FIFTH

USER'S MANUAL

BY

RICHARD TAYLOR

C R L

There were a lot of printing errors in this manual, particularly in the coding examples. I think I've corrected most of them, and I've checked that all the routines in this transcription work. **JimG**

INTRODUCTION TO MAIN INSTRUCTIONS

For a long time now there has been the need for a BASIC extension to improve its graphics capabilities. Graphics seem to have become the most important feature of any micro-computer. The Spectrum is no exception but fortunately has a quite reasonable graphic power. Its major short-coming is the fact that you can only have a maximum of two colours in a single character square. This problem can, however, be mainly solved by careful screen layout.

When you think about it there are few BASIC commands which actually affect the content of the screen: CLS, PRINT, PLOT, DRAW and CIRCLE. These can only directly make static displays. To produce a moving display is very complicated. By careful use of the PRINT command you can get some sort of "jumpy" movement. When you want to move more than about two objects at once then things get really complicated and even a very experienced BASIC programmer would find it extremely difficult to produce the convincing graphics. There is a least tendency to return to the sophistication of the early ZX81 BASIC games where in an aircraft carrier bombing game, for instance, the plane would stop dead in mid-air while the bomb would slowly, jumpily, move downwards. Since the pioneering days of the ZX80 machine graphics have improved dramatically. The BASIC, however, has not. It still contains the some old smattering of vague BASIC graphic statements.

"FIFTH" helps to bring BASIC back into "line" with the graphic power of today's machines. "FIFTH", although it does other things as well, is mainly designed to allow you to produce BASIC games with much the some effect as machine code ones. "FIFTH" also saves computer memory since operations that would have previously taken many lines of BASIC programming can be condensed into a small selection of the 25 "FIFTH" commands. "FIFTH" makes it so easy to get graphics moving around the screen that it makes it inviting to do so; encouraging you to write effective programs.

"FIFTH" graphics are incredibly smooth, the objects literally float across the screen making it a pleasure to watch them. You are also not limited to a few objects but as many as you like, within reason. "FIFTH" is not a language on its own but augments the already resident BASIC. You can make the two languages communicate with each other with the minimum of fuss.

You might think that the slow speed of BASIC would limit the performance of "FIFTH" but this is not so. One of the advantages of "FIFTH" is that the graphics are independent of the program; this does not mean you have difficult in controlling them but relieves you from the fuss of updating screen positions and erasing characters etc. This method is much faster than normal BASIC movement since the erasing, updating and re-printing routines are well written in ultra-fast machine code. To get a "FIFTH" object to move around the screen all you have to do is give the computer certain information about its direction and speed etc. "FIFTH" can then get on with the job of actually moving the object. It will keep moving it in the specified direction and speed until it goes off the edge of the screen or it hits something else. This is where another powerful feature of "FIFTH" comes in; a sort of "parallel" BASIC.

Parallel BASIC means that you have two independent programs running at the some time. "FIFTH" can't do that exactly since only one program can actually be running at once but appears to almost do so. What happens is that if, say, an object went off the edge of the screen then something called a service routine would be called. This is a short routine written in BASIC, which is supplied with the necessary information, (what went off the screen and in which direction) and has to do something appropriate. In most cases this would mean pointing the object the opposite way to what went off the screen and sending it on its merry way again, until it goes off the edge of the screen again or "hits" something else on its way. A routine to handle a collision between two objects, or "interacts" as they are called, is written in a similar way. In this case you would have to send each of the objects involved in the collision in opposite directions to avoid further ones. The advantage of this is that the service routine is called automatically, without any special prompting from the rest of the program. In fact, the rest of the program won't even know that it has been interrupted. This means that the "main-lap" of the program can be entirely unconnected with the objects moving on the screen. This lack of dependence on BASIC (except for the service routines and parts of the main loop) really means that the speed of BASIC is of less importance than usual. The moving graphics slow down BASIC quite a lot, depending on the amount of objects moving on the screen at that time. It still takes much less time, however, accessing each moving object individually like the normal way BASIC would produce movement.

As well as providing moving graphics, "FIFTH" also vastly improves the Spectrum's sound. The BEEP command does not produce anything like the zaps and bangs you would probably require in a game. The sound effects "FIFTH" provides are very useful for this purpose. "FIFTH" also has commands to rapidly change the on-screen colours and to print in larger than normal characters.

That completes this introduction to "FIFTH". I hope that it has given you an insight into the basic way "FIFTH" operates. Remember, you cannot write a program in just "FIFTH", it is an enlargement and extension to BASIC.

Printed by CLEARAPRINT
01-274 2527/0872

INSTRUCTIONS FOR "FIFTH"

Version 1.24 by Richard M Taylor, 1983

The ZX Spectrum is certainly a formidable and very powerful machine. The dialect of BASIC that it uses, Sinclair BASIC, is well blessed with a variety of useful extraneous commands. Like most other makes of machine it is used to a large extent for playing games on. Unfortunately, BASIC is not really designed for writing fast moving graphic games, This is especially true of Sinclair BASIC which lacks the speed of many other dialects. There is a complete absence of commands for moving characters, and larger blocks, around the screen with both smoothness and speed. The usual remedy for this problem has been writing games in machine code. This can certainly produce amazing effects but tends to be out of reach for the majority of users. Few people are willing to take the time and trouble of learning the whole new language of the machine code. "FIFTH" remedies these short-comings to a large extent by providing a large quantity of powerful, useful graphics commands. "FIFTH" is an extension to BASIC so there is only a small amount of learning to do if you already know BASIC - which I suggest you should, if you wish to use this program to its full potential.

Written entirely in machine code for the 48K version, "FIFTH" resides above RAMTOP. It occupies a shade over 4K or 4338 bytes to be precise. To LOAD use: CLEAR 61029: LOAD "" CODE

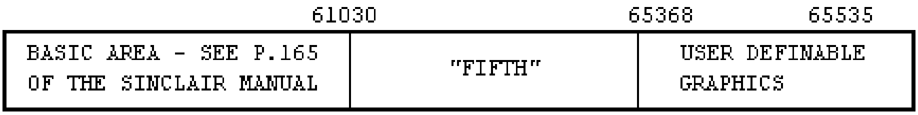


FIG.1 - "FIFTH's" Position in the Memory Map

Many of the "FIFTH" commands have to be cross-referenced with one another. References sometimes have to be made with commands not discussed at that particular point. For this reason, you may not be able to understand many things on the first reading of this manual but as you unconsciously inter-connect everything, your understanding should increase.

Examples are scattered liberally throughout this manual. They should be entered exactly as listed with the "FIFTH" toolbox program loaded. After running, if they do not stop automatically you should use BREAK. Type NEW to be sure of clearing the BASIC area before typing in the next example routine. You will not need to re-load the "FIFTH" program since it resides above RAMTOP and is not affected by NEW.

The commands are put into REM statements in the BASIC program. Every program which uses "FIFTH" must start with the following lines:-

```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
```

The 1000 in line 10 tells "FIFTH" how much memory to reserve for the object data; this may vary from program to program. This is fully described under the OBJECT command. Line 20 calls the main body of the actual "FIFTH" program. Although it is usual to have these two lines at the beginning of the program, this is not always so. In fact, it must be at the main entry point of the program which is better at the end rather than the beginning of a program.

After executing line 20 the interpreter carries on as normal (the interpreter is a larger routine in the Spectrum's 16K ROM which actually executes BASIC programs). There is, however, one exception; when the interpreter finds a REM Statement it treats it in a different way.. Normally a REM command would be completely ignored and the interpreter would go straight onto the next line.

However, the REM statement may now contain "FIFTH" commands so the interpreter acts accordingly. The first thing it does is to look at the first character; if this is an asterisk then it treats the REM as it would have normally. If it is not an asterisk then the interpreter can be sure that the REM contains "FIFTH" commands.

```
1000 REM * THIS IS A COMMENT
```

Comment lines can be put in as normal if you include an asterisk.

