VICKIT III in your VIC system, enabling it as described in that section.

If you have used VICKIT II (the predecessor to VICKIT III) before you should read the section 'Changes from VICKIT II' and then the 'Quick reference guide to VICKIT III statements'. These will probably give you enough information to be able to use VICKIT III immediately and you should rarely need to refer to the more detailed section. ALL BASIC PROGRAMS WRITTEN TO USE A PROPERLY INTITIALISED VICKIT II WILL WORK WITHOUT CHANGE ON VICKIT III.

If you have not used VICKIT II you should, after installing and enabling VICKIT III, start to work your way through the descriptions of the individual statements. Each description shows you how to use the statement and you should make sure you understand each one (the best way is to make up an example on your VIC, test it and then alter it in some way) before moving on to the next statement.

Throughout the manual a warning (WARNING) is used to indicate a section of particular importance that you should read carefully before preceding any further.

VICKIT III is available configured to fit in memory addresses $A000 to $AFFF (VICKIT III $A) or in memory adresses $B000 to $BFFF (VICKIT III $B). In this manual memory addresses and user calls are given for both ,$A version in brackets

VICKIT III is normally sold to fit in the $A000 address block and consquently will work with VICKIT I . To enable Storeboard users to benefit from VICKIT III, the device is sold as a $B version as well as the $A version .If a user requires one of these $B versions to work with a VICKIT I at a latter date the user must return the VICKIT III $B to STACK COMPUTER SERVICES Ltd and have it reprogrammed as a VIVKIT III $A. A small handling fee and postage and packing will be chargable .

## Use with VICKIT I

When used with VICKIT I the user must have a $A000 version VICKIT III. To enable the combination the user must first enable VICKIT I with

```
SYS 4096 * 11
```

Then enable VICKIT III with

```
SYS 4096 * 10
```

All commands from the two chips now work with the exception of the VICKIT I commands STEP and TRACE. These two commands will only work if the programme contains no VICKIT III commands.





## Installing, enabling and disabling VICKIT III

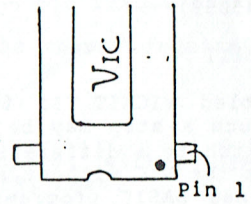
### Installing VICKIT III

First of all turn OFF your VIC and memory expansion. Locate the empty integrated circuit socket on your memory expansion board where VICKIT III is to be placed. This socket should be located at addresses $A000 (45056 decimal) to $BFFF (49151 decimal). On a STACK ROM SWITCH BOARD this can be either of the sockets labelled IC6 or IC8. Which one of these you should use is shown in the table below and depends on how much memory you have and whether you want VICKIT III to be available for initialisation on powering up your VIC or not.

| Memory | VICKIT III available on power up? YES | NO |
|---|---|---|
| Less than 24K | IC8 | IC6 |
| 24K | IC6 | IC8 |

Touch any safely earthed piece of metal to remove any static electrical charge from your body. Carefully unpack the chip from its carrier and examine its pins. If any are bent or out of alignment straighten them VERY GENTLY using narrow pointed pliers.

The VICKIT III EPROM viewed from above




The chip MUST be correctly orientated in its socket at $B000. Using the wrong socket will not harm VICKIT III but you will be unable to initialise it. If you reverse the chip in its socket you will probably DESTROY it instantly so note carefully the notch on the end of the EPROM near pin 1. There should be a corresponding notch in the socket for the EPROM but IF YOU ARE IN ANY DOUBT CHECK WITH THE SUPPLIER OF THE EPROM.

Take great care when inserting the chip into its socket. All of the pins must be aligned simultaneously and slow, firm pressure used to seat the chip. If you bend the pins at this stage they will probably break off when you try to straighten them so DON'T BEND THEM.

If you haven't done this before and don't feel confident then get assistance from someone with experience of chip insertion.

When VICKIT III has been installed turn on the VIC. If the CBM BASIC message does not appear TURN OFF AT ONCE and recheck the steps you took during in the installation.

## Enabling VICKIT III

If you are using VICKIT III in a STACK ROM SWITCH BOARD and chose not to have it available for initialisation on power up you must remember to switch in the socket containing VICKIT III with the appropriate POKE as explained in the documentation.

You can enable VICKIT III by using the following command:

```
SYS(11*4096)
or SYS(45056)
```

You should see a copyright message appear on the screen followed by the READY prompt from the VIC and the flashing cursor. If this does not happen TURN OFF AT ONCE and recheck your installation of VICKIT III, the enabling command that you gave and that the socket you used for VICKIT III is the correct one.

If, after you have checked everything, VICKIT III still does not work you should contact the supplier of the chip for assistance.

If you want to enable VICKIT III from inside a program you should use an alternative form of the enabling command which will not print the copyright message to the screen:

```
SYS(11*4096+3)
or SYS(45059)
```

## Disabling VICKIT III

Having just enabled VICKIT III it may seem strange discussing how to disable it but such a step may be necessary for a number of reasons. You might want to switch to a different set of ROMs or EPROMs which occupy the same addresses in the VIC without switching off. VICKIT III slows down the execution of normal BASIC programs somewhat (although by much less than VICKIT II) so you might want to switch it off to speed up a section of programming. For whatever reason the command:

```
OFF
```

(it can be used as a statement inside a program equally as well) will disable VICKIT III. If you have a STACK ROM SWITCH BOARD the command:

```
SWITCH b
```

will be more useful since it disables VICKIT III and switches in one of the two alternative blocks on the board (block is either A or B).

## Error handling by VICKIT III

Since any printing to the screen while the VIC is working in high resolution mode will display as rubbish VICKIT III intercepts the error and performs the equivalent of a TEXT UPPER statement before displaying the error message.





Quick reference guide to VICKIT III statements.

For each statement the keyword for that statement is given followed by the possible forms of that statement.

### Statements to set up the VIC for high resolution graphics

**GRAPHICS** — GRAPHICS

**CLEAR** — CLEAR n

where n is the number of the colour required from 1 to 8 for normal graphics, 9 to 16 for multicolour mode.

**SCREEN** — SCREEN n

where n is the number of the colour required in the range 1 to 16.

**BORDER** — BORDER n

where n is the number of the colour required in the range 1 to 8.

**GROUND** — GROUND n

where n is the number of the colour required in the range 1 to 16.

**TEXT** — TEXT f

where f is the font required, either U(PPER) (including graphic characters) or L(OWER) case.

### Statements dealing with single points

**SET** — SET (x,y)

where (x,y) is the point to be set.

**RESET** — RESET (x,y)

where (x,y) is the point to be reset.

**INVERT** — INVERT (x,y)

where (x,y) is the point to be inverted.

**POINT** — POINT (x,y),v

where (x,y) is the point whose status is to be found and v is the name of a simple (i.e. non-array) real variable.

LINE

```
LINE (x1,y1)-(x2,y2)
LINE (x1,y1)-(x2,y2),c
LINE (x1,y1)-(x2,y2),c,o
```

where (x1,y1) may be omitted in which case the current drawing position is substituted, c may be either S(ET), R(ESET) or I(NVERT) and o may be either B(OX) or F(ILL).

DRAW

```
DRAW s
```

where s is a string expression containing a sequence of DRAW instructions terminated by colons or semi-colons. Each DRAW instruction can be preceded by B (for Blank) or N (for No update) and is one of the following:

M

Move instruction, either absolute: M(x,y); or relative to current drawing position: Mx,y;

U
D
L
R

Up, Down, Left or Right line instructions followed by a length or variable, e.g. R100; or R DX;. Two such instructions can be separated by a comma, e.g. R 50,U 50;

X

X is followed by the name of a string, e.g. X S$;

C

followed by S(ET), R(ESET) or I(NVERT), e.g. C S;

T

followed by 0, 1, 2 or 3, e.g. T 2;

S

followed by 0, 1, 2, 3, 4, 5, 6 or 7, e.g. S 5;.

CIRCLE

```
CIRCLE (x,y),r
CIRCLE (x,y),r
CIRCLE (x,y),r,c
CIRCLE (x,y),r,c,h/w
CIRCLE (x,y),r,c,h/w,s
CIRCLE (x,y),r,c,h/w,s,f
```

where (x,y) is the centre of the circle or arc, r is its radius, c is either S(ET), R(ESET) or I(NVERT), h/w gives the ratio of the height of the circle to the width, s gives the proportion of the way around the circle to start drawing and f gives the proportion of the way around the circle to finish drawing.

