VIC KIT II USERS GUIDE

STACK
COMPUTER SERVICES LTD

VICKIT II User's manual

General description

VICKIT II is a 4k (4096) byte EPROM (Erasable Programmable Read Only Memory) which provides 21 new commands and statements for the VIC. Nine of these are used when entering and debugging a program while the other twelve greatly simplify the programming needed to produce high resolution graphics on the VIC.

VICKIT II makes a distinction between commands, which are typed into the VIC without a line number, and statements, which are always inserted in a program. While some VIC BASIC commands can be used as statements or commands (CLR, NEW, PRINT etc), VICKIT II commands and statements will only be recognised in their correct place. Thus using AUTO in a program or GRAPHICS as a command will give a ?SYNTAX error.

The sections on VICKIT II commands and statements can be read separately since they do not depend on one another to any great degree. However, you should have read about errors and the LIST command in the next section before starting to use VICKIT II statements. You are likely to learn most from your mistakes and you should use the facilities provided by VICKIT II to make the process as painless as possible.

Throughout the manual a warning sign (WARNING) is used to indicate a section of particular importance, often a section where misuse of VICKIT II facilities could result in damage to your program.

Carefully read the section on installing VICKIT II before installing it in your system. Once installed and enabled you can start to work your way through the commands and statements. If you have some experience with the BASIC Programmer's Toolkit for the PET you need only read the section giving the differences between VICKIT II commands and the Toolkit commands before starting to use the high resolution graphics statements.

WARNING

VICKIT II uses a number of memory locations that are used by the VIC for cassette and RS-232 handling. Neither of these features will work correctly if used at the same time as VICKIT II commands or statements. However, as long as the cassette/RS-232 operations are started and finished BETWEEN VICKIT II commands/statements there should be no problem since VICKIT II leaves no permanent information in the relevant locations.

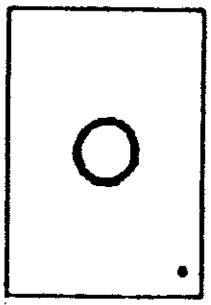
Installing, enabling and disabling VICKIT II

Installing VICKIT II

Locate the empty integrated circuit socket on your memory expansion board where VICKIT II is to be placed. If you have more than one socket it should be the one located at addresses $B000 (45056) to $BFFF (49151). Refer to the documentation supplied with the memory board if you are in any doubt.

Touch any safely earthed piece of metal to discharge any static electrical charge from your body. Carefully unpack the chip and examine its pins. If any pins are bent or out of alignment straighten them using narrow pointed pliers VERY GENTLY. Integrated circuit (chip) pins are fragile and will fracture very easily with bending. All VICKIT II chips are tested using zero insertion force sockets before dispatch and are packed in rigid pin-protecting tubes.

The VICKIT II EPROM



This dot marks pin 1

The chip MUST be correctly orientated in its socket at $B000. Placing it in the wrong socket will not harm the chip but it will not work. However, REVERSAL of the chip in its socket will probably DESTROY the chip instantly by breaking down the stored program in the chip or damaging its internal circuitry.

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 who has experience of chip insertion.

When VICKIT II has been installed turn on the VIC-20. If the CBM BASIC message does not appear TURN OFF AT ONCE and recheck the steps that you took during installation.

Enabling VICKIT II

You enable VICKIT II 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 check both your installation of the VICKIT II and the enabling command you gave. If all seems correct check that the socket you have used for the VICKIT II is the correct one.

If the VICKIT II still does not work you should contact the supplier of the chip for assistance.

Disabling VICKIT II

Having just enabled VICKIT II 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 II slows down the execution of normal BASIC programs so that you might want to switch it off to speed up a section of programming. For whatever reason the following command (it can also be used as a statement inside a program) will disable VICKIT II with no harmful effects:

```
SYS(58459)
```

The READY prompt will now reappear if you used the SYS as a command. If you incorporated the SYS in your program then the program simply continues to run as normal. Both VICKIT II commands and statements will now be reported as ?SYNTAX errors.

Changes to LIST and error handling with VICKIT II enabled

When VICKIT II has been enabled the LIST command is entered precisely as before but requires you to press CRSR down or STOP after each line is listed. This means that you can LIST the program, or section of it, as before but halt the listing if you see a line of interest. Press CRSR down and the listing continues, press STOP and the VIC returns to normal. All VICKIT II commands which print to the screen have this facility built in to avoid the annoying occasions when the line you are interested in has just scrolled off the screen and you have to re-enter the LIST command. Should you want to LIST at full speed, perhaps to a printer, simply precede the LIST command with a colon.

You will also notice that when you make an error the screen is cleared and the error message printed at the top of the screen. This is so that when an error occurs during high resolution graphics plotting the error message becomes visible. Should you want to disable this facility a method of doing this is given in the section of this manual on error handling.

Quick reference guide to VICKIT II commands and statements

For each command and statement the name of the command or statement is given (this is the word you type to tell VICKIT II that you want that particular command or statement) followed by the possible forms of the command/statement.

VICKIT II commands

AUTO

```
AUTO
AUTO n
AUTO n,m
```

where n is the starting line number and m is the step line number. If n or m is not specified they default to 10.

DELETE

```
DELETE n-
DELETE -m
DELETE n-m
```

where n is the lowest line number to be deleted and m is the highest line number to be deleted. If n is not given it defaults to the first line number in the program, if m is not given it defaults to the last line number in the program.

DUMP

```
DUMP
```

FIND

```
FIND c
FIND c,n-
FIND c,-m
FIND c,n-m
```

where c is a string either with or without quotes, e.g. "DATE" or PRINT. n is the lowest number to start searching from, m the highest. If n is not given it defaults to the first line number in the program, if m is not given it defaults to the last line number in the program.

HELP

```
HELP
```

OFF

```
OFF
```

RENUMBER

```
RENUMBER
RENUMBER n
RENUMBER n,m
RENUMBER n,m,p-
RENUMBER n,m,-q
RENUMBER n,m,p-q
```

where n is the starting line number for the renumbered program, m is the line number step, p is the first line number to be renumbered and q is the last line number to be renumbered. If n or m are not given they default to 10, if p is not given it defaults to the first line number in the program and if q is not given it defaults to the last line number in the program.

STEP	STEP
TRACE	TRACE

VICKIT II instructions

GRAPHICS GRAPHICS

CLEAR CLEAR n

where n is the number of the colour required, 1 for BLACK etc.

TEXT TEXT LOWER
TEXT UPPER

LOWER and UPPER may be abbreviated to L and U.

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),variable

where (x,y) is the point whose status is to be found and variable 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 is either S(ET), R(ESET) or I(NVERT) and if omitted has a default of S(ET). o is either B(OX) or F(ILL).

DRAW DRAW stringexpression

where stringexpression is a string containing a sequence of DRAW instructions terminated by colons or semi-colons. Each DRAW instruction can be preceded by B or N and can be:

M Move instruction, either absolute: M(x,y): or relative: 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. R50,U50:
X	to eXecute a string. X is followed by the name of the string, e.g. X S$:
C	to select a Colour for subsequent drawing, either S(ET), R(ESET) or I(INVERT), e.g. CS:
T	to select a Turn for subsequent U, D, L or R instructions, either 0, 1, 2 or 3, e.g. T2:
S	to select a scale factor for subsequent U, D, L or R instructions, either 0, 1, 2, 3, 4, 5, 6 or 7, e.g. S5:

FILL

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

where (x,y) is the point at which to start filling and c is the colour at which to stop filling, either S(ET) or R(ESET). If c is omitted its default is S(ET).

PUT

```
PUT arrayelement direction (x1,y1)-(x2,y2)
PUT arrayelement direction (x1,y1)-(x2,y2),rule
```

where arrayelement is the name of an array element, either integer or real, e.g. CHAR%(0,0), direction is either > or <, (x1,y1)-(x2,y2) gives a pair of opposite corners of the area on the screen in question and rule is a number in the range 0 to 15 inclusive. If rule is omitted it is taken to be 3 if direction is >, 5 if direction is <.

CIRCLE

```
CIRCLE (x,y),radius
CIRCLE (x,y),radius,c
CIRCLE (x,y),radius,c,height/width ratio
CIRCLE (x,y),radius,c,height/width ratio,start point
CIRCLE (x,y),radius,c,height/width ratio,start point,finish point
```

where (x,y) is the centre point of the circle, radius is the radius of the circle, c gives the colour of the circle S(ET), R(ESET) or I(NVERT), height/width ratio gives the ratio of the height of the circle to the width (the width will always equal the radius), start point gives the proportion of the way round the circle to start drawing (CIRCLE starts at 3 o'clock and goes clockwise) and end point gives the proportion of the way round the circle to finish drawing.

If c is omitted it is taken to be S(ET), if height/width ratio is omitted it is taken to be 1, if start point is omitted it is taken to be 0 and if finish point is omitted it is taken to be 1.

Using VICKIT II commands to enter and debug programs

VICKIT II is used in three stages of program development, while actually entering the program, during testing of the program and when tidying up the program to its final version.

This section of the manual illustrates these three stages by leading you through the development of a typical program using VICKIT II. When a new VICKIT II command is used it will be explained in detail with examples of its use on the developing program.

The program illustrated here is one to find the day of the week on which a given date fell and works for all dates after 1752, when there was a change in the calendar. The program comes from the Independent PET Users' Group magazine, volume 3 number 6.

You will need to enter the program, starting at line 10, with the second line number being 20, the third 30 and so on. To avoid entering the line number of each line (possibly getting it wrong and deleting a needed line) use the AUTO command.

AUTO

There are three forms of the AUTO command, the first specifying just the line number with which you wish to start, the second specifying both the starting line number and the step between line numbers (in the first case AUTO uses 10 as the step) and the third specifying neither the start line number nor the step (AUTO uses 10 for both). VICKIT II then generates the next line number for you whenever you press RETURN, starting after the AUTO command.