Like normal BASIC statements you can have multiple commands on one line but instead of being separated by colons they are separated by the rather neglected back-slash (back-slash is symbol shifted D in E mode). The parameters of "FIFTH" commands are separated by commas, as with the parameters of normal BASIC statements. Every BASIC command has its own token on the key-board but "FIFTH" commands obviously do not have their own tokens. YOU HAVE TO SPELL THEM OUT YOURSELF, INCORPORATING A TRAILING SPACE. It's up to you whether you use upper or lower case to do this. You can even change the case in the middle of command words eg:-

```
1000 REM TeMps\ lARge
```

is just as legal as:-

```
1000 REM temps\ large
```

or

```
1000 REM TEMPS\ LARGE
```

You can vary the case of the parameters in much the same way (the above two commands do not have any parameters - or arguments as they are more usually called). Personally, I think it is a good idea to type the commands in upper case and the parameters in lower case. This makes the listing more readable as well as making it look neater, You can just as freely incorporate control characters in "FIFTH" REM statements - see page 114 of the Sinclair Programming manual.

If the command you give is wrongly spelt or does not make sense for some reason then the computer will helpfully respond with error "Q Parameter error". This is usually used for the FN function. Since this is not utilised much in games writing, error Q very rarely occurs. Therefore error Q now takes on a second meaning of a syntax error in a "FIFTH" command. Error Q is also produced if a function name is incorrectly spelt.

Error "A Invalid argument" can occur if the arguments of a "FIFTH" command do not make sense. For more details see the section under the "FIFTH" functions. Other errors can occur but these are unique to each "FIFTH" command. They are fully explained under each command description. There is one other peculiarity of programming in "FIFTH". Normally, when a program is completed or a jump is made to a line number bigger than existing then "OK" will be the computer's response. However, in "FIFTH" OK is substituted for error "8 End of file". This report was originally designed for the elusive Sinclair Microdrive but has been put to this use in "FIFTH". The reason for using error 8 instead of error 0 is rather technical, so I will not go into it.

```
10 RANDOMIZE 1000
20 RANDOMIZE USR 61030
```

If you RUN this, notice how it stops
with error 8.

There now follows a detailed description of each of the 25 "FIFTH" commands:-

TEMPS

This command merely sets up colours for succeeding "FIFTH" commands. You will remember that there are two ways of using the BASIC colour statements, either universally as statements on their own or in graphic commands to specify temporary colours. This "FIFTH" command makes the temporary colours the same as the universal ones. It also transfers the state of INVERSE and OVER which can also be set up temporarily. This is because some "FIFTH" commands require "dummy" PRINT statements preceding them such as:-

```
       100 PRINT INK 6; PAPER 1; FLASH I;
```

This does not print anything but changes the temporary colours. If, however, you just wanted to use the universal colours then precede the "FIFTH" command with TEMPS. Commands which may use TEMPS are:

a) LARGE
b) FILL
c) REPLACE
d) COLOUR
e) PUT

Examples:-

```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 PAPER 5
       40 REM TEMPS\FILL
       50 PAPER 7

       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 PRINT PAPER 5
       40 REM FILL
```

Both of these routines use the FILL command, which is described next. They both do a similar job in making the background cyan. Notice how TEMPS is used in the first example.

FILL

This command is used to change the screen colours without actually affecting the screen display. It is an annoying feature of Sinclair BASIC that you have to clear the screen, using CLS, before new universal colours are shown. This command remedies the problem.

The colour you want to change the screen to is put into a dummy PRINT statement preceding the FILL command. e.g:

```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 PRINT "This is a demonstration of the
        FILL command"
       40 PRINT PAPER RND*7; INK 9;
       50 REM FILL
       60 GO TO 40
```

This program contains a lot of suitable points. It constantly changes the background colour always keeping the INK colour contrasted with it. Page 111 of the Sinclair BASIC Programming manual gives information about the use of colours 8 & 9 in PRINT statements. You can also use 8 & 9 in FILL commands; their use is explained below:-
"COLOUR" 8 - This leaves the appropriate type of colour (INK, PAPER, FLASH or BRIGHT) as it was previously.
"COLOUR" 9 - This can be used with either INK or PAPER. It makes one contrast with the other, in much the same way as in normal PRINT statements, e.g.

```
       100 PRINT PAPER 8; INK 9;
```

When used in front of a FILL command, this makes sure that all INK on the screen is in contrast with the PAPER, which is not changed.

Examples:-

```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 FOR a=0 TO 255
       40 PLOT a,0
       50 DRAW OVER 1;255-2*a,175
       60 NEXT a
       70 FOR a=0 TO 175
       80 PLOT 0,a
       90 DRAW OVER 1;255,175-2*a
       100 NEXT a
       110 PAUSE 50
       120 PRINT PAPER RND*7; INK 9;
       130 REM FILL
       140 GO TO 110
```

This program draws a "moire" pattern and then proceeds to change its colour once every second. The INK 9 in the line 120 ensures that the INK is never the same as the PAPER colour; i.e making the pattern invisible. You can of course use the TEMPS command instead of the dummy PRINT statement if the need arises; e.g.

```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 REM TEMPS\FILL
```

would make sure that the screen colours are the same as the universal ones.

REPLACE

The replace command is very similar to the FILL one except that it is more selective in the on-screen colours that it changes. It will only change a colour if it is another, specified, one. You have to specify two colours so the command uses both the universal and temporary colours. The temporary colour is the one to be searched for and the universal colour is the one which will replace it. For instance:-

```
       100 PRINT INK 1; PAPER 6;
       110 REM REPLACE
```

If incorporated into a program this would change every occurrence of blue ink on yellow paper to the current universal colour. "Colours" 8 & 9 as universal colours have their normal meaning, but 8 & 9 as temporary colours are interpreted slightly differently. Details below:-

Colour 8 - The appropriate colour type is ignored and so has no importance in the search; i.e.

```
       100 PRINT PAPER 8, INK 2;
```

If this was put before a REPLACE command then any attribute with red INK (PAPER is not important) would be set to the current universal colour.

Colour 9 - This has no importance or use in a REPLACE command. The TEMPS commands could be used but would not be of much use.

Can you see why? Any attribute that was already the universal would be replaced by the universal colour - not very useful.

Examples:-

```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 FOR a=0 TO 703
       40 PRINT INK RND*7;"*"; REM * An inverse space
       50 NEXT a
       60 PAUSE 0
       70 PRINT INK RND*7;
       80 REM REPLACE
       90 GO TO 60
```

This fills the screen with coloured blocks and every time you press a key, all blocks in a particular colour are changed to black. Press BREAK to escape from the program.

LARGE

This allows you to print larger than normal characters, or strings of characters, on the screen. Like the other commands discussed so far it has no parameters and needs a dummy PRINT statement before it. What is printed is determined by the state of 5 BASIC variables - x, y, t, w, and a$. The variables x and y determine where the top left-hand corner of the printout is to be. Unlike normal printing these are high resolution co-ordinates. x can be 0 to 255 inclusive and y can be 0 to 175 inclusive. When using normal BASIC high resolution statements the y co-ordinate starts from the bottom - (0,0) being the bottom left-hand corner of the screen. In contrast Hi-res "FIFTH" commands have the y co-ordinate starting from the top so (0,0) is the top left-hand corner of the screen. The variables t and w determine the size of the characters to be printed. The height is given in the variable t which should contain a number between 1 & 22. The width is given in the variable w which should contain a number between 1 & 32.
Here are some useful values:-

a) t=1, w=2 - double width
b) t=2, w=1 - double height
c) t=2, w=2 - double size
d) t=22, w=32 - A size at which a single character fills the whole screen.

If the value given in x, y, t or w is non-integer then it is rounded to the nearest integer. If this is out of range then error B will be given. a$ holds the string of characters to be printed. This can be of any length including zero characters (a "null" string). The string, however, should not contain any control characters. If you do then the computer will show its displeasure by replying with error "A Invalid Argument".
e.g.-

```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 LET a$="FIFTH"
       40 LET x=0: LET y=0: LET t=22: LET w=6
       50 REM TEMPS\LARGE
```

This example program will print "FIFTH" in large enough letters to cover the whole screen. Notice how the TEMPS command is used in line 50. If you precede the LARGE command with a dummy PRINT statement to set up temporary colours items then the large characters will be printed in the specified colours. The routine works by using part of the plot command routine, in the 16K ROM, to print the characters. Please note that the plot position is not changed by this command.

Examples:-

```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 LET x=0: LET y=0
       40 LET t=1: LET w=2
       50 LET a$="Double width"
       60 REM TEMPS\LARGE
       70 LET y=50: LET t=2: LET w=1
       80 LET a$="This is double height"
       90 REM TEMPS\LARGE
       100 LET y=100: LET w=2
       110 LET a$="Double size"
       120 REM TEMPS\LARGE
```

This will print in the three most used types of large text.