Statements dealing with areas of the screen

FILL

```
FILL (x,y)
FILL (x,y),c
```

where (x,y) is the point at which to start filling and c is





either S(ET), R(ESET) or I(NVERT).

PUT

```
PUT a d (x1,y1)-(x2,y2)
PUT a d (x1,y1)-(x2,y2),r
```

where a is the name of a numeric array element, e.g. CHAR%(0,0), d is either < or >, (x1,y1) and (x2,y2) define a rectangular area on the screen and r is a number in the range 0 to 15.

COPY

```
COPY n,d
COPY n,d,c
COPY n,d,c,s
```

where a file number is given by n in the range 1 to 255, d is the device number of the VIC-1515 GRAPHIC PRINTER attached to your VIC (4 or 5), c is the command number for upper/lower case printing and s is a string to be used as a title.

Statements dealing with text on the high resolution screen

FONT

```
FONT f
FONT f,(w,h)
FONT f,(w,h),(a)
```

where f can be U(PPER), L(OWER) or O(WN), (w,h) gives the position of each character relative to the previous one and (a) gives the address of the font information.

WRITE

```
WRITE r,e
WRITE r,e,e
WRITE r,e,e,e
etc
```

where r gives the rule for combining text with what is already on the screen as in PUT and each e is a string or numeric expression.

Statements disabling VICKIT III

OFF

```
OFF
```

OFF can be either a statement inside a program or a command outside.

SWITCH

```
SWITCH b
```

where b selects a block of ROM on a STACK ROM SWITCH BOARD, and is either A or B. SWITCH can be either a statement inside a program or a command outside.

## Using VICKIT III statements to produce high resolution graphics

VICKIT III adds twenty new statements to BASIC which allow you to use the high resolution graphics capability of your VIC as easily as possible. These statements make up a language which works with lines and points rather than numbers and strings as does the normal VIC BASIC.

When using high resolution graphics you should think of the screen as 176 points wide by 160 points high rather than as 22 columns wide and 23 rows high. The origin of the screen (i.e. the point (0,0)) is almost at the bottom left hand corner of the screen. Statements allow you to SET (to the drawing colour), RESET (to the screen colour) or INVERT (drawing colour to screen colour and vice versa) any point on this screen and for some purposes these statements will be useful. Usually however the more powerful statements LINE, DRAW and CIRCLE will be used since they do with single statements what would otherwise take tens or hundreds of SET, RESET or INVERT statements.

Once a shape has been drawn on the screen it can be FILLed in or PUT into a BASIC array with a single VICKIT III statement. The shape can then be moved to another place on the screen (a collision with another shape can be detected) and even mirrored about a horizontal or vertical line. As we shall see when discussing PUT this statement can be used to produce characters on the high resolution screen as well as graphics although the FONT and WRITE statements are usually better for this purpose.

To draw things on the VIC you must prepare both the VIC and your program for high resolution graphics and five statements are used to do this.

WARNING: with a statement such as:

```
100 IF (X<0) THEN ...
```

where ... represents any VICKIT III statement you MUST follow THEN with a colon, e.g.:

```
100 IF (X<0) THEN : ...
```

This corrects a slight problem caused by the VIC's handling of IF statements.

GRAPHICS

This statement has only one form:

```
GRAPHICS
```

and it should be the first statement in any program which is going to use high resolution graphics. You need only know that its effect is equivalent to a BASIC CLR statement so that if you have defined any variables BEFORE using the GRAPHICS command they will be undefined AFTER it. You cannot use GRAPHICS in a subroutine since it will destroy the information needed for the RETURN to operate properly.

WARNING: if you do not include the GRAPHICS statement in your





program and then go ahead to use any VICKIT III statements you are asking for trouble and your program will probably become corrupted, although CLEAR and SCREEN will detect this condition and signal a ?OUT OF MEMORY error.

CLEAR,
SCREEN,
BORDER and
GROUND

These statements are so similar that they are dealt with together. They are used to indicate the colours to be used for various parts of the high resolution display. The colours are selected by a number in the range 1 to 16 for CLEAR, SCREEN and GROUND and a number in the range 1 to 8 for BORDER. Numbers in the range 1 to 8 will always select the colour shown on the keys 1 to 8 on the VIC keyboard. Thus you use 1 for BLACK, 2 for WHITE and so on up to 8 for YELLOW.

Numbers in the range 9 to 16 are used in CLEAR and SCREEN to provide for multicolour mode (see Examples of programming with VICKIT III) and in GROUND to select one of another 8 additional colours from ORANGE to LIGHT YELLOW.

CLEAR and SCREEN both clear the normal VIC screen and display the high resolution screen but CLEAR also clears the high resolution screen before displaying it. Thus any graphics you have produced before on the screen will be lost. SCREEN allows you the option of producing a drawing in high resolution, returning to the normal text screen with the TEXT statement, perhaps to give instructions, and then redisplaying your original high resolution drawing. The forms of CLEAR and SCREEN are thus:

```
CLEAR n
and SCREEN n
```

where n is the colour number in the range 1 to 16.

BORDER and GROUND do not destroy any information but only select different colours for the border around the screen and the background colour of the screen. The form of the BORDER statement is:

```
BORDER n
```

where n is in the range 1 to 8. The corresponding form of the GROUND statement is:

```
GROUND n
```

where n is in the range 1 to 16. You do not have to be working in high resolution graphics to use the last two statements so BORDER and GROUND can be used at any time in your program to alter the colours of the border and the background screen.

TEXT

Having displayed the high resolution screen with CLEAR or SCREEN the TEXT statement returns the VIC to a cleared text screen of 23 rows by 22 columns. To select either lower case characters to be displayed as normal (upper case characters produced with the shift key) or upper case characters to be displayed as normal (graphics characters produced with the shift key) follow the word TEXT with either LOWER or UPPER (these can be shortened to L or U). Thus the forms of the TEXT statement are:

```
TEXT LOWER
or TEXT UPPER
```

As with BORDER and GROUND these statements can be used at any time in your program to clear the screen and select either LOWER or UPPER case.

A program using VICKIT III statements will start with a GRAPHICS statement to prepare the VIC for high resolution graphics, use CLEAR and SCREEN statements to display the high resolution screen and TEXT statements to revert to printing normal text on the screen. Programs should always make sure they do a TEXT statement before ENDing or STOPping. If you accidently find your VIC still in high resolution graphics when a program ends the simplest way to return to the normal text mode is to press the RESTORE key while holding the STOP key down.

SET,
RESET and
INVERT

These three statements have the following forms:

```
SET (x,y)
or RESET (x,y)
or INVERT (x,y)
```

where x and y are the coordinates of the point to be SET, RESET or INVERTED.

Using SET results in the point (x,y) taking on the colour specified in the last CLEAR or SCREEN statement. RESET results in the point taking on the colour specified in the last GROUND statement while INVERT will change a point currently in the CLEAR or SCREEN colour to the GROUND colour and a point currently in the GROUND colour to the CLEAR or SCREEN colour.

Values for x and y can be numbers, e.g. (100,100), variables, e.g. (X,Y) or expressions, e.g. (I+2,J-2). Thus this sample program draws a box in the middle of the screen:

```
10 GRAPHICS:REM ALWAYS FIRST STATEMENT
20 CLEAR 7:GROUND 2:REM DRAW BLUE ON WHITE
30 FOR X=80 TO 100
40    SET (X,80):SET (X,100)
```





```
50 NEXT X:REM BOTTOM AND TOP
60 FOR Y=80 TO 100
70    SET (80,Y):SET (100,Y)
80 NEXT Y:REM LEFT AND RIGHT SIDES
90 GET A$:IF (A$="") THEN 110
100 TEXT LOWER
110 END
```

After the box has been drawn line 110 waits until a key is pressed before using TEXT to return the VIC to the normal text mode.

Looking at the program you can see that the corners of the square will have been SET twice. SETting a point that is already SET or RESETting a point that is already RESET still has the effect intended, but INVERTing a point twice will leave the point in its original colour. If you alter the four SET statements to INVERT statements and RUN the program the corners will be the colour of the screen rather than the colour of the box.