Before proceeding give the command:

```
NEW
```

To use the first form enter the command:

```
AUTO <starting line number>
```

For example, to start generating line numbers for our program beginning at 10 with a step of 10 between lines simply enter the command:

```
AUTO 10
```

VICKIT II will respond with:

```
10 █
```

where █ represents the flashing cursor. You can now enter the first line of the program:

```
DIM DAY$(7):DAY$(1)="SUNDAY":DAY$(2)="MONDAY"
```

The VIC will take line 10 as the first line of your program and

VICKIT II will print the next line number for you. This will be line 20 since we used the first form of the AUTO command and VICKIT II takes 10 as the step between lines.

Now enter the second line of the program as follows:

```
        DAY$(3)="TUESDAY":DAY$(4)="WEDNESDAY":DAY$(5)="THURSDAY"
```

Line 20 will be taken in by the VIC and VICKIT II will print the next line number for you, line 30. You could now go on and enter the rest of the program but first let's look at AUTO in more detail.

First of all, to stop AUTO operating press RETURN only in response to a line number. Pressing RETURN when VICKIT II prints 30 will not display line number 40 but will switch AUTO off. Since typing a line number followed immediately by RETURN is the standard BASIC way of deleting a line with that number we need some other way to turn AUTO off if there is the slightest chance of deleting a wanted line and this other way is to use the DEL key to delete the line number BEFORE pressing RETURN.

The second form of the AUTO command can be demonstrated by typing in some lines of rubbish starting at line 30 and on lines 35, 40 and 45. The AUTO command to do this has the following form:

```
      AUTO 30,5
```

VICKIT II will respond with:

```
      30
```

to which you should enter anything you like (we will be removing these lines of rubbish in the next section). Pressing RETURN after entering line 30 will produce the next line number, line 35. Again type in some rubbish, press RETURN and continue until VICKIT II prompts with 50. Now simply press RETURN and turn off AUTO. You now have lines 30, 35, 40 and 45 which we do not want in our final program. We could delete them line by line in the standard BASIC manner but using VICKIT II's DELETE command makes things much easier.

The third form of AUTO has the form:

```
      AUTO
```

which works in exactly the same way as the form:

```
      AUTO 10,10
```

i.e. a starting line number of 10 and a step between lines of 10.

)ELETE

The DELETE command has a form, or rather a number of forms, rather like the BASIC LIST command for listing a section of program. If you wanted to LIST the unwanted lines in the program you would enter the command:

```
      LIST 3Ø-45
or    LIST 3Ø-
```

since the unwanted lines go from line 3Ø to the end of the program. To DELETE these lines give the command:

```
      DELETE 3Ø-45
or    DELETE 3Ø-
```

The unwanted lines will then have disappeared. Note that you can only use the second form in this case because the lines to be DELETEd run from line 3Ø to the end of the program. Had there been lines after line 45 that you wanted to preserve the first form would be needed.

You can also DELETE lines from the beginning of the program to a certain line number, e.g.:

```
      DELETE -2Ø
```

If you gave this command go back and insert lines 1Ø and 2Ø again.

/ARNING

The form of DELETE is exactly like that of LIST with two exceptions: DELETE followed by just one line number will DELETE from that line to the end of the program (typing only the line number and pressing RETURN is easier) and entering DELETE on its own will not DELETE the entire program since there is a perfectly good BASIC command (NEW) for this.

/ARNING

DELETE N-M where N is greater than M, e.g. DELETE 2ØØ-1ØØ, will corrupt your program. Always keep a backup copy of your program before using DELETE.

)ack to entering the test program again. Restart AUTO at line 3Ø with a .ine number step of 1Ø using the command:

```
      AUTO 3Ø,1Ø
or    AUTO 3Ø
```

:nter the following lines exactly as written, although not of course the ine numbers since AUTO does that job for you:

```
3Ø DAY$(6)="FRIDAY":DAY$(7)="SATURDAY":D=Ø:M=Ø:Y=Ø
4Ø INPUT "DATE EG 2Ø,11,1981";DAY,MNTH,YEAR:IF YEAR>=1753
   THEN 6Ø
5Ø PRINT "DATE MUST NOT BE PRIORTO 1753":GOTO 4Ø
```

```
60 K=INT(0.6+(1/MNTH)):L=YEAR-K:P=K/100
70 Z=INT(13*(MNTH+12*K+1)/5)+INT((5*L)/4)-
   INT(P)+INT(P/4)+DAY-1
80 Z=Z-(7*INT(Z/7))+1
90 PRINT "THE DATE";D;M;Y:PRINT "WAS OR WILL BE":PRINT
   "A ";DAY$(Z)
100 PRINT "ANOTHER DATE (YES OR":PRINT "NO)";:INPUT ANS$
110 IF ANS$="YES" THEN PRINT:GOTO 40
120 IF ANS$="NO" THEN 150:REM EXIT
130 PRINT "DO IT PROPERLY":GOTO 100
140 PRINT "GOODBYE":END
150
```

Press RETURN only for line 150 to stop AUTO from generating any more line numbers.

Now RUN the program and see what happens. Enter a few dates whose days are known to you and you should find a number of errors. The most obvious one is that the program does not work since it does not tell you the correct day in all cases. The second error is that the date is printed on the screen as 0, 0 and 0 rather than the date that you typed in. If you have added some errors of your own you may find the program not even giving an answer.

In programming there are two main sorts of errors. The first is an error in the logic behind the statements in the program which means that although the program RUNs the answers produced are not the correct ones. The second sort is an error in the actual program statement which the BASIC interpreter can detect. In this case a helpful error message is printed on the screen giving not only the type of error but the line in which it occurred. In general the first sort of error is the harder to cure but VICKIT II provides three commands to help you while with the second sort of error VICKIT II only provides one command. The sample program has both sorts of errors and you will use all the relevant facilities of VICKIT II to track them down.

To find the first sort of error the commands DUMP, STEP and TRACE are used and we shall look at the program first using the DUMP command. Wait until you are asked by the program for the response YES or NO in line 100 before pressing the RIGHT HAND SHIFT and RUN/STOP keys to break out of the program and get the READY prompt back again.

DUMP

To display the values of all variables used by the program (all simple variables that is, array variables are not DUMPed by VICKIT II) give the command:

```
DUMP
```

You will find that the variables used in the program so far will be listed, one to a line, on the screen. After each one is listed you can either press the STOP key to stop the DUMP or press the CRSR down key to continue with the listing. With the simple program above you will find the following being printed:

```
DA =
MN =
YE =
K  =
L  =
P  =
Z  =
AN$="LOAD"
```

Variables except ANS$ will have values that depend on what answers you gave to the questions. Note that only the first two characters of variable names are given in the DUMP, e.g. DA not DAY and that string variables are shown with their value inside inverted commas so that cursor and colour control characters will show up properly and can be altered using the screen editing on the VIC. All types (real, integer and string) of variables are DUMPed by VICKIT II and you can alter any values on the screen as if they were program lines. Thus a program can be stopped halfway through its RUN, its variables DUMPed and examined, possibly altered and then the RUN CONTinued.

ooking at the list of DUMPed variables you can see that DA, MN and YE ave the values that you last gave them in response to the INPUT statement n line 40 so why are they printed as 0, 0 and 0? If we LIST line 90 don't forget to press the CRSR down key after the line is printed) you an see that D;M;Y rather than DA;MN;YE was printed.

lter line 90 to print DAY;MNTH;YEAR and RUN the program again. You should ind that the date being printed on the screen is correct but that the day f the week being given is probably still wrong.

he last error in the actual calculation part of the program is of the ort you would be unlikely to find using VICKIT II. The statement:

```
P=K/100
```

hould in fact be:

```
P=L/100
```

ince the statement is correct BASIC it will not cause an error so that he only way that you would discover the error is if you knew the likely alue of P. In fact it is the number of centuries from 0 to the date nput, e.g. for 1952 P would be 19. DUMPing the variables would help you iscover that P was set to a value less than 1 and give a clue that the roblem was with a statement which set P to a value, i.e. a statement eginning P= (or LET P=). To find each statement in the program which sets to a value you can use the fourth of VICKIT II's commands: FIND.

IND

Assume that you don't know on which line the statement P=K/100 occurrs. To FIND which line it is on give the VICKIT II command:

```
FIND P=
```

Note that there MUST be a space after FIND.

VICKIT II will then list all the lines in the program which contain the section of programming P=. After each line is listed you must remember to press the CRSR down key in order to carry on or STOP to stop with the lines currently displayed on the screen. The FIND command is of course much more useful in larger programs. For example, suppose that the program you are writing requires a variable to be defined to hold a temporary value. Can you use the variable DT? You could look through the entire program for any uses of the variable DT, or if you have been systematic you would have written down each variable name as you used it. With VICKIT II you have a third option, which allows you to devote your time to programming, rather than to housekeeping. Simply give the VICKIT II command:

```
      FIND DT
```

and any lines of the BASIC program containing DT will be listed on the screen. If no lines are listed then you can use the variable DT freely.

You can also restrict the area of your program that VICKIT II will search when using FIND. After giving the section of programming you are interested in type a comma followed by a range of line numbers in the same form as you use for LIST and DELETE. Thus to FIND any sections of programming which use P= in the lines 10 to 150 simply type the following command:

```
      FIND P=,10-
or    FIND P=,-150
or    FIND P=,10-150
```

So far we have just looked for BASIC statements, or rather parts of them. There is another form of FIND which will look for strings in your BASIC program. Strings will occur in a number of place, most obviously in quotes, e.g. "DATE EG" in line 40. However they can also occur in REM and DATA statements for the following reason: when you type in a line of BASIC the VIC normally converts BASIC keywords and operators, e.g. IF, THEN, PRINT, =, * and +, into single characters called tokens. For example when the word PRINT is converted into a token, the five locations that would be used to store PRINT as single characters are reduced to one holding the token for PRINT. Tokens also speed up the execution of your program. However when you type PRINT within inverted commas, or in a DATA or REM statement, this 'crunching' into tokens is not performed, and PRINT will be stored as five separate characters.