N.B. If any of the five variables are not defined then error "2 Variable not found" will result.

SOUND

This is the command that produces those amazing sound effects you may [have] heard in the demonstration program. The BASIC BEEP command is very limited in the sounds that it can produce. The SOUND command "fills in" the enormous sound making gap. Unlike the commands so far described, Sound does need parameters (4 in all) to describe what sound to make but it does not need a preceding dummy PRINT statement. You can give parameters in two ways:

a) As a "FIFTH" function (see the section on "FIFTH" functions). This is the least used way.
b) As a single letter variable. The variable must however be numeric and must not be a subscripted variable. If the variable is not defined then error "2 Variable not found" will be produced. You are not allowed to do any mathematics in a "FIFTH" REM statement; i.e. addition or subtraction.

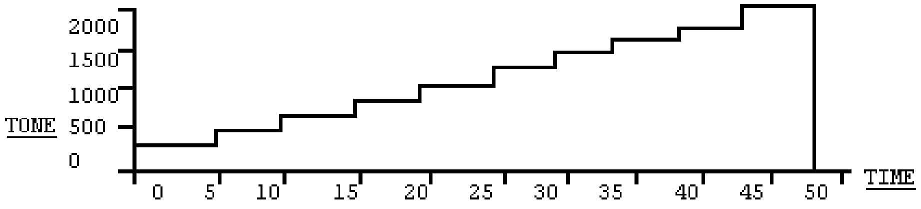
```
       100 REM SOUND a,b,c,d
```

This is the usual format for a SOUND command; each of the variables describe a different property of the sound:
VARIABLE a - THE REPEAT VALUE. It describes how many times a sound of length b and tone c should be produced and the current tone (initially c) should be added to d (the step) and the sound repeated before the particular sound statement has been finished.
VARIABLE b - The SOUND LENGTH. This describes the length of each component noise of the whole SOUND command.
VARIABLE c - THE SOUND TONE. This describes the starting pitch of the sound command. This has the variable d added to it after every repeat to find the new pitch (The NO. of repeats is determined by the a variable).
VARIABLE d - THE STEP VALUE. This is the value that is added to the last tone after every repeat to find the new tone; e.g.

```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 LET a=10: LET b=5: LET c=200: LET d=150:
       REM SOUND a,b,c,d
```

This produces a "phasor" like sound. The graph below shows how the phasor noise is made up.

Graph. FIG 2. - A closer look at the phasor noise.



As you can see, the phasor noise is actually made up of very short BEEPs, each one being slightly higher than the previous one. Now change the LET b=5 in line 30 to LET b=100. When you RUN the program now you can distinctly hear each individual tone. You may want the sound to decrease in pitch rather than increase. This is achieved by making d a number between about 65000 and 65535. This is because after adding d to the current tone it is taken to MOD 65536 (this means it divides by 65536 and takes the remainder, NOT the answer). Another way of looking at this is that if the number is bigger than 65535 then it subtracts 65536 from it. In other words, 65536 is the same as 0. For good sounds the variables should be within the ranges below:-
Variable A - Between 1 & 50 but this really depends on the sound length given in b. All the other numbers can have a range of 0 to 65535 inclusive except this one which must be 1 to 255 inclusive.
Variable B - Between 3 and 100
Variable C - Between 0 and 2000 if going up or between 2000 and 5000 if going down.
Variable D - Between 1 and 500 if going up or between 65000 and 65535 if going down.

Unlike during a normal BEEP statement, the SOUND command checks the BREAK key while producing the sound.

Examples:-

```
10 RANDOMIZE 1000
20 RANDOMIZE USR 61030
30 LET a=RND*20+3
40 LET b=RND*10+3
50 LET c=RND*1000
60 LET d=RND*200+50
70 IF RND>.5 THEN LET c=RND*1000+4000:
LET d=RND*50+65400
80 FOR e=1 TO 3-RND*10
90 REM SOUND a,b,c,d,
100 NEXT e
110 GO TO 30
```

Press BREAK to escape from this program. It makes random sound effects.
N.B. Although the variables a, b, c and d have been referred to throughout this description you do not have to use them. w, x, y and z could have been used just as easily. You can even use the same variable twice or even more times in a single SOUND command; e.g.

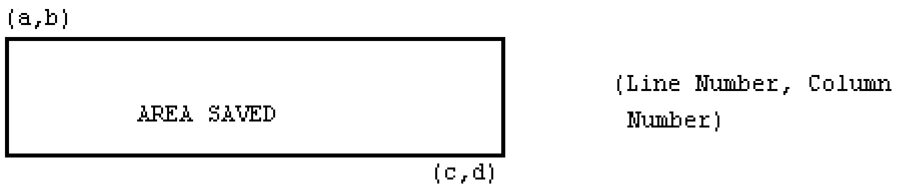
```
       100 REM SOUND e,a,e,z is perfectly legal.
```

<u>GET</u>

GET and PUT (described next) are used together. They are very powerful commands. They allow you to put part of the screen display into a BASIC string variable using the GET command and then PUT the information back onto the screen. GET has five parameters, 4 to tell it from which part of the screen to get the data from and 1 to tell it in which BASIC string the data should be stored. It is used in the form:-

```
       100 REM GET a,b,c,d,a$
```

a and c must be 0 to 21 inclusive and b and d 0 to 31 inclusive. As you can see GET uses PRINT positions, not high resolution co-ordinates. The reason for this is simple; the attributes (which are also saved by GET) are stored in character positions so it would be difficult to save the colours if GET was a high resolution command. The data which is stored is always in a rectangle, (a,b) being the top left-hand corner and (c,d) being the bottom right hand corner of the rectangle. In order for this to work properly c must be greater than or equal to a and d must be greater than or equal to b - otherwise you would have a rather strange rectangle, the right-hand side being more left than the left-hand side or the top being below the bottom! Fortunately, if co-ordinates do not make sense then error "B Integer out of range" will be produced.



<u>FIG. 3 - How the GET co-ordinates are arranged.</u>

The last parameter tells GET where to store the data. This should be a string from a$ to z$. If there was already a variable called a$ etc. then it is deleted and replaced by the new version. You cannot slice a string or use a subscripted string array; i.e.

```
       100 REM GET a,b,c,d,c$(2 TO 20) is not allowed.

       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 LET a=0: LET b=0: LET c=21: LET d=0
       40 REM GET a,b,c,d,x$
```

This puts the screen data in the left-most column of the screen in the variable x$. The only way to replace the data is to use the PUT command as the data is stored in a special format. Error "2 Variable not found" occurs if one or more of the first four parameter variables are not defined.

Examples:-

```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 LET a=0: LET b=0: LET c=21: LET d=31
       40 LET x=0: LET y=0: LET t=22: LET w=10:
       LET a$="GET"
       50 REM TEMPS\LARGE\GET a,b,c,d,a$
       60 FOR a=0 TO 31
       70 REM TEMPS\PUT b,a,a$
       80 LET b=b+0.6875
       90 PAUSE 50
       100 CLS: NEXT a
```

This draws "GET" in large letters across the screen and then proceeds to move it in a "south-easterly" direction. The PUT command (which is described next) is used in this example program.

PUT

This command, which is used in conjunction with GET, allows you to put data from a string back onto the screen after it has been collected by GET. It has 3 parameters; the first two tell it where to put the data on the screen and the third one tells it in which string the data to be used is. Unlike GET, this command must be preceded by a dummy PRINT statement or a TEMPS command. PUT is used in the form:-

```
       100  REM PUT x,y,a$
```

x must be in the range of 0 to 21 and is the line number. y, on the other hand, must be in the range of 0 to 31 as it is a column number. If you do not keep the numbers within these ranges then the computer will show its ingratitude by responding with error "B Integer out of range". The string variable must have been previously defined in a GET command. If no such simple string exists or it was not defined in a GET command (the computer has ways of telling whether it was or not) then error "2 Variable not found" will be produced. Under normal circumstances you would precede the PUT command with a TEMPS instruction unless you wanted to do something special with the colours. Only "colours" 8 & 9 are useful in PUT dummy PRINT statements; their use is explained below:-
COLOURS 8 - The particular colour type is left as it was previously (INK, PAPER, BRIGHT or FLASH). Normally, the colours that were saved by the GET command would be PUT back onto the screen. COLOUR 8 is a way [to] circumvent this behaviour; e.g.

```
       100 PRINT PAPER 8, INK 8; BRIGHT 8; FLASH 8;
```

in front of a PUT command; this dummy PRINT statement would ensure that the screen colours would remain the same.
COLOUR 9 - This is not as useful as colour 8.

```
       100 PRINT INK 9;
```

This will not change the on-screen PAPER colour but will ensure that the INK colour is contrasted with it.
INVERSE and OVER are also very useful in PUT dummy PRINT statements;
e.g.
INVERSE 1 - Will print the display in inverse to the way in which it was collected by GET.
OVER 1 - Normally, PUT will obliterate the display that was already on the screen. By using OVER 1 the two displays will be merged together; see page 113 of the Sinclair BASIC Programming manual.

```
       100 PRINT OVER 1; INVERSE 1;
```

This leaves the display exactly as it was previously but does change the on-screen colours (attributes).