You should also notice that although the shape drawn on the screen should have been square it looks more like a rectangle. Each side of the shape is 20 units long, where a unit is the distance between one dot and the next, but this distance differs between horizontal and vertical lines since vertical lines look about 0.6 of the length of a similar horizontal line. To get a more realistic square line 30 should read:

```
30 FOR X=80 TO 92
```

and line 90 should read:

```
90 SET (92,Y)
```

WARNING: if the point (x,y) used in SET, RESET or INVERT does not lie on the screen area, i.e. x is outside the range 0 to 175 and/or y is outside the range 0 to 159, but the coordinate is in the range 0 to 255 an out of range x coordinate is replaced by 175 and an out of range y coordinate by 159. Coordinates outside this extended range will give a ?ILLEGAL QUANTITY ERROR. This applies to any VICKIT III statement which uses coordinates.

POINT

The POINT statement has only one form:

```
POINT (x,y),v
```

where (x,y) is a point as described above and v is the real, non-array variable, e.g.:

```
PT
```

but not:

```
PT%
or PT(0,0)
```

When the POINT statement has been executed the variable given will be set to 0 if the point was RESET, 1 if it was SET and -1 if the point was off the screen. These three values mean that you can then use the BASIC ON ... GOTO statement as follows:

```
POINT (X,Y),PT:ON PT+2 GOTO 1000,2000,3000
```

which will GOTO line 1000 if the point (X,Y) was off the screen, 2000 if it was RESET and 3000 if it was SET. The PUT statement has a more powerful technique for detecting the state of a point or group of points but for some purposes the POINT statement is still useful.

## LINE

Using LINE you can draw lines, boxes (squares or rectangles) and filled-in boxes. The form of a LINE statement to draw a line from the point (x1,y1) to (x2,y2) is:

```
LINE (x1,y1)-(x2,y2),c
```

where (x1,y1) can be omitted (the - sign cannot) and c is either S(ET), R(ESET) or I(NVERT). If you omit c you must also omit the comma and the value for c is taken to be S(ET). The program which drew a box in the section on SET, RESET and INVERT can now be written as:

```
10 GRAPHICS
20 CLEAR 7:GROUND 2
30 LINE (80,80)-(100,80),S
40 LINE (100,80)-(100,100),S
50 LINE (100,100)-(80,100),S
60 LINE (80,100)-(80,80),S
70 GET A$:IF (A$="") THEN 70
80 TEXT LOWER
90 END
```

If you omit (x1,y1) then the coordinates of the end point of the last LINE, updated DRAW (see next section) or WRITE statement are substituted. Another version of the box drawing program replaces lines 40, 50 and 60 with:

```
40 LINE -(100,100),S
50 LINE -(80,100),S
60 LINE -(80,80),S
```

The ,S in the above example could also be omitted.

Drawing boxes on the screen is such a common requirement that a special form of LINE statement allows you to draw a box





with just one statement:

```
LINE (x1,y1)-(x2,y2),c,o
```

where o is either B(OX) or F(ILL). With B the box is drawn with 4 lines from (x1,y1) to (x2,y1), (x2,y1) to (x2,y2), (x2,y2) to (x1,y2) and from (x1,y2) to (x1,y1). The endpoint of the last line drawn is thus (x1,y1) and this is used if a future LINE statement omits (x1,y1).

With F a filled-in box is drawn by drawing horizontal lines from left to right and bottom to top so that the endpoint of the last LINE drawn is (x2,y2).

## DRAW

The form of the DRAW statement is quite simple:

```
DRAW s
```

where s can be string contant, variable or expression, e.g.:

```
   "R20:U10:L5:U10:L10:D10:L5:D10:"
or A$
or "B"+A$+"R10:U10:L10:D10:"
```

As the last example shows you can use string operations to 'add' shapes together. You could also use LEFT$, RIGHT$ and MID$ to 'subtract' parts of a shape or select parts of a shape for drawing.

As suggested by the examples above s consists of a number of instructions separated by colons (or semi-colons). Each instruction starts with one of the following, possibly preceded by B or N:

| | |
|---|---|
| M | Move |
| U | Up |
| D | Down |
| L | Left |
| R | Right |
| X | eXecute |
| C | Colour |
| T | Turn |
| S | Scale |

If an instruction which normally produces a line (M, U, D, L or R) is preceded by B (for Blank) then the line no longer appears (this is most often used with M) while if an instruction which normally produces a change in the current position is preceded by N (for No update) that instruction no longer produces the change.

DRAW instructions

M

The Move intruction has two forms depending on whether an absolute or a relative move is required. The absolute move has form:

M(x,y):

where (x,y) is the point to be moved to. Unless B precedes the M a line will be drawn from the current drawing position to the point (x,y).

Relative moves have the form:

Mx,y:

where x and y are displacements which are added to (or subtracted from if preceded by a minus sign) the current drawing position to produce the point to be moved to. Again a visible line is drawn unless the B option is used. Neither absolute or relative moves are affected by the current Scale or Turn.

To illustrate the Move instruction here is another way to produce the box drawn in previous sections:

```
10 GRAPHICS
20 CLEAR 7:GROUND 2
30 DRAW "BM(80,80):"
40 DRAW "M20,0:M0,20:M-20,0:M0,-20:"
50 GET A$:IF (A$="") THEN 50
60 TEXT LOWER
70 END
```

Where a number is used as the displacement in relative moves the + sign only may be omitted but if variables are used the + and/or - signs must be included and the variables must be positive.

U
D
L
R

The Up, Down, Left and Right instructions are used in DRAW to produce shapes which are 'relocatable' and can be drawn anywhere on the screen. While relative moves can produce shapes which are relocatable such shapes cannot be Scaled or Turned.

You will probably have guessed by now that Right draws a line to the right, Up a line up and so on. With this knowledge here is yet another way to produce the small box in the middle of the screen:

```
10 GRAPHICS
20 CLEAR 7:GROUND 2
30 DRAW "BM(80,80):R20:U20:L20:D20:"
```





```
40 GET A$:IF (A$="") THEN 40
50 TEXT LOWER
60 END
```

You are not restricted to drawing lines Up, Down, Left or Right since you can combine a horizontal line with a vertical one to produce a sloping line. If you separate a Left or Right instruction from an Up or Down one by a colon (or semi-colon) they are draw separately, so that:

"R100:U100:"

draws a line to the Right and then one Up. If you change the first : to a comma:

"R100,U100:"

the line that is drawn is a diagonal one, Right 100 units and Up 100 units. The following DRAW instructions draw a triangle on the screen (note that L or R must come first):

"R40:U30:L40,D30:"

You can of course precede any Up, Down, Left or Right instruction, whether combined or not, with B or N. The values that are used for length can be constants, e.g. 100, 40 or 30 as used above, or variables, e.g. DX, DY or SIDE, but must not include a sign since the instruction letter tells DRAW which way to move. If you use variables you can use one string to draw different sized shapes, e.g. by setting S to a suitable value the instructions:

"R S:U S:L S:D S:"

will always draw a square of size S (note that spaces can be used freely in DRAW instructions) with its bottom left hand corner at the current drawing position.

X

The eXecute instruction allows you to perform the equivalent of a BASIC GOSUB statement within a string of DRAW instructions. It is followed by the name of a string variable which must also hold one or more DRAW instructions. DRAW will then interpret the instructions in this new variable and when finished return to the original string after the X instruction. The 'subroutine string' can in turn contain an X instruction and this nesting can be repeated, although if there is insufficient room on the BASIC stack for an X instruction a ?OUT OF MEMORY error will be reported.

As an example of the X instruction suppose that you are drawing a square and want to place a symbol at each corner. Placing the DRAW instructions for the symbol in S$ you can use the following DRAW instructions:

```
"R S:X S$:U S:X S$:L S:X S$:D S:X S$:"
```

Now by just altering S$ you can alter the symbol drawn at each corner.

After an X instruction DRAW does not return to the last drawing position before the X instruction. Thus subroutine strings should usually return to their starting position.

The current Colour, Turn and Scale values are carried through to the subroutine string and any changes made to these values in the subroutine string are carried back to the string in which the X instruction occurred.

C

The Colour instruction allows you to select the colour of any lines drawn in future until another Colour statement or the end of the current DRAW statement. The form of the Colour instruction is:

```
Cc:
```

where c is S (SET), R (RESET) or I (INVERT).