Because of tokens you can see that there could be two forms of the character P=. The first would be when P= has appeared as a BASIC statement and the = has been replaced by its token. The second will be when P= has appeared within quotes, or in a DATA or REM statement, when the = will not have been replaced by its token.

So far we have been FINDing the first form but VICKIT II will also FIND the second provided that you enclose the string to be search for within quotes, e.g.:

```
        FIND "P="
```

This crunching into tokens means that you cannot FIND parts of BASIC keywords, e.g.:

```
        FIND TH
```

will not FIND all the lines containing THEN but only lines 2Ø, 4Ø, 6Ø, 7Ø, 9Ø (if you have corrected it) and 1ØØ.

If you are going to restrict the range of lines to be examined when using the second form of FIND you must remember to place both the comma and the line numbers after the closing quotes.

ÍARNING

The line:

```
        IFX>ØTHEN2Ø
```

is NOT the same as the line:

```
        IF X>Ø THEN 2Ø
```

since spaces are significant to FIND.

Experiment with FIND, in both forms, until you understand why it lists some line and not others depending on what you type and whether you use quotes or not.

laving altered the error in line 9Ø you can proceed to check the rest of :he program. You will probably have answered YES to the question in line .ØØ already, so now enter NO and press RETURN. You should find that the ›rogram clears the screen, prints an error message and stops. Now you can ıse the fifth command provided by VICKIT II: HELP.

lELP

The HELP command is only of use when you have just had an error reported by the VIC. Type the command:

```
        HELP
```

The line in error will be listed on the screen and the section of the line in error highlighted in reverse field. The HELP command must be the first thing executed after an error otherwise the information needed by VICKIT II may have been destroyed or altered.

You may find that he highlighted section of the line listed may be just before or after the section of the line in error but the information given by VICKIT II, together with that given by the VIC, should be enough to let you locate the error. If a BASIC keyword is in error then the entire keyword will be highlighted.

The sixth command that VICKIT II provides is really only useful when used in conjunction with STEP and TRACE so we ignore OFF for the moment and return to it later. The seventh command is RENUMBER:

RENUMBER

RENUMBER is the command which, as its name suggests, RENUMBERs the line numbers in your program. In order that the program still works it also RENUMBERs the destination line numbers in GOTO, GOSUB, LIST, RUN and IF...THEN statements.

There are 6 forms of the RENUMBER command ranging from a simple command to RENUMBER the entire program starting at line 10 with a step between line numbers of 10 to a version that lets you specify the range of line numbers to be RENUMBERed, the starting line number and the step between line numbers.

The simplest form is the command:

```
RENUMBER
```

which, since no range of line numbers has been given, RENUMBERs the entire program in memory, starting at line 10 (since no starting line number has been given) with a step between lines of 10 (since no step between line numbers has been given). Since the original program started at line 10 and had a step of 10 betwen line numbers there should be no change in the program, except that if you had not corrected the error in line 120 the line number to GOTO if ANS$ was "NO" would have become 0. If, when RENUMBERing a program or section of program, VICKIT II discovers a reference to a non-existent line number that line number is replaced by 0. You can use FIND to FIND any errors caused this way by giving the command:

```
FIND " 0"
```

although this will only work if you follow every GOTO, GOSUB or IF...THEN with a space (good practice anyway).

The second form of RENUMBER allows you to specify the starting line number. An example of this form is:

```
RENUMBER 100
```

The first line of t 110, the third 120 and so on.

WARNING

Using 0 as either the start line number or the step between line numbers will result in your program being corrupted.

The third form of RENUMBER allows you to specify both the starting line number and the step between line numbers. The step is separated from the starting line number by a comma, e.g.:

```
RENUMBER 200,5
```

will RENUMBER the entire program starting at line 200 with a step between lines of 5.

The last three forms are really variations of the same basic theme which allow you to specify the range of line numbers that RENUMBER works on. This means that RENUMBER can be made to RENUMBER sections of your program rather than the entire program. As with DELETE, LIST and FIND the range can be specified in one of three forms:

```
      <lowest line number>-<highest line number>
or    <lowest line number>-
or    -<highest line number>
```

the first form using lines from the first to the second number, the second from the first number to the end of the program and the third from the start of the program to the first number. All the numbers given are inclusive. The three forms of RENUMBER are illustrated by:

```
RENUMBER 1000,10,1000-1999
RENUMBER 1000,10,1000-
RENUMBER 1000,10,-1999
```

By specifying the range of line numbers that RENUMBER is to work with you can preserve the structure of your program, leaving subroutines to start at easily remembered lines for instance.

ARNING

These three forms of RENUMBER can be damaging to both your sanity and program if misused. For example, consider the program:

```
10 I=1
20 PRINT I
30 I=I+1:IF I<100 THEN 20
```

If we give the following command to VICKIT II:

```
RENUMBER 40,10,20-20
```

the result is:

```
10 I=1
40 PRINT I
30 I=I+1:IF I<100 THEN 40
```

which looks correct, if a little strange. In fact this program will not RUN, since the VIC will report a ?UNDEFINED STATEMENT error in line 30.

The moral of this is that you should be extremely careful when using the restricted range form of RENUMBER since no error checking is built in to the RENUMBER routine in VICKIT II. As with DELETE it is a good idea to save your program before you attempt this sort of RENUMBER on your program.

The final three commands (STEP, TRACE and OFF) are best dealt with in one fell swoop since they are so closely related. All three are used during the debugging of a program since TRACE and STEP allow you to monitor the progress of your program while it is actually RUNning by displaying the line numbers of the last five lines executed by the VIC. Thus you can see which way your program is 'going' by finding out which lines are being executed.

TRACE and STEP work in basically the same way and are started in much the same way. Decide which one of the commands fits the particular circumstances of your program and type its name as a command. When you next RUN the program the command selected will come into action. Once the program has been debugged the chosen option can be disabled by using the OFF command.

STEP

Typing:

```
STEP
```

starts STEP working when the program is next RUN. When the program is RUNing the numbers of the last five lines executed will be displayed in the top right hand corner of the screen. Before each statement is executed by the VIC VICKIT II will test the left hand SHIFT key. If it is pressed then the statement will be executed. If not then execution of any further statements will be stopped until either the left hand SHIFT key or the STOP key is pressed.

TRACE

Typing:

```
TRACE
```

starts TRACE working in exactly the same way as STEP except that the left hand SHIFT key now acts to slow down the rate at which the VIC executes the lines of your program rather than speed up the rate as with STEP.

OFF

Typing:

```
OFF
```

simply cancels whichever of the two commands STEP or TRACE was last selected. RUNning the program then proceeds as normal.

WARNING

Note that with both STEP and TRACE the line numbers are displayed in white on black in the top right hand corner of the screen. For certain colour combinations these numbers may become invisible. Also note that any INPUT statements which take effect in the top 5 lines of the screen may INPUT the line numbers there. Finally one of the side effects of STEP or TRACE is to make the STOP key become a little sluggish, so that it may have to be held down for longer than normal.

APPENDing programs on the VIC

Although there is no APPEND command provided by VICKIT II, the following procedure can be used to APPEND either from tape or from disk.

For those users unfamiliar with the Toolkit APPEND allows you build up a program by APPENDing sections of programming together. You will probably have noticed by now that some sections of programming are common to more than one program and you may have wondered if there was any way to avoid typing in the same sections of programming over and over again. Both APPEND on the Toolkit and the procedure below for the VIC provide that way for you.

The only restriction on this method of APPENDing BASIC programs is that the line numbers in the program you are APPENDing must be larger than the ones in the program to which you are APPENDing. Failure to observe this restriction will result in programs with duplicate lines and lines out of order.

The APPEND procedure

1 SAVE the program that you wish to APPEND to cassette or to disk with SAVE "name" or SAVE "name",8 respectively.

2 Enter the program to which you wish to APPEND the result of step 1 either by LOADing from cassette or disk or by typing in the program.

3 Give the following command:

```
PRINT PEEK(43),PEEK(44)
```

and make a note of the two numbers printed (they are referred to as A and B later on).

4 Give the following command:

```
PRINT 256*PEEK(46)+PEEK(45)-2
```

and make a note of the result (referred to as C in step 5).

5 Give the following commands:

```
POKE 43,C AND 255:POKE 44,INT(C/256):NEW
```

where C is the result you noted down in step 4.

6 Give the command:

```
   LOAD "name"
or LOAD "name",8
```

where name is the name that you used in step 1. Use the first form if you are LOADing from cassette, the second if you are LOADing from disk.

7 Give the following command:

```
PRINT FRE(0)
```

If you see a negative number displayed then the last program you APPENDed was too big for the VIC you are using. Give the command:

```
NEW
```

then step 8 of this procedure and your original program will be back.

8 Finally give the command:

```
POKE 43,A:POKE 44,B:CLR
```

where A and B are the numbers that you noted down in step 3.

You will now have a new program consisting of the original program with the APPENDed program added to its end. You can repeat this process as many times as you want subject only to the amount of memory on your VIC.

WARNING

Always SAVE a copy of your original program before APPENDing since this method alters information about your program needed by the VIC.

If you get a ?LOAD error in step 6 give the command:

```
NEW
```

followed by step 8. This restores the original program and should also be given if you mistakenly APPEND a program section with lines out of order.

Differences between VICKIT II and BASIC Programmer's Toolkit commands

AUTO

The default starting line number is 10 rather than 100.
Line numbers will only be generated by AUTO in the period between enabling AUTO and disabling it.
Screen editing will not produce unwanted line numbers and AUTO will remember neither the last line number it generated nor the last line number step used.