If the data stored in the string takes up the whole screen, say, then when you use PUT it will not fit completely on the screen unless you start printing at (0,0). In fact the PUT command will only put on as much as it can, anything else is left unprinted. GET AND PUT also allow you to have a limited form of animation. By drawing each frame and getting them into different strings you can rapidly go through them using the PUT command. Be warned, however, that the memory requirements for this can be quite considerable. A screenful of data needs 6K of RAM.

Examples:-

```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 PRINT "This is a downward scroll"
       40 PAUSE 10
       50 GO SUB 8000
       60 GO TO 40
       8000 LET a=0: LET b=0: LET c=21: LET d=31
       8010 REM GET a,b,c,d,a$
       8020 LET a=1
       8030 REM TEMPS\PUT a,b,a$
       8040 PRINT AT 0,0;"(32 spaces)"
       8050 RETURN
```

This demonstrates how you can use GET and PUT to do a downward scroll. The subroutine at line 8000 actually does the scrolling. You can also fix it to do rightward scrolling but NOT upward or leftward scrolling. Can you think why?

LET

This is almost exactly the some as the BASIC LET statement. The main difference is that the variable you are defining must be a single letter, non-subscripted, numeric variable. Its format is:

```
       100 REM LET a=("FIFTH" expression)
```

The a represents the variable that you are defining. The "FIFTH" expression would usually be a "FIFTH" function. You could have a variable as with normal parameters but this could be done with a BASIC LET statement. You may be wondering what use all this is. It allows you to access the "FIFTH" functions in BASIC and then do calculations on them (which you cannot do in "FIFTH"); e.g

```
       100 REM LET x=COLUMN invader
       110 LET x=x+12: REM MOVE invader,x,LINE invader
```

This uses the "FIFTH" LET command and BASIC calculation to make the invader jump 12 pixels right.

Examples:-
```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 LET a=1: REM OBJECT bomb,a
       40 REM PRINT bomb,v
       50 REM LET t=SCREEN bomb
       60 PRINT CHR$ t
```
Lines 50 & 60 use the LET command to find out what the bomb is printed as.

OBJECT

Up until now the commands have not been really related to one another. The OBJECT command is the basis for most of the remaining commands. The real power of "FIFTH" is its ability to define objects using the OBJECT command. The objects can move about the screen completely independently of BASIC. The BASIC you would have to laboriously erase and reprint an object to move it around the screen. This is slow as well as being impractical, if you wish to move more than a couple of objects simultaneously. With "FIFTH" all you have to do is tell the computer the following information:-
a) What the object is to be printed as; e.g. The letter "A" or a full stop.
b) In what direction is the object going to go in; e.g. up, down or left ("FIFTH" allows 16 different directions).
c) In what colour the object is to be printed; e.g. Red, Yellow or Green.
d) The speed at which the object is to move. In "FIFTH" you can also define how many pixels an object will jump at any one time. An object can jump as small as one pixel; this is 8 times smoother than BASIC graphics.
Then "FIFTH" can move the object around the screen. When the object goes off the edge of the screen or collides with another object then a special user-defined service routine is called, but this is described later. "FIFTH" allows you to give objects names. This makes programming easier than if you had to quote a long number every time. If you were writing a space invader type program you may want 40, say, invader objects. It would be very difficult to refer to each one individually. Fortunately "FIFTH" has a solution to this problem. Like a BASIC array you can define a number of objects with the same name. "FIFTH", however, is more flexible in

the way you can access them. You can either concentrate operations on on individual or collectively on the whole group. Some pieces of object information can be unique to each subscript. In fact only colour and the "print as" character have to be the same for all subscripts. You may remember that the RANDOMIZE 1000 at the beginning of [a] "FIFTH" program tells the computer how much memory to reserve for the object data. One thousand bytes is usually used because it is large enough for just about any application. To work out exactly how many bytes you will need use the below method:

No. of letters in the name + 10 + 6 x No of subscripts.

From this you can see that 10 invaders would take:

7 + 10 + (6 x 10) = 77 bytes - not very much.

An object command has the format:

```
100 REM OBJECT (name),("FIFTH" Expression).
```

The name can be any length and can contain any character (Except ":"). It is best, however, to stick to letters and numbers (Alphanumeric characters). The "FIFTH" expression is evaluated and tells "FIFTH" how many subscripts with that name you want. This can be anything from 1 to 255 inclusive. If you define an object with the same name as one already defined then "FIFTH" will not take any notice of the new version. It will, in fact, still store the new object data, so defining objects with the same name is just a way of wasting memory. You may be wondering where "FIFTH" keeps all this data. What it does is to lower RAMTOP (which is initially set at 61029) by the amount given in the first RANDOMIZE statement. For instance, after running one of the example programs, type:

```
PRINT PEEK 23730+256*PEEK 23731 (This finds RAMTOP)
```

This will reply with 60029 which is 1000 bytes lower than the initial setting of 61029. The initial RANDOMIZE instruction can have any argument except 0. If there is not enough room in the computer's memory then it will reply with error "4 out of memory".

Now for an example:-

```
10 RANDOMIZE 1000
20 RANDOMIZE USR 61030
30 LET a=100: REM OBJECT Missile,a
```

This will define 100 objects called "Missile". Now change the 100 in the line 30 to 250. When you RUN it this time, the computer responds with error "4 out of memory". This is because there is not enough room reserved for 250 objects. Change the 1000 in line 10 to 10000.

This reserves nearly 10K of memory for the objects - more than enough. When an object is defined, you have not given information about speed, direction, etc. The computer has to make them up. The data it uses is listed below:-

a) CURRENT SUBSCRIPT	- Set to ALL
b) COLOUR	- Set to the current universal colour
c) PRINT	- A Space character (CHR$ 32)
d) DIRECTION	- Direction 0 (Upwards)
e) SPEED	- Moves once every five seconds (250 interrupts). It makes one pixel jumps.
f) SCREEN POSITION	- It is ENABLED but is on Line 176, Column 0
g) ERASE STATUS	- It will be overprinted under any circumstances but this is possible with the above parameters anyway.

N.B. (Until you have read the rest of the manual you will not be able to understand this).

Examples:-

```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 LET a=10: REM OBJECT invader,a
       40 REM PRINT invader,x
       50 REM DISABLE invader
       60 FOR a=1 TO 10
       70 REM USE invader,a
       80 LET b=84: LET  c=20+20*a: REM MOVE invader,
       c,b
       90 NEXT a
```

This will print ten Xs across the centre of the screen. Actually these are the ten "invaders" defined in line 30. The program uses a variety of commands not explained yet to achieve this result.

Note:-

a) Any commands (or functions) that need an object name (EXCEPT OBJECT) can be given a string variable name instead. This, as usual, must not be subscripted or sliced; e.g.

```
    100 REM PRINT invader,A
```

and

```
    100 LET a$="invader": REM PRINT a$,A
```

Both have the same meaning.

If the string variable is not defined then error "2 Variable not found" will be produced.

b) If you use a name that has not been defined in an OBJECT command then error "a Invalid Argument" will result.
c) When referring to or defining object names it makes no difference whether you use upper or lower case; e.g:-
PLANE, plane, Plane and pLAne all refer to the same object type.