T

The Turn instruction allows you to rotate any further Up, Down, Left or Right instruction by any multiple of 90 degrees. The Turn instruction takes the form:

```
Tt:
```

where t is either a contant or a variable with the value 0, 1, 2 or 3. The effect can best be seen with a simple program which draws a rectangle and then rotates it through 90 degrees, then 180 degrees and finally through 270 degrees:

```
10 GRAPHICS
20 CLEAR 7:GROUND 2
30 RECT$="R 40:U 10:L 40:D 10:"
40 DRAW "BM(80,80):"
50 FOR F=0 TO 3
60   DRAW "T F:X RECT$:"
70 NEXT F
80 GET A$:IF (A$="") THEN 80
90 TEXT LOWER
100 END
```

Turns are not additive so that:

```
"T 1:T 1:"
```

is not the same as:

```
"T 2:"
```





S

The Scale instruction allows you to scale all subsequent Up, Down, Left or Right lines. Lines can be made larger or smaller by a factor of 1, 2, 4 or 8. The form of the Scale instruction is:

```
Ss:
```

where s is either a constant or a variable with a value in the range 0 to 7. The values have the following scaling effects:

```
0  full size
1  half full size
2  quarter full size
3  eighth full size
4  full size
5  twice full size
6  four times full size
7  eight times full size
```

You can see the effect of the Scale instruction by changing line 60 in the Turn program above to:

```
60 DRAW "T F:S F:X RECT$:"
```

when the rotating rectangles get smaller as they rotate. Inserting line 55:

```
55 G=F+2
```

and altering line 60 to:

```
60 DRAW "T F:S G:X RECT$:"
```

gives rotating rectangles which get larger rather than smaller.

WARNING: when a length is to be scaled with a factor of 1, 2 or 3 it must be in the range 0 to 255 BEFORE the scaling is performed. If scaling with a scale factor of 5, 6 or 7 the scaled length must not be more than 255.

CIRCLE

The CIRCLE statement produces not only circles but ellipses and arcs. The form of the statement is:

```
CIRCLE (x,y),r,c,h/w,s,f
```

where (x,y) is the centre of the circle, r is the radius of the circle, c is the colour of the circle (S(ET), R(ESET) or I(NVERT)), h/w gives the height to width ratio of the circle, s gives the proportion of the way around the circle that drawing will start and f gives the proportion of the way around the circle that drawing will end. c, h/w, s and f are

all optional and if omitted have values S(ET), 1, 0 and 1. If any of the optional values are omitted you must give all of the preceding optional values a value, e.g.:

```
CIRCLE (100,100),50
CIRCLE (100,100),10,S,1,0,.75
CIRCLE (50,50),20,,,,.25
```

and so on. If you want to use the default for any optional value simply omit the value but leave the correct number of commas in the statement.

The height/width ratio is the value of the height of the shape to be drawn divided by its width. For a circle of course this value is 1, for an ellipse which is taller than it is wide it will be greater than 1 while for an ellipse wider than it is tall it will be less than 1. The width of the circle will always be twice the radius if h/w is greater than 1 and the height will be twice the radius if h/w is less than 1 (the width will then be an integer multiple of the height, e.g. h/w=0.5 generates a width of twice (1/0.5) the height).

The start and finish points (s and f) give the proportion of the distance around the 'true' circle that the circle statement will start to draw. They are used to produce arcs of circles or ellipses rather than complete circles or ellipses. VICKIT III starts to draw the circle at 3 o'clock and proceeds clockwise. Thus .25 of the way around the circle is at 6 o'clock, .5 at 9 o'clock, .75 at 12 o'clock and so on. You should be aware that because of the resolution of the VIC screen the values for s and f may need a little experimentation to produce exactly the required result, especially when used with with small radii.

## FILL

The FILL instruction allows you to select an area on the screen which is RESET and surrounded by SET lines, e.g. a shape produced by DRAW, or an area on the screen which is SET and surrounded by RESET lines and FILL that area with SET or RESET points respectively. If the shape is a rectangle then you can do exactly the same thing with the F(ILL) option in LINE but using the FILL statement the shape need not be rectangular and you need not know the size of the area before FILLing.

The form of the FILL statement is:

```
FILL (x,y),c
```

where (x,y) is the point at which FILL will start to work and c is S(ET), R(ESET) or I(NVERT). This gives the colour of the boundary which stops FILL (using INVERT means that the colour to stop FILL will be the 'opposite' of the colour of the point (x,y)).





Some examples of the use of the FILL statement are as follows:

```
10 GRAPHICS:CLEAR 7:GROUND 2
20 DRAW "BM(5,5):"
30 DRAW "R15:U5:L5:U5:L5:D5:L5:D5:"
40 FILL (6,6),S
50 GET A$:IF (A$="") THEN 50
60 FILL (6,6),R
70 GET A$:IF (A$="") THEN 70
80 TEXT LOWER
90 END
```

This program will draw a shape on the screen at the bottom left hand corner, FILL it and then wait for a key to be pressed. It will then FILL it again but this time RESET the area. If no colour is specified FILL uses S(ET) and if you attempt to FILL an area which is not bounded the entire screen will be FILLed (the edges of the screen count as boundaries).

Some very irregular shapes may cause a ?OUT OF MEMORY error but this is unlikely. As FILL proceeds it keeps a count of how many points it SETs or RESETs and stored this count in two memory locations where they can be PEEKed to give an estimate of the area of the shape FILLed. After a FILL the expression:

```
256*PEEK(221)+PEEK(220)
```

gives an estimate of the area of the shape FILLed. For some irregular shapes this may be a few points out since the algorithm used for FILL may sometimes SET or RESET a point more than once.

## PUT

The PUT statement can be described simply but can be used in ways which are not at all obvious from its description. It moves areas of high resolution graphics information between the screen and BASIC arrays (if you have not done much work with arrays now is the time to start). On the screen these areas form images while in an array they form numbers which could be manipulated using normal BASIC commands although you will normally use PUT again to do any manipulation. PUT has the following form:

```
PUT a d (x1,y1)-(x2,y2),r
```

where a is the name of any numerical array element, e.g.:

```
   SC%(1,1)
or PT(10,10,10)
or CHAR%(0,1)
```

(although you can use real or integer arrays you will see

that integer arrays are usually easier to handle). It need not have been defined in a DIM statement if you are sure that there is enough space in the default dimensioned size of the array (11 elements for every dimension) to hold the information you are going to put in it.

The direction of the move is given by d, which is either > for moving array contents to the screen and < for moving screen contents to memory (the author apologises for the syntax).

(x1,y1)-(x2,y2) defines the rectangular area to be moved or to be moved to in terms of a pair of diagonally opposite corners of the area.

A rule for combining what is on the screen with what is in the array is given by the number r which is a number in the range 0 to 15. For simple PUT statements this can be omitted in which case it is taken as 3 if moving to the screen or 5 if moving to the array. A table at the end of this section gives details of the 16 possible rules but useful rules are pointed out as the details of PUT are filled out. While experimenting with PUT ignore the rule and PUT will select the most useful one for you automatically.

The simplest use of PUT is to move a shape around the sceen. Use the other facilities of VICKIT III to produce a shape on the screen, use PUT to place it in a BASIC array and then use PUT again to return it to the screen in a different location. For example:

```
10 GRAPHICS:CLEAR 7:GROUND 2
20 DRAW "BM(0,0):"
30 DRAW "R15:U5:L5:U5:L5:D5:L5:D5:"
40 PUT SHAPE(0) < (0,0)-(15,10)
50 PUT SHAPE(0) > (100,100)-(115,110)
60 GET A$:IF (A$="") THEN 60
70 TEXT LOWER
80 END
```

draws a shape on the sereen, reads that shape into the array SHAPE and then places a copy of the shape on the screen at a different place.

Array sizes

When you use PUT you will need to calculate the size of the area on the screen that you are interested in to be sure that the array you are using is the right size although no real harm will occur if the array is too small (except that you will not store all the shape) or if it is too large (except that you will be wasting space). Multiply the dimensions of the rectangular area you are interested in (16*11=176 in the above example), divide by 40 for a real array, 16 for an integer and round the result UP to the nearest integer (if need be). The result is the minimum number of elements that





the array must have for PUT to work correctly. For the above example divide 176 by 40 to get 4.4 and round up to 5 so that DIM SHAPE(4) (remember arrays have zero'th elements) would provide enough space. Had the array been integer you would have divided 176 by 16 to get exactly 11. Thus DIM SHAPE%(10) would provide enough space.