DELETE

DELETE n where n is a line number DELETEs from line n to the end of the program.
DELETE n-m where n is greater than m will corrupt the program.

DUMP

DUMPing will stop after each line printed until the CRSR down key is pressed.

FIND

A space MUST follow the command word FIND.
Enclosing a character string in quotes does not restrict the search to strings within quotes in the BASIC program. Thus the command:

```
FIND "A"
```

will FIND the statement A=B as well as the statement PRINT "ANOTHER". The quotes are needed only to protect the character string from being crunched to its token form.
Listing of lines stops after each line until the CRSR down key is pressed.

HELP

HELP is linked in to the error vector so that giving the HELP command after pressing STOP will not list the last line executed.

RENUMBER

The range of line numbers to be RENUMBERed can be restricted.
Setting either the start or step line numbers to 0 will result in VICKIT II corrupting the program.

STEP
TRACE
OFF

No known differences.

The abbreviated forms available to other VIC commands and the Toolkit commands, e.g. V SHIFT E for VERIFY or A SHIFT U for AUTO, are not available on VICKIT II.

Using VICKIT II statements to produce high resolution graphics

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

When using high resolution graphics on the VIC you should think of the screen as 176 points wide by 160 points high rather than 22 columns wide by 23 rows high with the origin (0,0) almost at the bottom left of the screen. Statements allow you to SET (make dark), RESET (make light) or INVERT (make light dark and dark light) any point on this screen and for some purposes, graph plotting etc, SET, RESET and INVERT will be useful commands. Usually however the more powerful commands LINE, DRAW and CIRCLE will be used since they do with single statements what would otherwise take tens or hundreds of SET, RESET and INVERT statements.

Once a shape or picture has been drawn on the screen it can be FILLed in or PUT in a BASIC array with a single BASIC 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 we discuss PUT this statement can even be used to produce characters on the high resolution screen as well as graphics. Thus graphs, shapes or pictures can be annotated easily and quickly, with either the normal VIC characters or a set of characters you have defined yourself.

To draw things on the VIC you must prepare both the VIC and your program for high resolution graphics and the first two statements we look at, GRAPHICS and CLEAR, allow this.

One final point needs to be made before you start using VICKIT II statements. With a statement like:

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

where ... represents any of the VICKIT II statements to be introduced in this section you should ALWAYS 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 which could otherwise mean the ... part of the statement being handled incorrectly.

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. For example:

```
100 XM=175:YM=159:GRAPHICS:PRINT XM,YM:END
```

will print 0 as the value for XM and YM since although they have been set to 175 and 159 respectively the GRAPHICS statement does a CLR and they become undefined. This effect also means that you cannot place GRAPHICS within a subroutine and use RETURN since GRAPHICS destroys the information that RETURN needs.

WARNING

If you do not include the GRAPHICS statement in your program and you then go ahead to use any other VICKIT II statements you are asking for trouble and your program will probably become corrupted.

If you are writing machine code for the VIC, using any technique that depends on where a BASIC program or its variables are stored or are just curious about the work that GRAPHICS does you should read the section of this manual entitled Information for machine code programmers.

GRAPHICS does not seem to do anything although its inclusion is essential in a high resolution graphics program. The second VICKIT II statement (CLEAR) is also essential and gives the first indication of VICKIT II actually doing something.

CLEAR

This statement has eight possible forms varying with which colour you want to draw lines and shapes. Follow the statement CLEAR with a number from 1 to 8. This number corresponds exactly to the colours shown on the keys 1 to 8 on the VIC keyboard. Thus use number 1 for BLACK lines, 2 for WHITE, 3 for RED and so on, up to 8 for YELLOW. Thus the form of the CLEAR statement is:

```
CLEAR c
```

where c is a number or variable with a value of 1 to 8 inclusive. If you use a number larger than 8 but less than 17 the statement will still be executed but the VIC graphics will then work in what is called MULTICOLOUR MODE (see Hints, tips and tricks).

You will notice that the screen appears to shrink in height and will immediately clear when the CLEAR statement is executed. The VIC's high resolution graphics are now working so that any attempt to PRINT anything to the screen will not produce the effect you want.

To make your program easy to understand it is a good idea to define variables with names that suggest the eight colours possible and values that give the correct colour when used with CLEAR, e.g.:

```
110 BLK=1:WHT=2:RED=3:CYN=4:PUR=5:GRN=6:BLU=7:YEL=8
```

and then use CLEAR with the correct variable, e.g.:

```
120 CLEAR BLU
```

to clear the high resolution graphics screen and set the colour of lines to be drawn to BLUe.

Having just made the VIC's high resolution graphics work the third VICKIT II statement (TEXT) stops the VIC working in high resolution graphics and returns it to the state where normal PRINT statements work on the screen. This automatically happens on an error once VICKIT II is enabled (unless you have disabled this feature) so that you can see exactly what error message is printed by the VIC.

TEXT

The TEXT statement has two forms, depending on whether you want to be PRINTing to the screen using LOWER case letters as normal (with upper case produced by using the shift key) or PRINTing to the screen using UPPER case letters (with the shift key producing graphics characters). These two forms are:

```
        TEXT LOWER
and     TEXT UPPER
```

where you can shorten LOWER and UPPPER to L and U respectively. When executed either of these statements clears the screen, resets the VIC to its normal 22 column by 23 row text screen and sets either LOWER or UPPER case for unshifted characters depending on which form you used.

A program using VICKIT II statements will start with a GRAPHICS statement to prepare the VIC for high resolution graphics, use CLEAR statements whenever a clear graphics screen is needed or when changing from a text screen (you can use PUT to save or restore an entire graphics screen image if you want to) and TEXT statements whenever the program reverts to printing normal text on the screen. Programs should always make sure they do a TEXT statement before ENDing or STOPping. If you press STOP while using high resolution graphics the simplest way to return to a normal text screen is to press the RESTORE key while holding the STOP key down.

Now you can set the VIC to work with high resolution graphics it is time to introduce the first three statements will actually produce something on the screen: SET, RESET and INVERT. These three statements have very similar forms so we deal with them all at the same time.

**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 x and y co-ordinates of the point to be either SET, RESET or INVERTed.

Using SET will result in the point (x,y) taking on the colour specified in the last CLEAR statement. RESET results in the point (x,y) taking on the colour of the screen background (this is usually white unless you have used the POKE command or statement to alter the screen and/or frame colours as shown on page 134 of the PERSONAL COMPUTING ON THE VIC-20 handbook supplied with your VIC). Using INVERT will take the colour of the point (x,y) and invert it, i.e. if it is the colour of the screen background INVERT makes it the colour last specified in a CLEAR statement while if it is the colour last specified in a CLEAR statement INVERT makes it the colour of the screen background.

Values for x and y can be either 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                       :REM DRAW BLUE ON WHITE
30 FOR X=80 TO 100
40   SET (X,80)
50   SET (X,100)
60 NEXT X                        :REM BOTTOM AND TOP
70 FOR Y=80 TO 100
80   SET (80,Y)
90   SET (100,Y)
100 NEXT Y                       :REM LEFT AND RIGHT SIDES
110 GET A$:IF A$="" THEN 110
120 TEXT LOWER
130 END
```

After the box has been drawn line 110 waits until a key is pressed before using TEXT to return the VIC to a normal TEXT screen. Although each VICKIT II statement is on a separate line the program will work properly if as many as possible are crushed on to a line, although the result may not be terribly easy to understand.

Looking at the program you can see that the four points at the corner of the square ((80,80), (100,80), (100,100) and (80,100)) will have been SET twice. SETting a point that is already SET or RESETting a point that is already RESET has no effect. However alter the SET statements in lines 40, 50, 80 and 90 to INVERT and see what happens. INVERTing a point twice is equivalent to leaving the point as it was, SET or RESET, so the vertices of the square are not SET but RESET.

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 length between one dot and the next one, but the size of horizontal and vertical dots differs. If you compare the physical lengths of a horizontal and a vertical line of the same drawn length you should find that the vertical line measures about 0.6 of the horizontal line. To get a more realistic square the loop in line 30 should read:

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

and line 90 should be:

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

Using VICKIT II statements you can now SET, RESET or INVERT any point on the screen. Perhaps you may have wondered what happens if you try to SET, RESET or INVERT a point which is not on the screen, i.e. a point (x,y) where either x is out of the range 0 to 175, y is out of the range 0 to 159 or both are out of their respective ranges. If VICKIT II gets an x or y coordinate which is out of its true range but inside the range 0 to 255 the offending coordinate is replaced by 175 if it is an x coordinate and 159 if it is a y coordinate. If a coordinate is out of this extended range a ?ILLEGAL QUANTITY error is reported. If you are plotting a curve or shape and unexpected lines appear at the top or right of the screen this is a signal that you are trying to plot points out of range.

Once you have used SET, RESET and INVERT you might want to find out in which state a point is on the screen. Is it SET or RESET, i.e. the colour specified in the last CLEAR statement or the background screen colour? The VICKIT II statement POINT provides a means for getting this information.

POINT

The POINT statement has only one form:

```
        POINT (x,y),variable
```

where (x,y) are as described before and variable is a real, non-array variable, e.g.:

```
        POINT (100,100),PT
```

but not:

```
        POINT (100,100),PT%
or      POINT (100,100),PT(0,0)
```

When the POINT statement has been executed the variable used will be set to 0 if the point specified was RESET, 1 if it was SET and -1 if the point specified did not lie on the screen. These three values mean that you can then use the BASIC ON statement as in this example:

```
        100 POINT (100,100),PT:ON PT+1 GOTO 1000,2000,3000
```

which will GOTO line 1000 if the point was off the screen, 2000 if it was RESET and 3000 if it was SET.

When the PUT statement is introduced you will have a more powerful technique for detecting the state of a point or group of points on the screen but until then POINT will suffice.

The five remaining statements provided by VICKIT II are LINE, DRAW, FILL, PUT and CIRCLE.

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 the point (x2,y2) is:

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

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

```
10 GRAPHICS
20 CLEAR 7
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 or updated DRAW (see next section) are substituted. Another version of the box drawing program replaces lines 40, 50 and 60 of the above version with:

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

The ,S in all the above lines could also be omitted.