USE

This is one of the commands which determines whether a particular object type will have its subscripts accessed individually or as a whole group. This is the command which will allow you to access subscripts individually. Its format is:-

```
       100 REM USE (object type name),("FIFTH" expression)
```

The object name is that of the object type that you wish to access on a single subscript basis in succeeding operations. The "FIFTH" expression tells the computer which individual subscript you wish to use; e.g. if you defined 10 objects called a "torpedo" then this "FIFTH" expression can be evaluated to a number between 1 and 10. If the number is not between 0 and 255 inclusive the error "B Integer out of range" will be produced. If the number is then bigger than the number of subscripts (in this case bigger than 10) then error "6 Number too big" will be the result.

```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 LET a=10: REM OBJECT invader,a
       40 LET a=7: REM USE invader,a
```

This defines an object called "invader" and then sets the "CURRENT" subscript to 7. If you change the LET a=7 in line 40 to LET a=11 then when RUN, the program will stop with error 6. This is because you have tried to use a subscript that does not exist. Now change the LET a=10 in line 30 to LET a=11 and the program will work all right again.
You may think that if you use 0 for the second parameter then some error would be produced because subscripts start at 1. In fact this is not the case as:-

```
       100 LET a=0: REM USE bomb,a (or something similar)
```

has the same meaning as ALL, which is described next.

Examples:-

```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 LET a=100: REM OBJECT rocket,a
       40 REM PRINT rocket,X\DISABLE rocket
       50 FOR a=1 TO 100
       60 REM USE rocket,a
       70 LET x=INT (RND*256): LET y=INT (RND*176)
       80 REM MOVE rocket,x,y
       90 NEXT a
```

This randomly positions 100 Xs around the screen. It uses a host of unexplained commands but notice how the USE command is utilised in line 60. The program goes through all 100 rocket's individually and places each at a random position.

ALL

This is the command which allows you to access all subscripts of a particular object type at the same time. It is used in the form:-

```
       100 REM ALL (Name of object type)
```

Unlike USE, ALL does not need a second parameter to tell it which subscript you want to USE as it assumes you want to access all subscripts. After using ALL, every operation you do to that particular object type will be done to every subscript; i.e. If you did an operation to move a "rocket" to position (231,67) then if ALL had been used on the "rocket" object type all subscripts would move to (231,67). Conversely, if USE had been carried out on "rocket" then only the selected individual would move to (231,67) and all the others would stay where they are.

```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 LET a=10: REM OBJECT invader,a
       40 REM DISABLE invader\PRINT invader,H
       50 FOR a=1 TO 10
       60 REM USE invader,a
       70 LET x=RND*255: LET y=RND*175
       80 REM MOVE invader,x,y
       90 NEXT a
       100 REM ALL invader
       110 PAUSE 0
       120 LET y=176: REM MOVE invader,a,y
```

If you RUN, ten Hs will appear at random positions on the screen. If you press a key then they will all disappear. Now change line 100 to:-

```
       100 LET z=10: REM USE invader,z
```

If you RUN the program again and press any key then only one H will disappear. This illustrates the power of ALL and USE. Try and account for the difference in the two RUNs.

N.B. Although a USE command with a second parameter with a value of 0 is the same as an ALL command, it is good programming practice to use ALL except in situations where it is much more "elegant" to utilise the first method. A good use for this property of USE is given later in this manual.

PRINT

This is the command which describes what character an object will print as. This can be any character, including graphic symbols and user-definable graphics. It has the form:-

```
       100 REM PRINT (object name), (Character)
```

If no such object with the name you give exists then the computer will respond with error "A Invalid Argument". If the character you give is a token (anything with more than one character in it; i.e. PRINT or SCREEN) or unprintable (anything with a code below 32) then error A will result. The only other character that you cannot use is the space (CHR$ 32). There is however a graphic symbol which is the same as a space. It is accessible on key 8 in GRAPHICS mode. An individual subscript cannot have its personal "print as" character. In fact, every subscript in each object type must be printed as the same character. This also means that the PRINT command is not affected by ALL or USE commands.
Examples:-

```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 LET a=1: REM OBJECT missile,a\DISABLE missile\
       PRINT Missile,s
       40 LET x=124: LET y=84: REM MOVE missile,x,y
```

Changing the second parameter of the PRINT command in line 30 to other characters.

COLOURS

This command determines what colour an object is printed in. Like PRINT, only one set of colours can be defined per object type. The command must be preceded by a dummy PRINT statement. This is the colour that the object will be printed in. You could of course use a TEMPS command if you wanted the object to be printed in the current universal colour. A colour command is used in the format:-

```
       100 REM COLOUR (Name of object type)
```

"Colour" 8 has its usual meaning although "colour" 9 is not used by the COLOUR command.

Examples:-

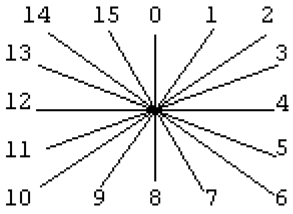
```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 LET a=1: REM OBJECT invader,a
       40 REM DISABLE invader\PRINT invader,X
       50 PRINT INK 2: REM COLOUR invader
       60 LET x=124: LET y=84: REM MOVE invader,x,y
```

Try changing the dummy PRINT statement in line 50 to print the "X" in different colours.

VECTOR

This is the command which decides the direction in which an object will move. "FIFTH" gives you a choice of 16 directions, each one signified by a number between 0 and 15 inclusive. Direction 0 is an upward direction, 1 is slightly rightward to this (or eastward). This scheme carries on until direction 15 which is just leftward to direction 0.

FIG.4 - How the VECTOR directions are arranged.



A VECTOR command has the format:-

```
       100 REM VECTOR (name of object type), (new direction)
```

Unlike PRINT or COLOUR, this command takes notice of ALL and USE. If the "Current" of the object type is ALL then all subscripts will have their direction changed. If, however, the current is USE then only the direction of the individual selected in the USE command will have its direction altered. If the direction value is not between 0 and 15 inclusive then error "B Integer out of range" will result. If the direction value is not a whole number then it is rounded to the nearest one. Even if an object is disabled (see the disable command) then the direction will be changed but will not take effect until the object is re-enabled. The same will happen if the object is at an "Off-screen" position (described under the MOVE command).

Examples:-
```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 LET a=1: REM OBJECT ball,a
       40 REM PRINT ball,o
       50 REM SPEED ball,a,a
       60 LET a=8: REM VECTOR ball,a
       70 LET x=124: LET y=84: REM MOVE ball,x,y
       80 GO TO 80
```
When you RUN this program an "o" appears at the centre of the screen and moves downwards until it goes off the edge of the screen. Press BREAK. Change the value of a [at] line 60 to get an idea of the different directions of the VECTOR command. The program uses a variety of unexplained commands to achieve this end. Lines 30-60 just set up the object called ball so that the computer has enough information to move it around the screen which it does after it has been positioned in the middle of the screen by line 70. Line 80 is necessary to allow the object to move. This is because if an error report is produced (i.e. in this case error "8 End of file") then all moving objects on the screen will come to a halt.

SPEED

This is the command which allows you to change the speed at which an object moves on the screen. "FIFTH" allows a tremendous choice of speeds but more important it determines how smooth the graphics of a program will be. "FIFTH" allows you to define how many pixels an object will jump at any one time. A speed command has the format:
```
       100 REM SPEED (Name of object type),("FIFTH" expression),
       (expression)
```
This command, like VECTOR, is affected by ALL and USE. The first "FIFTH" expression describes the delay in 1/50 of a second before the object is moved, and must be between 1 and 255 inclusive. Fifty times a second, the Spectrum's Z80A processor receives an "Interrupt" from Sinclair's ULA chip which is also inside that black case of your Spectrum. This interrupt signal tells the processor to read the keyboard and increment the frames counter. However, when using "FIFTH" the processor also has to move "FIFTH" objects. This expression, therefore, tells the "FIFTH" system how many interrupts it has got to wait through before it is that particular object's turn to be moved. As you can see a value of one would mean that the object would be moved every time and therefore 50 times a second. If the value was 50 then the object would only be moved once every second.

If the value was 2 then the object would be moved 25 times a second. From this it can be seen that the formula:

$$\text{Number of movements per second} = \frac{50}{\text{The value of the first expression}}$$

can be made.