Reflections

PUT does not require that you use the same pair of corners for PUTting to the screen as for PUTting to the array. In fact by altering the relationship of the two corners you can reflect a high resolution graphics shape. To see this add line 42 to the example above:

```
42 PUT SHAPE(0) > (100,100)-(115,90)
```

which produces the shape reflected about a horizontal line. To reflect about a vertical line add:

```
44 PUT SHAPE(0) > (100,100)-(85,110)
```

and to reflect about A horizontal and then a vertical line add:

```
46 PUT SHAPE(0) > (100,100)-(85,90)
```

By specifying a large enough array (DIM SCREEN(704) or DIM SCREEN%(1760)) you can even read the entire screen into a BASIC array and then reflect it to produce kaleidoscope patterns.

Other points to note about the PUT statement are that you need not always specify the first element of the array and that the shape and/or size of the area on the screen need not be the same for PUTting to the screen as for PUTting to the array (an example in section 5 uses this to produce italic characters on the screen).

Collisions

Immediately after a PUT statement and before any other VICKIT III statement the expression (PEEK(229) AND 1) is 0 if there were no points set in the screen area examined or 1 if there were some points set. Similarly (PEEK(229) AND 2) is 0 if there were no points set in the array examined or 2 if there were some points set. This collision detection feature allows you to discover easily if placing a shape on the screen will overwrite a shape already there. An example in section 5 shows how to use this facility to discover what are the maximum and minimum coordinates of all the set points on the screen, perhaps in order to draw a border around them.

Rules

As mentioned earlier PUT allows you to specify a rule for

combining what is in the array with what is on the screen at the corresponding point. The result is then placed in the array or on the screen depending on the direction specified with PUT. The rule is a number in the range 0 to 15 and the choices are described below together with their effects. A 0 result produces a RESET point if placed on the screen and a 1 a SET point. A represents an array point and S the corresponding screen point.

| Rule number | Result | |
|---|---|---|
| 0 | always 0 | |
| 1 | A and S | (1 if A=1 and S=1) |
| 2 | A and (not S) | (1 if A=1 and S=0) |
| 3 | A | (1 if A=1) |
| 4 | (not A) and S | (1 if A=0 and S=1) |
| 5 | S | (1 if S=1) |
| 6 | A xor S | (1 if A<>S) |
| 7 | A or S | (1 if A=1 or S=1 or both=1) |
| 8 | not (A or S) | (1 if A=0 and S=0) |
| 9 | A equiv S | (1 if A=S) |
| 10 | not S | (1 if S=0) |
| 11 | A or (not S) | (1 if A=1 or if S=0) |
| 12 | not A | (1 if A=0) |
| 13 | (not A) or S | (1 if A=0 or S=1) |
| 14 | not (A and S) | (1 if A=0 or S=0 or both=0) |
| 15 | always 1 | |

Some of these rules seem useless while others simply provide alternative ways of doing things, e.g. using rule 15 with direction > gives another way of filling a rectangle. Some can be used in tricky ways, e.g. using rule 3 with direction < makes no actual changes but does allow you to detect any non-zero elements in the array using collision detection. For some examples see section 5.

COPY

The COPY statement is only of use when you have a VIC-1515 GRAPHIC PRINTER attached to the serial bus on your VIC, since it allows you to get a printed version of whatever is drawn on the high resolution screen of your VIC using the graphics capabilities of the printer. The form of the COPY statement is very like that of the normal OPEN statement on the VIC:

COPY n,d,c,s

where n is a file number in the range 1 to 255, d is the device number of the printer (4 or 5), c is a command whfch is normally only of use if you include s whch is a string that will be printed at the top of the copy as a title. If the string is to be displayed in upper case (and possibly graphics) then you should use a value of 0 for c but if lower case is wanted use a value of 7 for c. You can use the control code 14, described on page 24 of the 1515 printer manual, in the string s in order to produce double width characters and hence make the title more striking.





VICKIT III finds the maximum and minimum y coordinates of set points on the screen and will only copy to the printer the section of the high resolution screen between those limits, thus cutting down on the time taken to produce the copy (a complete copy of the screen can take up to 7 minutes). If you want to stop the printer while it is COPYing a screen simply press the Commodore key on the VIC.

When copied to the printer every set dot on the screen is expanded to a 2 by 2 dot, thus making the size of the printed copy as close as possible to that of the screen. However, whereas the length of a horizontal line drawn on the screen is about 0.6 of the length of the same line drawn vertically the ratio on the printer is the other way about. Thus a square drawn on the screen and copied to the printer will look taller than it is broad by a factor of about 1.14, closer than the screen to a proper square but still not totally correct.

FONT

Both FONT and the statement that follows it, WRITE, are used to make the job of mixing text and graphics on the VIC's high resolution screen as easy as possible. FONT is used to specify the type of characters that are to be printed on to the high resolution screen and WRITE, which has a form very close to PRINT, actually places the characters there. FONT has one of the forms:

```
FONT c
FONT c,(dx,dy)
FONT c,(dx,dy),(a)
```

where c selects the type of characters to be used in WRITE and is either U(PPER), L(OWER) or O(WN). U(PPER) and L(OWER) select one of the two VIC character sets as the font currently in use, either the upper/graphics case or lower/upper case one. (dx,dy) gives the spacing between characters placed on the screen using WRITE and is normally (8,0), i.e. each character is 8 points to the right of the previous one and at the same level. By using a spacing of (0,-8) any text written to the screen will be written vertically downwards, a spacing of (0,8) will produce text vertically upwards and so on. (a) is only used with the O(WN) font, which is a user defined character set based on an 8 by 8 matrix in exactly the same way as the two resident VIC character sets. When the O(WN) font is selected (a) gives the address of the first byte of the character set, thus:

FONT OWN,(8,0),(32768)

has exactly the same effect as:

FONT UPPER

since the upper case font is stored starting at $8000 (32768

decimal) in the VIC. By defining the spacing as, say, (4,0) one could define a thin character set only 4 dots wide instead of the standard 8 and WRITE would cope with this difference. An example of the use of the FONT and WRITE is given in section 5 together with information about constructing an O(WN) font.

RITE

The WRITE statement is very like the normal PRINT statement on the VIC since it is designed to perform much the same task. It takes a variable number of expressions, either string or numeric, and prints them to the screen. The differences from PRINT can be seen by looking at the form for WRITE:

WRITE r,e ...

where r represents a rule and is a number in the range 0 to 15, exactly the same as used in the PUT statement. e ... represents a variable number of expressions each of which must be separated from the expression that precedes it and the one that succeeds it by a comma. The comma does not provide any of the formatting that is normally associated with PRINT and is there merely to separate the expressions. Each expression is evaluated in turn, converted to a string if necessary, and then written to the screen using a form of the PUT statement. The starting position for the first character is the current drawing position (i.e. the endpoint of the last LINE, DRAW or WRITE statement). The value of r determines how each character is combined with whatever is on the screen. For example a value of 7 (i.e. (A or S)) will mean that as each character is written to the screen it will not overwrite what is already there but will merge with it.

The rule can be omitted (the leading comma cannot) in which case a value of 3 is selected automatically. Using a value of 12 (i.e. (not A)) would result in reverse field characters being displayed on the screen but there is an easier way of doing this using normal VIC control characters which can be incorporated into any string. The WRITE statement handles REVERSE ON, REVERSE OFF, HOME, CLEAR, CURSOR RIGHT and CURSOR LEFT in exactly the same way as the normal PRINT statement but the RETURN, CURSOR UP and CURSOR DOWN characters are handled slightly differently. The RETURN character (every WRITE statement writes one of these after the last expression written) returns not to the left hand side of the screen one row further down but one row (8 points always) down immediately under the position at which the WRITE statement started. CURSOR UP and DOWN do not move up and down by 8 points but rather by 4, allowing true subscripts or superscripts on the screen, e.g. in chemical or mathematical equations.

There is no scrolling of the screen and the results of trying to WRITE further to the left than the left hand side of the



screen (with spacing (-8,0)) or further up than the top of the screen (with spacing (0,8)) are undefined.

OFF

The OFF statement (or command since it can be used outside a program) has only one form:

OFF

Its purpose is to remove the links between VIC BASIC and VICKIT III so that VICKIT III statements or commands are no longer recognised and will cause ?SYNTAX errors. You should use this when wanting to run a normal BASIC program that little bit faster or when switching over to another EPROM or ROM which is incompatible with VICKIT III. If the other EPROM or ROM is situated in another $B000 socket in a STACK ROM SWITCH BOARD the last of the new statements supplied by VICKIT III is more useful.