Drawing boxes is such a common requirement that a special form of the LINE statement allows you to draw a box with one statement:

```
LINE (x1,y1)-(x2,y2),colour,option
```

where option is either B (BOX) or F (FILL). With B a box is drawn with four lines from (x1,y1) to (x2,y1), (x2,y1) to (x2,y2), (x2,y2) to (x1,y2) and finally (x1,y2) to (x1,y1). The endpoint of the last LINE drawn is then (x1,y1) and this is used if a future LINE statement omits (x1,y1).

With the F option 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).

The DRAW statement provided by VICKIT II takes a series of instructions held in a string and produces a shape by interpreting each of these instructions. These strings can be stored on cassette or disk and a general purpose drawing routine used to draw them rather than a specific program producing each specific shape. Since the instructions are stored as strings they can be manipulated by the program itself, so part of the string can be drawn, part erased and so on. You can use DRAW simply but it will amply repay the effort needed to use it to its full extent.

DRAW

The form of the DRAW statement is quite simple:

```
DRAW stringexpression
```

where stringexpression can be a string constant, string variable or string expression, e.g.:

```
      "R2Ø:U1Ø:L5:U1Ø:L1Ø:D1Ø:L5:D1Ø:"
or    A$
or    "B"+A$+"R1Ø:U1Ø:L1Ø:D1Ø:"
```

As the last example shows you can use string concatenation 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 the stringexpression 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 instruction has two forms, depending on whether an absolute move 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 (or subtracted if preceded by a minus sign) to the current 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 DRAW here is another way to produce the box drawn in the previous section:

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

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. Obviously an absolute Move instruction will only draw at one place on the screen and while relative Moves can produce shapes which are relocatable the shapes cannot be Scaled and Turned.

You will probably have guessed by now that Right draws a line to the right, Up draws a line up, Left a line left and Down a line down so here is yet another way to produce a small box in the middle of the screen:

```
10 GRAPHICS
20 CLEAR 7
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 vertical line (Up or Down) with a horizontal one (Left or Right) to produce a sloping one. If you separate a Left or Right instruction from an Up or Down instruction by a colon (or semi-colon) they are DRAWn separately, so that:

```
        "R1ØØ:U1ØØ:"
```

DRAWs a line to the Right and then a line Up. If you separate the two instructions by a comma:

```
        "R1ØØ,U1ØØ:"
```

the line that is DRAWn is a diagonal one, Up 1ØØ units and Right 1ØØ units. The following DRAW instructions DRAW a triangle on the screen (L or R MUST come first):

```
        "R4Ø:U3Ø:L4Ø,D3Ø:"
```

You can of course precede any Up, Down, Left or Right instruction, whether combined or not, with B or N (the combination B and N makes very little sense). The values that are used for the lengths can be constants, e.g. 1ØØ, 4Ø or 3Ø as used above, or variables but must not include a sign since the U, D, L or R tells DRAW which way to move. Using variables with Up, Down, Left and Right means that one string can be used to DRAW different sized shapes, e.g. by setting SIDE to a suitable value the instructions:

```
        "R SIDE:U SIDE:L SIDE:D SIDE:"
```

will always DRAW a square of size SIDE (spaces can be freely used 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 instruction 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. The 'subroutine string' can also contain an X instruction and this nesting can be repeated, although if not enough space is available in the BASIC stack a ?OUT OF MEMORY error will be reported.

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

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

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 plotting 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 Colour, Turn or Scale within a 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 from then on until either another Colour instruction or the end of this DRAW statement. The form of the Colour instruction is:

```
Cc:
```

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

T

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

```
Tt:
```

where t is either a constant, e.g. 0, 1, 2 or 3, or a variable which should have a value of 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
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 and Right lines. Lines can be made larger or smaller by a factor of 1 (i.e. full size), 2, 4 or 8. The form of the Scale instruction is:

```
Ss:
```

where s is either a constant with a value in the range 0 to 7, or a variable with a similar value. 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 example Turn program to:

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

Not only do the rectangles rotate but they also get smalle. Inserting a line 55:

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

and changing 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 scale 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.

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 a RESET area or 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 using LINE with the F(ILL) option but using the FILL instruction the shape need not be rectangular and you need not know the size of the area to be FILLed before starting.

The FILL algorithm, the process that VICKIT II goes through FILL an area, is not very sophisticated so there are areas wi which it will not cope. Shapes with internal angles less th 181 degrees will usually be FILLed correctly, although as t angles get smaller and smaller FILL may experience problem However any shape can be split up into smaller shapes which wi be FILLed successfully.

The form of the FILL statement is:

```
      FILL (x,y),colour
```

where the point (x,y) is the point at which FILL will start work and colour is either S(ET) or R(ESET). Colour gives tl colour of the the boundary which stops FILL.

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

```
10 GRAPHICS:CLEAR 7
20 DRAW "BM(5,5):R15:U5:L5:U5:L5:D5:L5:D5:"
30 FILL (12,12),S
40 GET A$:IF A$="" THEN 40
50 FILL (12,8),R
60 GET A$:IF A$="" THEN 60
70 TEXT LOWER
80 END
```

This program will DRAW a shape on the screen at the bottom lef hand corner, then FILL it and wait for a key to be pressed. I will then FILL the shape again, but this time RESET the area. I no colour is specified FILL uses SET and if you attempt to FIL an area which is not bounded by the colour specified FILL stop at the edges of the screen.

It is a good idea to experiment with FILL and see what shapes i will or will not FILL correctly. Often you will find that givin a different starting point for the FILL statement will make th statement succeed where it failed before.

The effect of internal angles approaching 0 can be seen with th following short program:

```
10 GRAPHICS
20 CLEAR 7
30 DRAW "BM(0,0):R100:U100:L100,D100:"
40 FILL (50,10)
50 GET A$:IF A$="" THEN 50
60 TEXT LOWER
70 END
```

which DRAWs a triangle and almost FILLs it.

PUT

The PUT instruction can be described simply but can be used in ways which are not at all obvious from its description. As with DRAW the effort required to use PUT to its full extent will be amply rewarded.

The PUT instruction just 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, in an array they form numbers which can be manipulated with normal BASIC commands (you can even talk about operations such as 'adding 1 to a shape') although you will usually use PUT again to do any manipulation. The PUT statement has the following form:

```
        PUT arrayelement direction (x1,y1)-(x2,y2),rule
```

where arrayelement is the name of any array element, e.g.

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

Arrayelement MUST be a number, real or integer (as you will see integer arrays are 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 an array (11 elements for every dimension) to hold the information you are going to put in it.

Direction is either > or <. > signifies that the contents of the array are to be placed on the screen and < that the contents of the screen are to be placed in the array (the author apologises for the horrendous syntax).

(x1,y1)-(x2,y2) gives the coordinates of two diagonally opposite corners of a rectangular area on the screen. This is the area that will be transferred to the array (using <) or filled with the array (using >).

Rule is a number in the range 0 to 15 which tells VICKIT II how to combine what is on the screen with what it is in the array. The result of this calculation is then placed on the screen or in the array. For simple applications this is omitted in which case it is taken as 3 if moving to the screen, 5 if moving to the array. A table at the end of this section details the 16 possible rules but useful rule numbers will be pointed out as the details of PUT are filled in.

While experimenting with PUT omit ,rule for the moment. The most useful rule for the PUT statement you have written will be selected automatically.

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

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

DRAWs a shape on the screen, reads that shape into the array SHAPE then places it back on the screen in a different location.

Array sizes

When using PUT you need to calculate the size of the area on the screen you are interested in to be sure that the array you are using is large enough. No harm will occur if the array is too small (except that PUT will not work correctly) or if it is too large (except that you will be wasting memory). 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 have provided a large enough array. 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

Having described how to use PUT in its simplest form it is time to look at some of its more powerful features. In the description of PUT it was stated that '(x1,y1)-(x2,y2) gives the coordinates of two diagonally opposite corners of a rectangular area on the screen'. 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(7Ø4) or DIM SCREEN%(176Ø)) you can even read the entire screen into a BASIC array and then reflect it to produce kaleidoscope patterns.

You should also notice that since you specify the array element for PUT to start with there is no requirement that it be the zero'th element all the time and you need not specify the same shape or size of area on the screen for replacing as you did when saving. To get the most out of PUT you should read the section in **Information for machine code programmers** about it.

Collisions

Immediately after a PUT statement and before any other VICKIT II statements have been executed PEEK(175) AND 1 is zero if there were no points set in the screen area examined or 1 if there were any points set. In addition PEEK(175) AND 2 is zero if there were any points set in the array area examined or 2 if there were any points set. This 'collision detection' facility allows you to discover easily if placing a shape on the screen will overwrite a shape already there.

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' of the PUT. Rules are numbers from Ø to 15 and, together with their results, are described below. The result of any rule is given as Ø or 1 and depends on the value of the screen point S (Ø for a RESET point, 1 for a SET point) and the corresponding array point A. A zero result produces a RESET point if placed on the screen while a non-zero (i.e. 1) result produces a SET point.

Rule number	Result	
Ø	Always Ø	
1	A and S	(1 if A=1 and S=1)
2	A and (not S)	(1 if A=1 and S=Ø)
3	A	(1 if A=1)
4	(not A) and S	(1 if A=Ø and S=1)
5	S	(1 if S=1)
6	A xor S	(1 if A=1 and S=Ø or if A=Ø and S=1)
7	A or S	(1 if A=1 or if S=1 or if both=1)
8	not (A or S)	(1 if A=Ø and S=Ø)
9	A equiv S	(1 if A=S)
1Ø	not S	(1 if S=Ø)
11	A or (not S)	(1 if A=1 or if S=Ø)
12	not A	(1 if A=Ø)
13	(not A) or S	(1 if A=Ø or S=1)
14	not (A and S)	(1 if A=Ø or S=Ø or both=Ø)
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 besides those offered by LINE and FILL. Some can be used in 'tricky' ways, e.g. using rule 3 with direction < seems to accomplish nothing at all but does allow you to detect the existence of any non-zero array elements using collision detection. The section of this manual on Hints, tips and tricks contains a number of these unusual ways to use PUT and you will probably discover new ones for yourself.

CIRCLE

Although the CIRCLE instruction is detected by VICKIT II there is simply not enough room on the VICKIT II chip to hold the programming needed for it. To provide CIRCLE therefore a separate program has been written to load the programming needed into the programmable memory of the VIC. This will be available separately and you should enter this program into your VIC and save it, using the APPEND technique given earlier to add it to your own programs. An attempt to use CIRCLE without this program will probably result in a ?ILLEGAL QUANTITY error.

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