This is all very well but there is a trade off between speed and how fast BASIC is. Since the objects are moved at times when the BASIC program would be normally executed it can be seen that the more often an object is moved, the slower BASIC will be. This is especially true if a large number of objects are being moved. Fortunately, "FIFTH" has a remedy to this difficult problem. It comes in the form of expression NO.2 which must also be in the range of 1 to 255 inclusive. It tells "FIFTH" how many pixels it should move the object every time it is its turn to be updated during an interrupted response. Most of the smooth graphics that you may have seen in the demonstration program were done using a movement of one pixel 50 times a second. These are the smoothest graphics possible but are expensive on the speed of BASIC if you have more than a few of these objects moving simultaneously. It is a good idea, however, to try and use these smooth graphics whenever possIble as the effect produced by the objects "floating" across the screen can be quite incredible. If you want to keep the same speed but do not want the connected slowness of BASIC then increase the number for both the first and second parameters. For instance, the parameters may become equal to 2,2 after previously being 1,1. The object will still have the same overall speed but its movement will be a little more "jumpy" (but still 4 times better than BASIC movement), instead of moving one pixel 50 times a second it will be moving 2 pixels 25 times a second. But this is really a small price to pay for the speed increase in BASIC. If you just increase the size of the second parameter without the first then the overall speed of the object will be increased. Now for a practical example:-

```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 LET a=1: REM OBJECT thing,a
       40 REM PRINT thing,<
       50 LET d=12: REM VECTOR thing,d
       60 LET a=1: LET b=1: REM SPEED thing,a,b
       70 LET x=255: LET y=84: REM MOVE thing,x,y
       80 GO TO 80
```

The SPEED command is used in line 60. On the first RUN, a "<" will move across the screen from right to left. This is at the smoothest speed. Change the values in line 60 to get the "hang" of the SPEED command. You should now know what every command in the listing does except the move command, although its use should be quite obvious.

Examples:-

```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 LET a=10: REM OBJECT cars,a\DISABLE cars\PRINT cars,^
       40 FOR a=1 TO 10
       50 REM USE cars,a
       60 LET b=1: REM SPEED cars,a,b
       70 LET y=170: LET x=20*a: REM MOVE cars,x,y
       80 NEXT a
       90 REM ALL cars\ENABLE cars
       100 GO TO 100
```

This program places 10 "^"s at the bottom of the screen. Then they start moving but everyone is slightly slower than the previous one, looking from left to right. The SPEED command is used in the USE mode in line 60.

MOVE

This command is used to move a particular object to a given position. It is used in the form:-

```
       100 REM MOVE (Name of object type),(x coordinate),(y
coordinate)
```

Like VECTOR and SPEED, MOVE is affected by ALL and USE. The x co-ordinate is a "FIFTH" expression which must have a value between 0 and 255 inclusive. If the expression evaluates to a non-integer then It is rounded to the nearest one. The y co-ordinate must also have a value of 0 to 255 inclusive although numbers bigger than 175 do not have the usual meaning. As you may remember, "FIFTH" Hi-res co-ordinates have them starting from the top of the screen so that (0,0) is the top left-hand corner of the screen. This is in contrast with the BASIC PLOT, POINT and CIRCLE statements which have their y co-ordinate starting from the bottom of the screen. There are 176 possible y co-ordinates as there are 22 lines each of 8 pixel height (22 x 8 = 176). The two bottom lines of the display cannot have objects printed on them but they provide a useful area to print scores and times etc. as there is no danger of them being overwritten by the moving objects. If, say, the y co-ordinate was 175 or the x co-ordinate was 251 then there would not be enough room to fit the whole of the 8 x 8 character on the screen. "FIFTH" only prints on as much as it can; anything else is left unprinted. This gives the impression that there is an area off the screen that cannot be seen. It is as if the screen is just a window on a larger area. The co-ordinates always refer to the top left-hand corner of the character. If the y co-ordinate is greater than 175 then the object is not printed anywhere on the screen. The object will cease to be moved by interrupts and therefore the only way to make it reappear on the screen is to use another MOVE instruction. Please note that MOVE does not implement an interact response cycle even if another object was collided with. You have to do this manually by using the FIND command to check the position that you are going to move the object to. (You will probably not understand this until you have read the rest of this manual). A MOVE instruction automatically erases the old image of the object.

```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 LET a=1: REM OBJECT object,a
       40 REM PRINT object,0\DISABLE object
       50 LET x=100: LET y=100: REM MOVE object,x,y
```

This program sets up a single object called "object" (confused) and then proceeds to move it to position (100,100). If you change the values of x and y in line 50 then you can get some idea of the way in which MOVE works. Try positioning the object near the bottom of the screen.

Examples:-

```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 LET a=1: REM OBJECT arrow,a
       40 REM DISABLE arrow\PRINT arrow,>
       50 LET y=84
       60 FOR x=0 TO 255
       70 REM MOVE arrow,x,y
       80 NEXT x
       90 GO TO 60
```

This program illustrates how you can use the MOVE command to give you animation. This is obviously not as goad as "FIFTH" automatic movement but at least BASIC is not slowed down. You can vary the speed of the movement by introducing a STEP statement in the FOR instruction at line 60. For Instance, a STEP 2 doubles the speed of the movement.

Note:-
When you first define an object it is moved to position (0,176). As you can see, this is on off-screen position. This is to prevent the object immediately appearing on the screen after definition when perhaps you do not want it to. After setting up the parameters of the object it can be moved into the active screen area using a MOVE command.

RMOVE

This is similar to the MOVE command. The R stands for relative MOVE. This works in a way similar to Sinclair BASIC's DRAW command. You do not give an actual screen position but one to be added to the existing one. A RMOVE has the format:-

```
100 REM RMOVE (Name of object type), (relative x),(relative y)
```

The name of the object type to be used works in exactly the same way as in the MOVE command. The x relative co-ordinate is added to the current x co-ordinate to produce the new one. This relative x co-ordinate must be in the range of 0 to 255 inclusive. The relative y co-ordinate works in exactly the same way and must be in the range 0 to 175 inclusive. The co-ordinate are in a "wrap-round" form so if you add 1 to an x co-ordinate that was previously 255 then the new x co-ordinate would be 0. The y co-ordinate is also "wrap-round" but wraps at 176. For instance if you added 1 to a y co-ordinate that was previously 175 then the new y would not be 176 but 0. This also means that you cannot move objects to off-screen positions using the RMOVE command, you have to use MOVE. One Problem with "FIFTH" expressions is that you cannot have negative numbers. The RMOVE command works alright as long as you are moving down or right but consider what happens when you try to move left or up. The DRAW statement allows negative arguments but a "FIFTH" expression does not. The solution to this problem is similar to that used for decreasing pitch in the SOUND command. If you want to move the x co-ordinate left then use the formula: -
Number to use as first parameter = 256 - number of steps left
This works as long as you do not want to move 0 pixels left, but this is not really moving left anyway. The formula for the y co-ordinate is:-
Number to use as second parameter = 176 - number of steps up
so for instance, to move 2 pixels up and 1 right the values would be (1,254). Like the move command an interact service routine is not called if the move would mean a collision with another object.

You have to do this manually using the FIND command. A limit service routine is also not called if the move means the object going off the edge of the screen.

Examples:-
```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 LET a=1: REM OBJECT arrow,a\PRINT arrow,>\
       DISABLE arrow
       40 LET x=0: LET y=84: REM MOVE arrow,x,y
       50 LET x=1: LET y=0
       60 REM RMOVE arrow,x,y
       70 GO TO 60
```
This example program is a modified one of that used for the MOVE example program, It uses the RMOVE instruction to move the arrow across the screen. To increase the speed of the arrow, increase the size of the x in line 50. Change the LET y=0 in line 50 to LET y=1 and the arrow will move diagonally.

FIND

This command is used to determine whether there is an object at a given screen position. It has the form:-
```
       100 REM FIND (x co-ordinate), (y co-ordinate)
```
The x co-ordinate must be in the range of 0 to 255 inclusive and the y must be in the range of 0 to 175 inclusive. The name of the object type is returned in the BASIC variable j$ and the number of the subscript is returned in the BASIC variable j. If, however, there is no "FIFTH" object at that position then j$ will be the null string (a string containing no characters - "") and j will have the value 0.

```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 LET a=1: REM OBJECT invader,a\PRINT invader,A\
       DISABLE invader
       40 LET x=1: LET y=100: REM MOVE invader,x,y
       50 REM FIND x,y
       60 PRINT j$,j
```
This should print "invader" and "1" at the top of the screen. If you miss out the PRINT command in line 30 then the routine will still work. In fact, the FIND command does not look at the screen at all. It simply goes through the co-ordinates of all the objects and uses the first object that seems to [be] near enough to the position you gave. The name of the object type, which is given in j$,

is always given completely in lower case letters, even if the object was defined completely in upper case.