SWITCH

The SWITCH statement is really only of use if you have a STACK ROM SWITCH BOARD which allows more than one ROM or EPROM to share the same memory addresses in the VIC (although not at the same time). As with OFF the SWITCH statement can be used outside a program as a command and has the form:

SWITCH b

where b is either A or B since the two blocks of ROM space on the STACK ROM SWITCH BOARD are known as A and B. The SWITCH statement first removes the links between VIC BASIC and VICKIT III in exactly the same way as OFF and then switches in one of the two blocks of ROM, either the A or the B block. It does this by moving a small section of programming into the cassette buffer and then executing this program which performs the equivalent of the POKE that you will have used before. If you are interested in details of the machine code program (its length and location) then you should read section 7 of this manual. Part of the programming moved allows one to reinitialise VICKIT III with a single SYS(836) command or statement rather than a POKE and a SYS. You should note that VICKIT III's support of the STACK ROM SWITCH BOARD does not mean that you can avoid giving the POKEs to lower the top of memory if you have a full 24K memory expansion.

WARNING: it is your responsibility to make sure that any SWITCH commands or statements reference the correct block to be switched in, i.e. SWITCH B if VICKIT III is located in IC6, SWITCH A if it is in IC8. Failure to observe this precaution will cause problems when you come to reinitialise VICKIT III with SYS(836).

## Examples of programming with VICKIT III

The samples of programming that have been shown so far have been fairly simple ones, designed to illustrate a certain point. The programs in this section are designed to give you a feel for the sort of things that you can do with VICKIT III's high resolution graphic instructions and should show you a number of techniques you will be able to apply to your own programs.

All the programs listed here will start with:

```
10 GRAPHICS:CLEAR 7
```

and end with:

```
1000 GET A$:IF A$="" THEN 1000
1010 TEXT LOWER
1020 END
```

and so these 4 lines will not be included in the listings.

Program 1

```
20 REM RANDOM INVERTED SQUARES
30 FOR X=0 TO 175 STEP 5
40   LINE (X,0)-(X,159)
50 NEXT X
60 FOR I=1 TO 100
70   LINE (INT(RND(TI)*176),INT(RND(TI)*160))-
          (INT(RND(TI)*176),INT(RND(TI)*160)),I,F
80 NEXT I
```

You will be able to get line 70 all on one line.

Program 2

```
20 REM RANDOM BOXES
30 FOR I=1 TO 100
40   LINE (INT(RND(TI)*176),INT(RND(TI)*160))-
          (INT(RND(TI)*176),INT(RND(TI)*160)),S,B
50 NEXT I
```

Program 3

```
20 REM RANDOM STICKS
30 FOR I=1 TO 100
40   LINE (INT(RND(TI)*176),INT(RND(TI)*160))-
          (INT(RND(TI)*176),INT(RND(TI)*160)),S
50 NEXT I
```

Program 4

```
20 REM SIN AND COS CURVES
30 FOR X=0 TO 175
40   SET (X,80+SIN(X/16)*80):SET (X,80+COS(X/16)*80)
50 NEXT X
```





Program 5

```
20 REM OFFSET REPEATED, FILLED-IN BLOCKS
30 DRAW "BM(0,0):R15:U5:L5:U5:L5:D5:L5:D5:"
40 FILL (7,7)
50 PUT SQ(0,0)<(0,0)-(15,10)
60 FOR X=0 TO 160 STEP 17
70   FOR Y=0 TO 150 STEP 12
80     PUT SQ(0,0)>(X,Y)-(X+15,Y+10)
90   NEXT Y
100 NEXT X
```

Program 6

```
20 REM MIXING TEXT AND GRAPHICS SEE NEXT SECTION FOR DETAILS
30 DIM CHARS%(3,26)
40 FOR I=0 TO 26
50   FOR J=0 TO 7
60     POKE 4096+J,PEEK(32768+8*I+J)
70   NEXT J
80   PUT CHARS%(0,I)<(0,152)-(7,159)
90 NEXT I
100 REM HAVING READ IN CHARACTERS WRITE A MESSAGE
110 LINE (88,0)-(88,59)
120 LINE -(175,59)
130 K=86:LET TEXT$="MIRROR"
140 FOR I=1 TO LEN(TEXT$)
150   PUT CHARS%(0,ASC(MID$(TEXT$,I,1))-64) > (K,0)-(K-7,7)
160   K=K-8
170 NEXT I
180 K=90
190 FOR I=1 TO LEN(TEXT$)
200   PUT CHARS%(0,ASC(MID$(TEXT$,I,1))-64) > (K,0)-(K+7,7)
210   K=K+8
220 NEXT I
230 K=100
240 FOR I=1 TO LEN(TEXT$)
250   PUT CHARS%(0,ASC(MID$(TEXT$,I,1))-64) > (K,61)-(K+7,68)
260   K=K+8
270 NEXT I
280 K=100
290 FOR I=1 TO LEN(TEXT$)
300   PUT CHARS%(0,ASC(MID$(TEXT$,I,1))-64) > (K,57)-(K+7,50)
310   K=K+8
320 NEXT I
330 DRAW "BM(50,110):"
340 FOR I=0 TO 75 STEP 0.2
350   IF I=0 THEN: LINE -(50+SIN(I*40)*50,110+COS(I*40)*50),R
360   LINE -(50+SIN(I*40)*50,110+COS(I*40)*50)
370 NEXT I
```

Note that variables whose names clash with VICKIT III statement words, e.g. TEXT$ in line 130 can be used providing LET is used. TEXT$ in the middle of a statement causes no problem. This feature may be removed in future versions of VICKIT high resolution graphics chips so it should be'

avoided if at all possible.

Also note in line 340 the need for the extra colon after THEN.

By adding ,12 to the PUT statements in lines 150, 200, 250 and 300 the writing will be displayed in reverse field.

Program 7

```
20 REM WORK OUT WHAT THIS DOES BEFORE READING THE NOTES
30 FOR I=1 TO 100
40   LINE -(176*RND(TI),160*RND(TI)),S,B
50 NEXT I
60 DIM SCREEN(703)
70 PUT SCREEN(0) > (0,0)-(175,159),6
80 PUT SCREEN(0) < (0,0)-(175,159),6
90 PUT SCREEN(0) > (0,0)-(175,159),6
100 GET A$:IF A$="" THEN 80
```

You will have to press a key twice to halt this program. The program depends on a property of the xor function (hence the ,6 on each PUT statement). If you have an image stored in an array, and an image area of the same dimensions on the screen, then PUTting the array to the screen with rule 6, then PUTting the new screen to the array again with rule 6 and finally PUTting the new array to the screen with rule 6 results in what was originally on the screen being stored in the array and what was originally in the array being stored on the screen. More importantly we have swapped a screen and an array image without using a second array the same size as SCREEN.

Program 8

```
20 REM ITALIC AND ITALIC BOLD CHARACTERS
30 DIM ITALICS%(8,26):CLEAR 2:GROUND 2:BORDER 2:FONT UPPER
40 FOR I=0 TO 25
50   DRAW "BM(0,0):"
60   WRITE 3,CHR$(ASC("A")+I)
70   PUT ITALICS%(0,I) < (0,0)-(15,7)
80 NEXT I
90 CLEAR 7
100 FOR I=0 TO 20
110   X=I*8
120   DRAW "BM(X,100):"
130   PUT ITALICS%(0,I) > (X,100)-(X+14,107),7
140 NEXT I
```

Line 30 of this program sets the border, ground and drawing colour to white. Thus anything drawn on the screen will not show up. Since lines 30 to 80 are concerned only with setting up the italic type face in the array ITALICS% there is no need for this to be visible on the screen. As each character is written on to the screen in an 8 by 8 square it is stored in the ITALICS% array as a 16 by 8 rectangle. When line 130 PUTs the character back to the screen it places it in a 15 by 8 rectangle. Thus the second and subsequent lines are moved right by one point from the line below giving the italic effect. The rule in line 130 is specified as 7





(i.e. (A or S)) to prevent the overhang of one character (its kern) being destroyed by a subsequent character (try changing the rule to 3 and see what happens).

If you add line 115:

```
115   PUT ITALICS%(0,I) > (X+1,100)-(X+15,107),7
```

or:

```
115   PUT ITALICS%(0,I) > (X,101)-(X+14,108),7
```

you will get italic bold characters (in two versions).