```
CIRCLE (x,y),radius,colour,ratio,start,finish
```

where	(x,y) is the centre point,
	radius is the radius of the circle,
	colour is the colour (S(ET), R(ESET) or I(NVERT)),
	ratio is the height/width ratio,
	start gives the start point of the circle,
and	finish gives the end point of the circle.

The defaults for colour, ratio, start and finish (the values used if they are not given explicitly) are S(ET), 1, Ø and 1.

The height/width ratio is the value of the height of the figure to be drawn divided by its width. For a circle of course this value should be 1, for an ellipse which is taller than it it is wide it will be greater than 1 while for an ellipse which is wider than it is tall it will be less than 1.

The start 'round' the circle that the start point and be. VICKIT II will start to draw the circle at 3 o'clock and draw clockwise. Thus .25 of the way round the circle is at 6 o'clock, .5 at 9 o'clock, .75 at 12 o'clock and so on. Giving values for start and finish allows you to produce arcs of either circles or ellipses.

If you give any of the four optional values (colour, ratio, start, finish) a value, 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.

Error handling by VICKIT II

When enabled VICKIT II handles errors in a somewhat different way to that taken by a normal VIC. Since the error may have happened while the VIC was in high resolution graphics mode VICKIT II first of all intercepts the error before the VIC handles it, saves the error message and restores the screen to normal text mode (in effect it executes a TEXT UPPER statement). The error message is then restored and the VICKIT II routine for setting up HELP information is executed. This stores the information that might be needed later if you enter the VICKIT II command:

```
HELP
```

VICKIT II then rejoins the normal VIC error handling routine.

A side effect of the TEXT UPPER routine is that the screen is cleared since otherwise the screen would become full of garbage. While the VIC is working with high resolution graphics this is reasonable but you will probably have noticed that getting an error while entering a program or a command results in the screen being cleared unnecessarily. If you are prepared to put up with this you can ignore the rest of this section, but if you find it annoying read on.

To alter the error handling by VICKIT II so that the screen is not cleared on an error you should follow these steps once VICKIT II has been enabled:

1 Check that location 768 contains 49. You can do this with the command PRINT PEEK(768). If location 768 does not contain 49 then either you have already altered the error routine with these steps (in which case you should get 56), you have not initialised VICKIT II (in which case you should get 58) or you have some other system enabled which also intercepts the error.

2 If location 768 does contain 49 then replace it with 56, using the command POKE 768,56.

3 To reverse the process, that is to let VICKIT II clear the screen on an error simply follow steps 1 and 2 again but look for 56 in step 1, and replace it with 49 in step 2.

You can perform these steps in your program if you wish with sections of programming such as:

```
1000 REM REMOVE TEXT UPPER ON ERROR
1010 IF PEEK(768)=49 THEN POKE 768,56:RETURN
1020 PRINT "ERROR LINK NOT CORRECT":STOP

2000 REM RESTORE TEXT UPPER ON ERROR
2010 IF PEEK(768)=56 THEN POKE 768,49:RETURN
2020 PRINT "ERROR LINK NOT CORRECT":STOP
```

Examples of programming with VICKIT II

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 II's high resolution graphic instructions. Taken together with the next section (Hints, tips and tricks) this should give 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
```

rogram 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
```

rogram 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
```

e that variables whose names clash with VICKIT II statement words, e.g. T$ 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 TRY TO 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.

Hints, tips and tricks

This section is devoted to describing features of VICKIT II commands and statements not immediately obvious from their descriptions. Often these were not designed into VICKIT II but arise from looking at the statements in unusual ways. For example when the LINE statement has the FILL option VICKIT II takes the two points given and swaps the coordinates around (if need be) so as always to draw left to right and bottom to top. While PUT could also do this you get far more power by leaving the coordinates as they are given since PUT then provides a way to do reflections.

AUTO

When AUTO is working it prints a line number and what looks like a space. It is in fact a shift/space which means that you can include spaces before the BASIC statement you are about to type and they will be kept in your program. You will have noticed that all the statements inside FOR...NEXT loops in the sample programs have been indented since this helps to show up the structure of the program.

One point to watch is that if you screen edit a line which has been indented by this method you will have to re-insert the shift/space in order to keep the indentation.

STEP and
TRACE

STEP and TRACE are totally incompatible with VICKIT II high resolution statements.

Screen size

You will have noticed by now that an area at the bottom of the screen is never used to plot high resolution graphics. The purpose of this area is to allow short messages to be displayed on the screen without disrupting what is on the screen. If you want to use this facility the following short BASIC subroutine will display a two letter message at the bottom left hand corner of the screen:

```
5000 REM MESSAGE ASSUMED TO BE IN CH$ (LENGTH 2)
5010 FOR I=1 TO 2
5020   CH=ASC(MID$(CH$,I,1)):IF CH>=64 THEN CH=CH-64
5030   FOR J=0 TO 7
5040     POKE 7616+J+(I-1)*8,PEEK(32768+CH*8+J)
5050   NEXT J
5060 NEXT I
5070 RETURN
```

Should you want to remove the inaccessible area include the following statement in your program after each CLEAR statement:

```
POKE 36867,PEEK(36867) AND 253
```

Most of the simple VICKIT II statements, GRAPHICS, CLEAR, TEXT, SET, RESET, INVERT and POINT have no hidden depths which can generate suprises although you can use the TEXT instruction even if you are already using the VIC for text. In this case you clear the screen and set the character generator to produce UPPER or LOWER case as specified in the TEXT statement which can be easier to understand than the corresponding:

```
        PRINT "[cls]";CHR$(14)
or      PRINT "[cls]";CHR$(142)
```

LINE

LINE can be used in a number of ways which may not be immediately obvious from the description. For example, to produce random lines across the screen you would use:

```
20 FOR I=1 TO 100
30    LINE (176*RND(TI),160*RND(TI))-(176*RND(TI),160*RND(TI))
40 NEXT I
```

which produces unconnected LINEs. To produce a random set of LINEs on the screen which are joined to each other replace line 30 with:

```
30    LINE -(176*RND(TI),160*RND(TI))
```

If you want to produce thick-walled boxes using LINE the easiest way is to first produce the outside wall using LINE with colour SET and option FILL, then produce the hole in the middle of the area using LINE with colour RESET and option FILL, e.g.:

```
20 LINE (0,0)-(100,100),SET,FILL
30 LINE (10,10)-(90,90),RESET,FILL
```

DRAW

By using an array CHAR$ defined as CHAR$(25) you can use DRAW to give yet another way of writing characters on the screen. For example, if CHAR$(0) was defined by:

```
CHAR$(0)="R4,U9:R4,D9:BL2,U4:L4:BR7,D4:"
```

then DRAWing CHAR$(0) writes an upper case A on the screen at the current DRAWing position and leaves the DRAWing position ready for another character. Since you use DRAW rather than PUT the Scale and Turn instructions can be incorporated to produce variations on the characters.

A point about writing which holds for PUT as well is that if you use the CI: instruction in DRAW, or the ,6 rule for PUT, then DRAWing or PUTting in the same location twice effectively erases what was DRAWn or PUT and restores what was originally on the screen. Thus instructions can be placed on the screen and removed when the user has taken notice of them leaving the high resolution screen unchanged.

PUT

Many of the tricks possible with PUT have been described already but there are still one or two intriguing possibilities left. For example you can define readable characters which fit in an area on the screen 4 points wide by 8 high. If the characters are placed in an array as with example 6 in Examples of programming with VICKIT II then PUT can be used to produce a 44 column VIC simulation.

Defining your own specialised characters (mathematical, APL, Greek, Russian etc) is yet another possibility that PUT allows. Possibly the best idea would be to write a character font generation program in BASIC which would allow you to define and edit easily a given character set, placing them in an array which could then be saved to tape or disk. By using a common type of array between programs you could write programs which could run with alternative character sets.

Besides defining smaller characters it is of course equally easy to define larger ones. With more space available much finer detail can be incorporated into a character, so that it should be possible to produce specific type faces on the VIC for animated noticeboards and the such like.

GENERAL HINTS ON GRAPHICS PROGRAMS

Always make sure that the user of the program knows what is going on. A number of ways have been described for getting characters and messages on the screen at the same time as graphics so use them.

Using sound as well can be a greater enhancer of programs. Simply giving a 'beep' to draw the user's attention to the screen can save a lot of time and trouble.