Examples:-

```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 LET a=10: REM OBJECT abc,a\PRINT abc,H
       40 LET a=1: REM SPEED abc,a,a
       50 LET a=1+RND*9: REM USE abc,a
       60 LET x=124: LET y=175: REM MOVE abc,x,y
       70 LET y=40
       80 REM FIND x,y
       90 PRINT AT 5,0;j$;"(3 spaces)";j
       100 GO TO 80
```

This program defines 10 subscripts of "abc". It then randomly selects one of these and moves it up the screen. One of the positions it has to pass through is monitored by a FIND command whose results are printed on the screen. From the results you should be able to determine which subscript it was.

DISABLE

This command gives you the facility to stop an object moved by interrupts. It is used in the form:-

```
       100 REM DISABLE (Name of object type)
```

Like most of the commands connected with "FIFTH" automatic movement, this command is affected by ALL and USE. The given object will be disabled as far as automatic movement is concerned. The object must be re-enabled using an ENABLE instruction for movement to continue. All other commands such as MOVE still work as normal. Commands that just give information (e.g. PRINT or COLOUR) also still work alright although their effect is not shown until the object is re-ENABLED. When first defined, an object is enabled. The only time that it is disabled (except manually in a BASIC program using DISABLE) is after it collides with another object or goes off the edge of the screen in which case an appropriate service routine is called which would usually re-enable it anyway.

```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 LET a=1: REM OBJECT ball,a\PRINT ball,O
       40 REM SPEED ball,a,a
       50 REM DISABLE ball
       60 LET x=124: LET y=170: REM MOVE ball,x,y
       70 GO TO 70
```

When RUN this program will print most of an "O" at the bottom of the screen. If you now delete line 50 then the "ball" will then move up the screen. This is because the DISABLE instruction at line 50 prevents the "ball" from moving. Remember that if you DISABLE an object then it will not be erased from the screen and can still, therefore, be involved in a collision with another object.

Examples:-

```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 LET a=1: REM OBJECT ball,a
       40 REM PRINT ball,O\SPEED ball,a,a
       50 LET x=124: LET y=165: REM MOVE ball,x,y
       60 IF INKEY$="0" THEN REM DISABLE ball
       70 IF INKEY$="1" THEN REM ENABLE ball
       80 GO TO 60
```

This is a modified version of the last program you may have typed in. You can start or stop the ball at any time as it travels to the top of the screen. Key "0" stops the ball and key "1" starts it again. N.B. If you try and DISABLE an object which is already disabled then the command will have no net effect.

ENABLE

This is the complementary command to DISABLE, as you might expect. It has a similar format to DISABLE; i.e.

```
       100 REM ENABLE (Name of object type)
```

Everything that was explained about the DISABLE command is applicable to the ENABLE command. Except, of course, the object is enabled instead of disabled.

Examples:-

The example given for the DISABLE command will also function as on example of the ENABLE command, which it contains in line 70.

LIMIT

This command is used to define which line number will be jumped to if an object goes off the edge of the screen. "FIFTH" is very powerful in the sense that it will automatically jump to a certain line number when an object reaches the edge of the active screen area. It will perform something similar to a BASIC GO SUB but no GO SUB instruction is needed in the main loop of the program. In fact the service routine (the BASIC routine that is called when a limit condition occurs) must be terminated with CONTINUE, not RETURN. Most of this, however, is explained under the LMTPARAM command. It is up to the programmer to write a short service routine at the given line number to handle the limit condition. A LIMIT command has the form:-

```
       100 REM LIMIT ("FIFTH" expression)
```

The "FIFTH" expression gives the line number to be jumped to if a limit occurs and must evaluate to between 0 & 65535 inclusive otherwise error "B Integer out of range" will result. If the expression comes to more than 9999 (the highest number possible) then when a limit condition occurs, no line will be jumped to and so program execution will carry on as normal.

Now for an example:-

```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 LET a=1000: LET b=1: REM LIMIT a\OBJECT thing,b
       40 REM PRINT thing,^\SPEED thing,b,b
       50 LET x=124: LET y=175: REM MOVE thing,x,y
       60 LET a=0
       70 GO TO 60
       1000 STOP
```

Programs similar to this already used as examples in this manual do not stop when the "^" reached the top of the screen. This program however, does stop with error "9 STOP statement". The stop statement is at line 1000. As you can see, there is no actual statement to jump [to] line 1000 in the main loop of the program. This is caused by a limit condition occurring when the object tries to go off the screen. You may wonder what the seemingly redundant line 60 is doing in the program. This is needed because "FIFTH" cannot jump to a service routine if the main loop of the program consists of a single GO TO statement which jumps to itself. There are other rules concerning this to be complied with but these are explained under the LMTPARAM command. When you first execute the "RANDOMIZE USR 61030" at the beginning of the program, the limit line is set at 10000 so that if a limit condition occurs then no service routine will be called (mainly because the computer does not know whether you have written one or not!).

Examples: -

```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 LET a=8000: LET b=1: REM OBJECT ball,b\LIMIT a
       40 REM PRINT ball,o\SPEED ball,b,b
       50 LET x=128: LET y=88: REM MOVE ball,x,y
       60 LET a=0
       70 GO TO 60
       8000 REM LMTPARAM
       8010 IF i=0 THEN LET z=7
       8020 IF i=1 THEN LET z=11
       8030 IF i=2 THEN LET z=15
       8040 IF i=3 THEN LET z=3
       8050 LET z=z+INT (RND*3)
       8060 IF z>15 THEN LET z=z-16
       8070 REM VECTOR ball,z\ENABLE ball
       8080 CONTINUE
```

This program produces an "o" bouncing around the screen; do not worry about how it works at the moment.

<u>INTERACT</u>

This is similar to LIMIT but determines the line that will be jumped to when two objects collide with each other. Its form is:-

```
       100 REM INTERACT ("FIFTH" expression)
```

Most of the details are the same as for the LIMIT command. The following information applies both to the LIMIT and INTERACT commands. Before a service routine can be jumped to, the interpreter must finish the statement that it was executing when the collision happened. This means that "FIFTH" must temporarily store the information pertaining to the collision or limit condition. There has to be room for more than one lot of information in case a lot of events all happen at once. It can now be told that the temporary store is the "service stack". In fact there are two of these, one for LIMITs and one for INTERACTs. There is room for 16 outstanding service routine calls in each stack. If more than 16 become outstanding at one time then error "4 Out of memory" results. This error is a bit strange as it can occur at any line since the objects are being moved and collisions etc. are being stored at the same time BASIC is running. The rule is to look into the possibility that the error was caused by an overflow of the service stacks before spending ages looking for a non-existent error in the actual program. Under normal circumstances the service stacks should never overflow unless you are doing something terribly wrong. The service stacks are LIFO (Last in - First out) structures so that the last object condition that happened is always the first one to be processed. As well as this, interacts "have priority over limits" so before a limit condition is seen to, there must be no outstanding interact conditions. "FIFTH" does not allow "nested" service routines - so another service routine will not be called while another is in progress. The interpreter knows that it's finished the service routine when it comes to the delimiting CONTINUE statement.

Examples:-
```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 LET a=1000: LET b=2: REM INTERACT a\OBJECT bomb,b
       40 REM PRINT bomb,O
       50 LET a=1: REM SPEED bomb,a,a\ERASE bomb
       60 REM USE bomb,a
       70 LET c=8: REM VECTOR bomb,c
       80 LET x=124: LET y=0: REM MOVE bomb,x,y
       90 REM USE bomb,b
       100 LET y=175: REM MOVE bomb,x,y
       110 LET a=1
       120 GO TO 110
       1000 REM INTPARAM
       1010 BEEP 2,-30
       1020 REM ALL bomb\ENABLE bomb
       1030 CONTINUE
```
This program defines two objects hurtling towards each other. When they meet, a series of low frequency BEEPs are produced.