Program 9

```
20 REM TRICOLOUR BY E. HULME
30 CLEAR 11:GROUND 6:BORDER 7
40 POKE 36878,(PEEK(36878) AND 15) OR (7*16)
50 X=80:REM LINE 40 SELECTS AUXILIARY COLOUR
60 FOR Y=0 TO 70
70   SET (X,Y):REM FOREGROUND COLOUR
80 NEXT Y
90 FOR I=0 TO 60
100   X=2*I+1:SET (X,I):REM BORDER COLOUR
110 NEXT I
120 FOR I=0 TO 70
130   X=168-I*2:X1=X+1
140   SET (X,I):SET (X1,I):REM AUXILIARY COLOUR
150 NEXT I
```

When using the CLEAR or SCREEN statements the number used is usually in the range 1 to 8 for normal high resolution graphics, but a number in the range 9 to 16 selects a graphics mode called the multicolour mode. In this mode you can produce graphics in 4 different colours although the resolution of the screen drops from 176 by 160 to 88 by 160.

In the high resolution mode there is one bit in the VIC's memory which corresponds to each point on the screen and each point is one dot wide. When you use the SET instruction you are SETting this bit and hence producing a dot on the screen. In the multicolour mode however there are two bits corresponding to each point on the screen and each point is two dots wide (hence the resolution is 88 by 160). While one bit in the high resolution mode can only be 0 or 1 (SET or RESET) two bits in the multicolour mode can be 0 and 0, 0 and 1, 1 and 0 or 1 and 1 and depending on the value of these two bits one of four colours is selected for the display of the corresponding dot.

The meanings of the four patterns are as follows:

0 and 0 display the point in the screen colour
0 and 1 display the point in the border colour
1 and 0 display the point in the foreground colour
1 and 1 display the point in the auxiliary colour

where the screen and border colours are set up with the GROUND and BORDER statements, the foreground colour is the colour you specified in the last CLEAR or SCREEN statement (the number used in the CLEAR statement is the number for the colour you require plus 8, e.g. 1+8=9 for BLACK foreground in the multicolour mode) and finally the auxiliary colour is one of the sixteen colours available for the screen colour produced with the following command or statement:

```
POKE 36878,(PEEK(36878) AND 15) OR (C*16)
```

where C is the number for the colour that you want, 0 for BLACK, 1 for WHITE and so on to 15 for LIGHT YELLOW.

As you can see you use the SET and RESET statements to produce dots of any of the four colours selected on the screen. To produce a dot of the screen colour at (X,Y) (note that X must be in the range 0 to 87) you must RESET (turn to 0) the two points controlling the colour of that point using:

```
RESET (2*X,Y):RESET (2*X+1,Y)
or LINE (2*X,Y)-(2*X+1,Y),R
```

For a point of the border colour use:

```
RESET (2*X,Y):SET (2*X+1,Y)
```

For a point of the foreground colour use:

```
SET (2*X,Y):RESET (2*X+1,Y)
```

while for a dot in the auxiliary colour use:

```
SET (2*X,Y):SET (2*X+1,Y)
or LINE (2*X,Y)-(2*X+1,Y)
```

You will be able to use LINE, DRAW, FILL and PUT as normal but the colour results may not be what you expect. For example, horizontal SET LINEs which start at an even x value and finish at an odd one will produce lines in the auxiliary colour but vertical SET LINEs will either come out in the foreground colour or the frame colour depending on whether the x coordinate of the line is even or odd.

Program 10

```
500 REM RETURN THE LIMITS OF THE DRAWING IN X0,X1 AND Y0,Y1
510 X0=0:X1=175:Y0=0:Y1=159
520 PUT DUMMY%(0) < (X0,0)-(X0,159)
530 IF ((PEEK(229)AND1)=0) THEN X0=X0+1:IF X0<>176 THEN 520
540 IF (X0=176) THEN X0=0:X1=0:Y1=0:GOTO 610
550 PUT DUMMY%(0) < (X1,0)-(X1,159)
560 IF ((PEEK(229) AND 1)=0) THEN X1=X1-1:GOTO 550
570 PUT DUMMY%(0) < (0,Y0)-(175,Y0)
580 IF ((PEEK(229) AND 1)=0) THEN Y0=Y0+1:GOTO 570
590 PUT DUMMY%(0) < (0,Y1)-(175,Y1)
600 IF ((PEEK(229) AND 1)=0) THEN Y1=Y1-1:GOTO 590
610 RETURN
```





This subroutine returns the limits of the drawing currently on the screen (you need not be displaying the drawing at the time) in the variables X0, X1, Y0 and Y1. Thus a statement:

```
LINE (X0,Y0)-(X1,Y1),S,B
```

draws a box around the drawing on the screen. This example illustrates the use of the collision detection facility built into PUT.

Program 11

```
20 REM ONLY FOR VICS WITH MORE THAN 3K EXPANSION
30 FOR I=0 TO 2047
40   POKE 1024+I,PEEK(32768+I)
50 NEXT I:REM TRANSFER UPPER CASE FONT INTO RAM
60 REM REDEFINE @ CHARACTER (FIRST IN CHARACTER SET)
70 FOR I=1024 TO 1031
80   READ BM:POKE I,BM
90 NEXT I
100 FONT OWN,SPACING (9,0),AT (1024)
110 CLEAR 7:GROUND 2:BORDER 2
120 DRAW "BM(100,100):"
130 WRITE 3,"@@@@@@"
140 DATA 60,66,165,129,165,153,66,60
150 REM DATA FROM PROGRAMMER'S REFERENCE GUIDE
```

This example illustrates the use of the FONT and WRITE statements. The program first creates a copy of the upper case font in RAM, starting at 1024 (this means that with more than 3K expansion the 3K expansion memory must be in place). The @ character is then redefined as a face using data from the Programmer's Reference Guide (each character definition occupies 8 bytes).

To find the location for a character contained in the string A$ add the address of the start of the font (1024 in the example above) to the result of this small routine:

```
200 A=ASC(A$):RESULT=A AND 63
210 IF (A>127) THEN RESULT=RESULT+64
220 IF RVS THEN RESULT=RESULT+128
230 RESULT=RESULT*8:RETURN
```

where RVS is true (i.e. -1) if a reverse field version of the character is wanted otherwise 0. You can then POKE 8 bytes of data at the start of font address plus RESULT and display the character with a WRITE ,A$.

The FONT statement allows you to include any letters after the commas providing they do not include any BASIC tokens. Thus AT is used rather than LOCATION since LOCATION includes the BASIC keyword ON.

Program 12

```
20 CIRCLE (80,80),50,,,0.125
30 LINE (30,80)-(130,80)
40 LINE (80,30)-(80,130)
50 LINE (44,44)-(116,116)
```

```
60 LINE (116,44)-(44,116)
70 CIRCLE (105,65),50,,,,0.124
80 LINE (105,65)-(155,65)
90 LINE (105,65)-(141,29)
100 FILL (126,64)
110 DRAW "BM(0,152):":FONT LOWER
120 WRITE 3,"Using VICKIT III is a"
130 WRITE ,"(cd)    (rvs)piece(cu)(off) of (rvs)(cd)cake"
```

cd) stands for cursor down, (rvs) for reverse on, (cu) for cursor up, and off) for reverse off.





## Changes from VICKIT II

1 All the VICKIT II non-graphics instructions, AUTO, DUMP etc, are no longer available. If you wish to have the best of both worlds you will need VICKIT III, VICKIT (or VICKIT II) and a STACK ROM SWITCH BOARD. You can then use the SWITCH statement or command included in VICKIT III to help switch between the two EPROMs.

2 Normal VIC BASIC statements will be interpreted up to 30% faster than with VICKIT II provided that the statements begin with a token. This means that assignment statements will not be speeded up unless they are preceded by LET.

3 A slight bug concerning the priorities of the Invert colour and the Blank modifier in DRAW has been corrected. Previously if the colour in a DRAW statement had been set to Invert lines that should have been blank due to the presence of the Blank modifier would come out as Inverted. The solution to this for VICKIT II is to return to normal colour after each Inverted line.

4 The statements SCREEN, COPY, FONT, WRITE, BORDER, GROUND, OFF and SWITCH have been added. The CIRCLE statement is included on the EPROM.

5 The use of page zero RAM has been altered so that cassette and RS-232 handling can be carried out at the same time as VICKIT III statements.

6 The FILL algorithm has been altered so that it will cope with most, if not all, areas on the screen. An area calculation facility has also been added.

7 Error messages are still intercepted by VICKIT III but the TEXT UPPER statement is only executed if the VIC chip currently has its double character bit set.