Always try and make something happen on the screen, even if it is only a flashing cursor:

```
6000 REM HIGH RESOLUTION FLASHING CURSOR AT POINT (X,Y)
6010 PUT CURSR(0) < (0,0)-(7,7),15
6020 FOR N=1 TO 100
6030   PUT CURSR(0) > (X,Y)-(X+7,Y+7),6
6040   FOR DELAY=1 TO 200:NEXT DELAY
6050   PUT CURSR(0) > (X,Y)-(X+7,Y+7),6
6060   FOR DELAY=1 TO 200:NEXT DELAY
6070 NEXT N
6080 RETURN
```

although some sort of small but intricate drawing is even better. This is particularly useful if you are doing some set up routine which may take a long time, for example reading in an entire character set to an array for PUT or DRAW.

Multicolour mode

When using the CLEAR statement the number following CLEAR is usually in the range 1 to 8 for normal high resolution graphics, but a number in the range 9 to 16 selects a mode of graphics 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 as described on page 134 of PERSONAL COMPUTING ON THE VIC-20, the foreground colour is the colour you specified in the CLEAR 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.

You can now 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)
```

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)
```

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.

As an example of the multicolour mode here is a program showing lines in three different colours:

```
10 REM TRICOLOUR BY E. HULME
20 GRAPHICS
30 CLEAR 11
40 POKE 36879,94:POKE 36878,(PEEK(36878) AND 15) OR (7*16)
50 X=80:REM LINE 40 SELECTS SCREEN, FRAME AND AUXILIARY COLOURS
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
160 GET A$:IF A$="" THEN 160
170 TEXT LOWER
180 END
```

Conversion from or to other graphics systems

Four alternative graphics systems will be considered here, those provided by the IBM Personal Computer (tm), the Tandy Color Computer (tm), the BBC Microcomputer and, perhaps most interestingly, the Turtle Graphics approach.

On the whole VICKIT II statements are compatible with those on the IBM and Tandy machines and where possible steps have been taken to make the compatibility greater. The most common difference is in the names of the statements (e.g. VICKIT II uses FILL where both IBM and Tandy use PAINT) but the syntax of statements is often different due to the reduced resolution and colour alternatives available on the Commodore VIC.

Screen size, layout and resolution

Whereas the VIC with VICKIT II handles only one high resolution mode (176 by 160) the IBM, Tandy and BBC machines can work in various resolutions: 640 by 200 down to 160 by 100 for IBM, 256 by 192 down to 128 by 96 for the Tandy and 640 by 256 down to 160 by 256 for the BBC. Thus the IBM machine has a statement SCREEN to select which mode to work with (including a mode for text), the Tandy PMODE and the BBC simply MODE. Once the mode is selected IBM use CLS to clear the screen, Tandy use PCLS and the BBC CLG (CLS for the text screen). The nearest equivalent to these statements with VICKIT II is the CLEAR statement, which also incorporates the information about colour supplied by IBM and Tandy with the COLOR statement and by the BBC with the GCOL statement. VICKIT II uses TEXT to return to graphics mode which has no direct equivalent in any of the other machines.

A more awkward problem is that of the screen resolution and arrangement. Tandy place the origin of the graphics screen (0,0) at the top left hand corner of the screen and it appears that IBM may have followed a similar approach. Together with the different resolutions there may be some problems in transferring programs which make use of the highest resolution and/or the position of the origin.

Instructions dealing with single points

SET and RESET instructions have direct equivalents (PSET and PRESET respectively) in both IBM and Tandy machines. There does not appear to be any INVERT statement on either machine nor any way of giving LINE or DRAW the colour INVERT. PSET and PRESET set the current drawing position whereas with VICKIT II you must DRAW a blank move to do this. The BBC machine uses PLOT n,x,y where n is 69 for SET, 71 for RESET and 70 for a form of INVERT.

The POINT instruction on VICKIT II has no direct equivalent on any of the other three machines since they use POINT as a function which can be inserted in any expression as a normal BASIC function. Thus the IBM, Tandy and BBC line:

```
        10 IF POINT(X,Y)<>0 THEN 100
```

must be replaced by:

```
10 POINT(X,Y),T:IF T<>0 THEN 100
```

with VICKIT II.

Advanced VICKIT II statements

LINE

The LINE statement is practically identical on the IBM, Tandy and Commodore machines, the only differences being due to syntax. With Tandy the colour can only be SET or PRESET and option is either B or BF (BOX and BOX FILL). Although no information was available at the time of printing about the IBM LINE syntax it offers the same facilities and is written by Microsoft who also wrote the Tandy system so it should be similar. The BBC machine appears not to have any built in instruction for drawing boxes or filled-in boxes but PLOT has a value for n giving filled-in triangles.

DRAW

DRAW is fairly similar on the IBM, Tandy and Commodore machines with some minor additions to the VICKIT II version. Both the IBM and Tandy machines seem to be limited to horizontal, vertical or diagonal lines (they use E, F, G and H for diagonal lines) whereas VICKIT II allows arbitrary lines using the , separator. VICKIT II DRAW allows both : and ; as separators between DRAW instructions, : since it is closer to BASIC and ; to increase compatibility with other machines. VICKIT II also insists that EVERY DRAW instruction is terminated with either : or ; whereas both the other machines relax this requirement.

The M instruction on the Tandy allows the B and N options as on VICKIT II but uses + and - signs to signify a relative move whereas VICKIT II uses parentheses to signify an absolute point in every command that requires one. The IBM move uses a separate instruction (B or N) to produce the effect of BM or NM on either the Tandy or VICKIT II. Whether or not the B or N options can still be used with L, R, U or D is not obvious.

The C instruction is much the same on all three machines although of course both the IBM and Tandy offer a larger selection of possibilities (although not, it seems, INVERT).

The T instruction is renamed A (for Angle) on both the IBM and Tandy machines but otherwise works in a similar fashion.

The Scale instruction on the Tandy (and presumably the IBM) has a much larger range of values (from 1/4 to 62/4 on the Tandy). Both Turn and Scale can take their values directly from variables whereas the IBM machine at least requires an equals sign between T (or S) and the variable name.

The X instruction appears to work identically on all three machines.

DRAW on the BBC machine is used to draw lines from the current drawing position to an absolute point. MOVE is used to produce a blank DRAW.

FILL

The VICKIT II FILL statement has an almost identical form to that used by both IBM and Tandy although they both use the word PAINT (which seems to be easily confused with POINT). The PAINT statements can also specify a larger range of boundary colours at which to stop. The only equivalent on the BBC machine appears to be a value for n with PLOT which gives filled-in triangles.

PUT

There are a a large number of differences between the IBM, Tandy and Commodore machines with the PUT statement. Both IBM and Tandy use PUT to place information on the screen and GET to place it in an array whereas VICKIT II uses just the one word PUT and the > and < direction characters. The rectangular area concerned is specified in the same way on all three machines but it is not clear whether either the IBM or Tandy allows the reflection of images in the way that VICKIT II does. The rule for combining screen and array images is limited to PSET, PRESET, AND, OR and NOT on the Tandy machine (rule numbers 3, 12, 1, 7 and 10) and PSET, PRESET, AND, OR and XOR on the IBM (rule numbers 3, 12, 1, 7 and 6). In addition the rule number is not specified when transferring information to the array, only when transferring to the screen.

The BBC machine appears to have no equivalent to PUT and GET although the GCOL statement allows AND, OR, XOR and INVERT colour operations, somewhat like the ,rule option for PUT in VICKIT II.

CIRCLE

There seem to be no obvious differences between the machines with CIRCLE although the IBM machine does have an apparently undocumented feature which allows the connection of arc end points to the centre and appears to specify the start and finish points with respect to angles expressed in radians. Thus default values for start and finish points would be 0 and 2*pi (2*pi radians=360 degrees).

Turtle Graphics

Turtle graphics is an approach to teaching, rather than to graphics, which uses the idea of a turtle moving on a screen and (usually) leaving a trace behind it. Using simple ideas such as FORWARD, LEFT, BACKWARD, RIGHT, PEN UP and PEN DOWN the turtle can not only produce a large variety of patterns and shapes on the screen but can lead the inquisitive user to branches of mathematics and even physics which seem to have little connection with the original idea of the turtle.

The best book available to describe turtle graphics is Turtle Geometry, by Harold Abelson and Andrea diSessa, published by the MIT Press. Besides giving a large number of procedures for drawing shapes and patterns on the screen of a computer this book can lead to such things as General Relativity, curved space-time and spherical geometry.

As an appendix to the book the authors provide information on how to handle turtle graphics in languages such as BASIC and it is from that appendix that the sample program below was developed. In it a basic turtle graphics 'shell' of subroutines is defined, and a simple procedure given to produce an interesting pattern. For a more useful implementation of turtle graphics one would probably want to create an interpreter of the turtle graphics primitives but the shell approach will be initially useful.

The turtle has no overall view of the world it inhabits since it expresses its movements with reference to its current location and heading. Thus typical turtle commands are:

```
FORWARD 100
RIGHT 25
```

and so on. FORWARD makes the turtle move in the direction it is currently heading a certain distance while RIGHT makes the turtle turn clockwise a certain number of degrees. To make things easy we also define BACK and LEFT. PEN UP and PEN DOWN should be obvious.

The turtle shell defined here has an initialisation section and separate subroutines for each turtle primitive. To write a procedure for the turtle one therefore embeds a number of GOSUBs to the relevant routines within the turtle shell. On RUNning the program the turtle goes through its paces.

Turtle shell

```
100 GRAPHICS
110 GOSUB 10000
120 REM HIGHEST LEVEL VARIABLES/ARGUMENTS SET UP HERE BEFORE 130

130 REM HIGHEST LEVEL PROCEDURES START AT 130

250 GET A$:IF A$="" THEN 130
260 GET A$:IF A$="" THEN 260
270 IF A$="C" THEN 130
280 TEXT LOWER:STOP
290 REM ANY KEY HALTS TURTLE, C THEN CONTINUES

300 REM LOWER LEVEL PROCEDURES START HERE

1000 REM FORWARD ROUTINE (D=DISTANCE TO MOVE)
1010 DX=COS(FNR(HEADING))*D:DY=SIN(FNR(HEADING))*D
1020 MX$="RDX,":IF DX<0 THEN MX$="LDX;":DX=-DX
1030 MY$="UDY:":IF DY<0 THEN MY$="DDY:":DY=-DY
1040 DRAW BLANK$+MX$+MY$
1050 RETURN
2000 REM BACK ROUTINE (D=DISTANCE TO MOVE)
2010 D=-D:GOTO 1000

3000 REM RIGHT ROUTINE (A=ANGLE TO TURN)
3010 HEADING=HEADING-A
3020 RETURN
```

```
4000 REM LEFT ROUTINE (A=ANGLE TO TURN)
4010 A=-A:GOTO 3000

5000 REM PEN UP ROUTINE
5010 BLANK$="B"
5020 RETURN
6000 REM PEN DOWN ROUTINE
6010 BLANK$=""
6020 RETURN

10000 REM INITIALISE
10010 CLEAR 7:DRAW "BM(88,80):"
10020 DEF FNR(D)=D*3.14159/180
10030 RETURN
```

An example of the use of the turtle shell is taken from page 30 of Turtle Geometry and is produced by ADDING the following lines to the shell:

```
10 REM SAMPLE OF TURTLE GRAPHICS ON THE VIC, IMPLEMENTING THE
20 REM POLYROLL PROCEDURE ON PAGE 30 OF 'TURTLE GEOMETRY' BY
30 REM ABELSON AND DISESSA. PROCEDURE IN TURTLE NOTATION IS:
40 REM
50 REM 'TO POLYROLL SIDE ANGLE1 ANGLE2'
60 REM '   REPEAT FOREVER'
70 REM '       POLYSTOP SIDE ANGLE1'
80 REM '       RIGHT ANGLE2'
90 REM

120 SIDE=40:A1=60:A2=45
130 GOSUB 300
140 A=A2:GOSUB 3000
150 REM
160 REM THIS IS EQUIVALENT TO 'POLYROLL 40 60 45'
170 REM

300 REM POLYSTOP
310 TURN=0
320 D=SIDE:GOSUB 1000
330 A=A1:GOSUB 3000
340 TURN=TURN+A1
350 IF (TURN/360)<>INT(TURN/360) THEN 320
360 RETURN
370 REM
380 REM THE POLYSTOP PROCEDURE IN TURTLE PROCEDURE NOTATION IS:
390 REM
400 REM 'TO POLYSTOP SIDE ANGLE'
410 REM '   TURN ← 0'
420 REM '   REPEAT'
430 REM '       FORWARD SIDE'
440 REM '       RIGHT ANGLE'
450 REM '       TURN ← (TURN + ANGLE)
460 REM '   UNTIL REMAINDER (TURN, 360) = 0'
470 REM
```

Although a turtle should have a larger resolution display to wander about

on than that provided by the VIC a suprisingly large number of th
subjects treated by Abelson and diSessa can be simulated on the VIC. T
really get a feel for the possibilities you need to read the book but i
is to be hoped you now have some idea of the possibilities.

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 II. A number of useful subroutine entry points are given, together with a description of the operation of the GRAPHICS and PUT instructions and details of memory used, both in page zero and absolute memory.

Little information is given about the VICKIT II command routines since they are so specific to the given commands but details of the memory used by the commands are given.

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 the example program 6, 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 II 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.

Page zero usage

Extensive use is made of page zero locations used by the cassette and RS-232 handling software. It is unlikely that cassette and/or RS-232 operations can be performed at the same time as VICKIT II instructions.

Location name	address	uses
PLOTOK	$A3	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	$A4	Start point for line drawn by DRLINE (x coordinate)
STRTY	$A5	Start point for line drawn by DRLINE (y coordinate)
LASTX	$A6	End point for line drawn by DRLINE (x coordinate)
LASTY	$A7	End point for line drawn by DRLINE (y coordinate)
LINCOL	$AD	Line colour, $00 for a SET line, $80 for a RESET line and bit 0 set for an INVERTed line
COLLID	$AF	Used in PUT to hold the results of the collision detection routine

All locations between $A3 and $B3 are used, the ones listed are the most useful.

Useful absolute locations

Location name	address	purpose
CUPOSX	$0334	Holds the current position (x coordinate)
CUPOSY	$0335	Holds the current position (y coordinate)
	$03ED-$03FF	Used by VICKIT II commands.
ADDRSS	$B85F	Enter with X and Y being coordinates of a point on the screen. 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	$B8A1	Start of a table of single bit masks, $80, $40 etc.
GTPNT	$B940	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.
TEXTL	$BA77	Restores screen to text with lower case option set.
TEXTU	$BA7A	As for TEXTL but restore with graphic option set.
SETPT	$BA9F	Assumes ADDRSS has been executed immediately before. SETs the point calculated by ADDRSS.
RESTPT	$BAA6	As for SETPT but RESETs the point.
INVERP	$BAB0	As for SETPT and RESTPT but INVERTs the point.
DRLINE	$BB42	Assumes STRTX, STRTY, LASTX, LASTY and LINCOL have been set up. Draws a line of the specified colour between (STRTX,STRTY) and (LASTX,LASTY).
CLEAR+3	$BA03	Enter with (X) being colour code used in CLEAR statement, e.g. 1 for BLACK. The screen is cleared and the VIC enters high resolution graphics mode.

Debugging Programs.

When using the programmers aid instruction of the VICKIT II on programs not containing graphics commands some precautions must be taken.

The VICKIT II error handling is geared towards the graphics section and hence when a programming error occurs the VICKIT error handling routine relocates the screen to 7680. If the VIC memory was expanded above 8192 then this new screen position would be overwriting the program area and corruption would occur. To avoid this either all programs must contain a graphics command as the first line or the VICKIT error handling routine must be disabled as described in the manual under ERROR HANDLING BY VICKIT II page 6:39.

ie. POKE 768,56.

SAVING PROGRAMS.

NOTE. After using graphics on the VICKIT II you must restore the cassette function otherwise you will be unable to save programs.

This can be achieved by:-

```
                    POKE 178,60
                    POKE 179,3
```

or by turning off the VIC and turning it on again.

LOADING THE CIRCLE ROUTINE

The procedure to be followed in order to use CIRCLE with VICKIT II depends on the amount of memory that you have in your VIC. If you have just a 3K memory expansion (giving 6655 bytes free when you turn the VIC on) follow the instructions headed CIRCLE for VICs with 3K memory expansion. If you have more than 3K memory expansion follow the instructions under the heading CIRCLE for VICs with more than 3K memory expansion.

Details of memory used by the CIRCLE routine are given in the section Information for machine code programmers.

In the VICKIT II user's manual a small section of the instructions for the CIRCLE statement is missing. This section concerns the use of the start and finish points in the CIRCLE statement. The fourth paragraph on page 5:37 should start:

> The start and finish points give the proportion of the distance 'round' the circle that the start point and finish point are to be.

You should note that because of the limited resolution of the VIC screen the values of start and finish may require a little experimentation to produce exactly the required result, especially when using a small radius.

Ellipses produced with the CIRCLE statement will be drawn so that the height is twice the width if ratio is 2, three times the width if ratio is 3 and so on. The width will always be twice the specified radiu in these cases. Where ratio is less than one the height will always be twice the radius specified and the width will be an integer multiple of the height. Thus a ratio of 0.5 generates a width of twice (1/0.5) the height, 0.33 a width of 3 times (1/0.33 approximately) the height and so on. This is different to the approach taken by the Tandy and IBM machines where the width is always twice the radius.

CIRCLE for VICs with 3K memory expansion

When your VIC is switched on LOAD the CIRCLE program with the command:

```
     SYS 4096 x 11
     LOAD "CIRCLE"
```

When the program has been LOADed correctly give the following commands:

```
     POKE 1,206:POKE 2,4:SYS(1028) : NEW
```

You can now use the CIRCLE statement as described in the VICKIT II user's manual.

CIRCLE for VICs with more than 3K memory expansion

When your VIC is switched on enter the following commands:

```
     POKE 1024,0:POKE 44,4:NEW
```

and then LOAD the CIRCLE program with the command:

VICKIT II and III CIRCLE.

When either VICKIT II or III draws a circle using the CIRCLE command it draws a mathematical circle, but unfortnately because of the 3:2 aspect of the Vic's screen it appears as an oval.

This limitation can be over come by use of a BASIC subroutine.

EXAMPLE

```
10 GRAPHICS : CLEAR 7.
.
.
.
.

80 X = 80 : Y = 40 : R = 20 : GOSUB 10000
.
.
.
.

9999 END.
10000 REM CIRCLE SUBROUTINE
10010 FOR T = Ø to 2 * 2 STEP .Ø5
10020 X1 = SIN (T) X R + X
10030 Y1 = COS (T) *(R* 1.33) + Y
10040 SET (X1,Y1)
10050 NEXT T
10060 RETURN
```

In line 80 X & Y are co-ordinates for the centre of the circle and R is the radius of the circle.

By altering the value of .Ø5 in line 10010 the degree of smoothness of the circle can be altered, but this is traded off against speed. The smaller this value the longer it will take to draw a circle.

This routine is slower than the one contained within the VICKIT chips, so where speed is required use the CIRCLE command and where a 'true' (it appears as a circle, but is not a mathematical circle) circle is required use the subroutine.

NOTE FOR CARTRIDGES ORDERED WITHOUT 3K RAM EXPANSION.

1). VICKIT 3/2 require extra RAM to operate. This extra ram must be made available before using the firmware cartridge. RAM size minimum 3K.

---

2). Circle Routine on VICKIT II is leaded in to the 3K memory area 1024-4096. If this is not made available by a 3K RAM cartridge or similar then the circle function will not be available.