8 The case change (shift and the Commodore key) has been disabled while in the graphics mode.

9 Attempting to use CLEAR or SCREEN without having performed a GRAPHICS statement will result in a ?OUT OF MEMORY error.

10 Using SYS(11*4096+3) (or SYS(45059)) initialises VICKIT III without printing the copyright message. This is intended for use inside a program.

11 Provided VICKIT III is initialised as the last add-on and there is no conflict between the memory requirements (unlikely) VICKIT III will co-exist with other add-ons.

## Information for machine code programmers

This section is intended to be of use to those people who are interested in linking machine code routines to VICKIT III. A number of useful subroutine entry points are given, together with a description of the operation of the GRAPHICS, PUT and SWITCH instructions and details of memory used, both in page zero and absolute memory.

Where absolute addresses are given you are warned that future issues in the VICKIT series are not guaranteed to maintain compatibility.

### GRAPHICS

The operation of the GRAPHICS instruction varies depending on the amount of memory the VIC being used contains. The general idea is to locate the screen memory at locations $1E00 to $1FFF (any numbers preceded by $ are in hexadecimal) and the character bit map memory at locations $1000 to $1DFF. Thus a VIC with only the 3K memory expansion needs no alteration to the program since there is no clash between program and graphics memory. A VIC with greater than 3K memory expansion starts BASIC program storage at $1200 which would clash with the character bit map memory. A GRAPHICS statement in a greater than 3K expansion VIC therefore moves the entire program up to start at $2000 (altering all the necessary page zero pointers on the way). A check is made for sufficient memory and if not enough exists the ?OUT OF MEMORY error is given. Before rejoining the BASIC program a BASIC CLR operation is performed.

### PUT

The operation of the PUT instruction is of interest to anyone wanting to take slices through a BASIC array when PUTting to the screen. PUT treats array storage simply as a contiguous block of memory so that providing you only PUT 'whole' arrays there is no problem. When taking slices through the array, as in example programs 6 and 8, you will need to know about the way that the VIC stores arrays to ensure that you get the correct results.

The VIC stores arrays so that the first dimension varies most rapidly so an array dimensioned as SQ(2,1) would be stored in the order SQ(0,0), SQ(1,0), SQ(2,0), SQ(0,1), SQ(1,1) and SQ(2,1). Since VICKIT III stores and loads from the array in a sequential manner you can see that the information for any area which is accessed separately from the main part of the array should be accessed by columns. Thus in example program 6 the array was dimensioned as CHAR%(3,26) so that a single character is stored in array elements with row indices 0, 1, 2 and 3 and can be referenced easily.

The order in which points are stored or loaded given an area in the PUT instruction such as (x1,y1)-(x2,y2) depends on the relative values of the coordinates but can be described best as follows:

1 Start at (x1,y1) and progress to (x2,y1)

2 When a horizontal 'scan' across the area in interest has been performed check the values of y1 and y2. If they are not equal then increment y1 by 1 if y1 < y2 or decrement y1 by 1 if y1 <= y2 and then repeat step 1 again.





3 If y1 equals y2 terminate the PUT instruction.

### SWITCH

The program moved into the cassette buffer by the SWITCH statement or command occupies locations $033C to $034B inclusive. SYS(836) will, after a SWITCH, return the switched out block into the VIC's memory and reinitialise VICKIT III.

#### Page zero usage

| Location name | address | uses |
|---|---|---|
| PLOTOK | $D9 | Set to $00 if ADDRSS carried out an address calculation correctly. Bit 7 set if Y was out of range, bit 6 set if X was out of range. |
| STRTX | $DA | Start point for line drawn by DRLINE (x coordinate). |
| STRTY | $DB | Start point for line drawn by DRLINE (y coordinate). |
| LASTX | $DC | End point for line drawn by DRLINE (x coordinate). |
| LASTY. | $DD | End point for line drawn by DRLINE (y coordinate). |
| LINCOL | $E3 | Line colour, $00 for a SET line, $80 for a RESET line and bit 0 set for an INVERTed line. |
| COLLID | $E5 | Used in PUT to hold the results of the collision detection routine. |
| SVCUX | $EA | During a WRITE holds the x co-ordinate of the first character written by this WRITE, i.e. CUPOSX on entry. |
| SVCUY | $EB | During a WRITE holds the y co-ordinate of the first character written by this WRITE, i.e. CUPOSY on entry. |

All locations between $D9 and $EB are used, the ones listed are the most useful.

#### Useful absolute locations

| Location name | address | uses |
|---|---|---|
| | $02A1-$02AF | A range of locations used mainly in CIRCLE. |
| INDERR | $02B0-$02B1 | Address of original error vector copied from $0300-$0301 |
| STNBLK | $02B2-$02B3 | Address of original start new BASIC statement vector copied from $0308-$0309 |
| CUPOSX | $0334 | Holds the current position (x coordinate). |
| CUPOSY | $0335 | Holds the current position (y coordinate). |
| FNTADL | $0336 | Holds the low byte of the address of the |

first byte of the current font used in WRITE.

FNTADH $0337 Holds the high byte of the address of the first byte of the current font used in WRITE.

SPACEX $0338 Holds the horizontal spacing to be used in any WRITE statements. Can be two's complement, e.g. $FC for -4.

SPACEY $0339 Holds the vertical spacing as for SPACEX.

ADDRSS $B156 ($A156)
Enter with (X) and (Y) being the x and y coordinates of a point. Exit with ($5F),Y being the address of the byte in memory concerned and A containing a mask (1 bit only set) to the bit within that byte which is mapped to the point required. PLOTOK is set up as described above. If either or both of the coordinates are out of range they are replaced by 175 (for x) and 159 (for y).

MASK $B198 ($A198)
Start of a table of single bit masks, $80, $40 etc.

GTPNT $B238 ($A238)
Enter with (X) and (Y) as for ADDRSS. Exit with minus flag set if coordinates were off the screen, (X)=$81 if point was set, $00 if it was clear.

CLEAR+$03 $B316 ($A136)
Enter with (X) being colour code used in the CLEAR statement, e.g. 1 for BLACK. The screen is cleared and the VIC enters high resolution graphics mode after clearing the high resolution screen.

DOCLRS-$01 $B332 ($A332)
Enter with (X) being colour code used in the SCREEN statement. The text screen is cleared and the VIC enters high resolution graphics mode without clearing the high resolution screen.

TEXTL $B398 ($A398)
Restores screen to text with lower case option set.

TEXTU $B39B ($A39B)
As for TEXTL but restore with graphic option set.

SETPT $B3C5 ($A3C5)
Assumes ADDRSS executed immediately before. SETs the point calculated by ADDRSS.

RESTPT $B3CC ($A3CC)
As for SETPT but RESETs the point.

INVERP $B3D6 ($A3D6)
As for SETPT and RESTPT but INVERTs the point.

DRLINE $B468 ($A468)
Assumes STRTX, STRTY, LASTX, LASTY and





LINCOL have been set up. Draws a line of the specified colour between (STRTX,STRTY) and (LASTX,LASTY).

FORM $B919 ($A919)
Main body of PUT. On entry $E3 must hold $80, $47, $48 give the starting address of the memory location involved in the transfer, STRTX/STRTY and LASTX/LASTY must define the rectangular area on the screen, COLLID must hold $00 if collision detection is to be valid, $E6 must hold $40 if direction of PUT is to memory, $00 if to screen, $DE must hold the rule number to be used and $E7, $E8 must hold in low/high byte order, the maximum number of bytes to be used from the memory address in $47, $48 onwards.

WRTCH $BE71 ($AE71)
Enter with (A) being a character to be written to the high resolution screen as in WRITE. FNTADL/FNTADH, SPACEX/SPACEY and CUPOSX/CUPOSY must have been set up correctly. SVCUX/SVCUY must hold the co-ordinates of the first character written in order for return characters and off screen writing to be carried out correctly.

GROUND+$03 $BFC9 ($AFC9)
Enter with (X) being colour code used in GROUND statement, e.g. 1 for BLACK, 16 for LIGHT YELLOW. The screen colour is set to the colour specified.

BORDER+$0D $BFB3 ($AFB3)
Enter with (X) being colour code used in BORDER statement. The border colour is set to the colour specified.

OFF $BFE0 ($AFE0)
Restore the error and get the new BASIC statement links from INDERR and STNBLK, i.e. switch off VICKIT III.