LMTPARAM

LMTPARAM stands for limit parameters. The command is used in limit service routines to assign BASIC variables with information about the limit condition. This command has no parameters after it. When it is executed it returns the name of the object type that went off the screen in the BASIC variable h$. This will always be composed entirely of lower case letters so remember that when you do tests on this variable. The actual subscript of the object type that went off the screen is returned in the BASIC variable h. The direction that it went off the screen is returned in the variable i. This is 0 if it went off the top of the screen, 1 for the right-hand side, 2 for the bottom and 3 for the left-hand side. The co-ordinates the object had just before it went off the screen are kept - the object is not erased and still remains on the screen. What happens in fact, is it is disabled to stop it causing another limit condition on its next move. You must remember, however, that the object may not be near the edge of the screen if you gave a particularly large number for the second parameter of its SPEED command. It is the job of the service routine to "point" the object in another direction or do something appropriate and then re-enable the object. Even if you are not going to use the information given by a LMTPARAM command, you must still put it in. It is best, therefore, to always put it as the first line of your service routine. A service routine is finished when the interpreter comes to a CONTINUE statement; this would be the last line of your service routine. As was pointed out in the description of the INTERACT command, another service routine will not be called while one is being executed. What happens, therefore, if the limit service routine marks the end of that particular part of the program and no CONTINUE statement is needed? The answer to this is to add the line "POKE 23681,0". This tells "FIFTH" that the service routine is finished, just like a CONTINUE statement would.

Now for a working example:-
```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 LET a=2000: LET x=1: REM OBJECT ball,
       x\LIMIT a
       40 REM SPEED ball,x,x\PRINT ball,O
       50 LET x=124: LET y=100: REM MOVE ball,x,y
       60 LET a=1: GO TO 60
       2000 REM LMTPARAM
       2010 LET b=(7 AND i=0)+(11 AND i=1)+
       (15 AND i=2)+(3 AND i=3)
       2020 LET b=b+INT (RND*3)
       2030 IF b>15 THEN LET b=b-16
       2040 REM VECTOR h$,b\ENABLE h$
       2050 CONTINUE
```

This is a similar example given under the LIMIT command. It produces a "O" bouncing around the screen. The main loop of the program is in line 60 which as you can see, is a "do nothing" loop. Lines 10 - 50 merely set up the object. From line 2000 onwards is the service routine. Notice how it starts with a LMTPARAM command and ends with a CONTINUE statement. Line 2010 makes the b variable equal to a suitable parameter for the VECTOR command but pointing in the order direction to which the object went off the screen, Line 2020 adds some randomness to the VECTOR selection, otherwise the object would just bounce the same way all the time. Line 2030 makes sure that this variable does not come to more than 15 and if it does it subtracts 16 - making direction 16 equal to direction 0. Line 2040 actually changes the direction. Notice now h$ is used for the name instead of [a] "real" name. In this particular example we know that h$ will always be assigned as "ball" but in other programs this may not always be the case. Here is [a] more sophisticated version of the same program; it moves 8 balls simultaneously:-

```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 LET a=6000: LET b=8: REM OBJECT ball,
       b\LIMIT a
       40 REM PRINT ball,O
       50 LET a=1: REM SPEED ball,a,a
       60 LET x=124: LET y=50: REM MOVE ball,x,y
       70 LET a=1: GO TO 70
       6000 REM LMTPARAM
       6010 LET b=INT (RND*3)+(7 AND i=0)+(11 AND i=1)+
       (15 AND i=2)+(3 AND i=3)
       6020 IF b>l5 THEN LET b=b-16
       6030 REM LET c=CURRENT h$\USE h$,h\VECTOR h$,
       b\ENABLE h$\USE h$,c
       6040 CONTINUE
```

This program contains a lot of interesting points. Here is a description of it:-

Lines 10-60: Set up the object type called "ball". Line 30 also sets the service routine at line 6000.

Line 70: The main loop of the program. As you can see this contains the seemingly redundant "LET a=1". This is because the main loop of the program must not contain a single statement; i.e. None of the below are legal:-

```
70 FOR a=0 TO 1000000: NEXT a (FOR is only executed once)
70 GO TO 70
70 IF 2>1 THEN GO TO 70 (Always goes back to line 70)
```

Line 6000: Gets the information necessary for the service routine.

Line 6010: Makes b equal to a suitable number for a VECTOR command.

Line 6030: This performs most of the work of the service routine. First of all, it makes c equal to the current subscript being used of h$. In this particular program this is not really needed but is incorporated to illustrate a point. As you can see the second command is a USE instruction. This changes the current of h$ which could spell disaster for the main loop of the program. A rule when writing service routines is to leave everything exactly as you found it. Be very careful about BASIC variables. Always use different variables in the service routine to those used in the main loop of the program. It is certainly very difficult to get used to service routines. Many program bugs can be attributed to using the same variable (s) in both the service routines and the main program loop. Also watch out for other things that you do in a service routine that may affect the main program execution.

Line 6040: This terminates the service routine.

In most programs you would want to do different things according to the object type that went off the edge of the screen. You could do this by a number of IF...THEN GO TO... statements near the beginning of the service routine; i.e.

```
       9000 REM LMTPARAM
       9010 IF h$="invader" THEN GO TO 8000
       9020 IF h$="bomb" THEN GO TO 8500
       9030 IF h$="missile" THEN GO TO 7000
       etc.
```

Examples:-

```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 LET a=8000: LET b=1: REM OBJECT arrow,b\LIMIT a
       40 REM PRINT arrow,>\SPEED arrow,b,b
       50 LET z=4: LET x=0: LET y=0: REM VECTOR arrow,z
       60 REM MOVE arrow,x,y
       70 LET a=1: GO TO 70
       8000 REM LMTPARAM
       8010 LET y=y+8: IF y>170 THEN STOP
       8020 REM MOVE arrow,x,y\ENABLE arrow
       8030 CONTINUE
```

This program produces an arrow moving from left to right. When it gets to the edge of the screen it goes back to the left-hand side but 8 pixels lower and continues the cycle.

<u>INTPARAM</u>

This command is very similar to the LMTPARAM except it is used for interact service routines. The main difference are the variables that it defines and their meaning. A collision obviously involves two objects. The name of the first object is given in h$ and its subscript number in h. The second object has its name returned in i$ and its subscript number in i. As usual, the names are given entirely in lower case, so remember this when performing tests on them.

When a collision occurs both of the involved objects are disabled. The two objects never actually touch each other. In fact although the characters are printed on an 8 x 8 pixel grid the "FIFTH" system tests the bordering pixels to see if they are set to the INK colour. If they are then "FIFTH" knows that it has collided with another object. One spin-off from this is that if one of the print as characters is equal to a space the collision can never occur because it will never be detected. If one of the things involved in the collision is not recognised as a "FIFTH" object then it is given the name "" (the null string) and has subscript number 0.

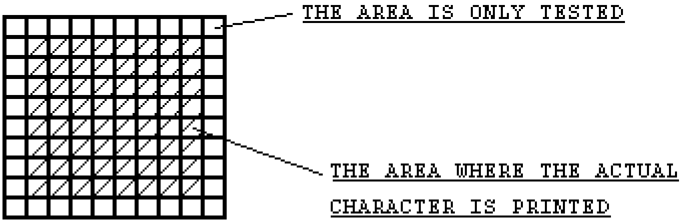
Examples:-

```
       10 RANDOMIZE 1000
       20 RANDOMIZE USR 61030
       30 LET a=5000: LET b=2: REM INTERACT a\OBJECT
       star,b
       40 LET a=1: REM PRINT star,*\SPEED star,a,a\
       ERASE star
       50 REM USE star,a
       60 LET x=0: LET y=150: LET z=4: REM MOVE
       star,x,y\VECTOR star,z
       70 REM USE star,b
       80 LET z=12: LET x=255: REM MOVE star,x,y\
       VECTOR star,z
       90 LET a=1: GO TO 90
       5000 REM INTPARAM
       5010 BEEP.05,50
       5020 REM USE h$,h\ENABLE h$\USE i$,i\ENABLE i$
       5030 CONTINUE
```

This program produces two horizontal "*"s moving in opposite directions. When they collide a BEEPING noise is produced as the interact service routine is called.
N.B. - When you write an interact service routine and want to test whether the BASIC variables h$ and i$ are the names of particular object types do it both ways; i.e.

```
100 IF (h$="bomb" AND i$="missile") OR (h$="missile" AND
i$="bomb")
THEN etc ...
```

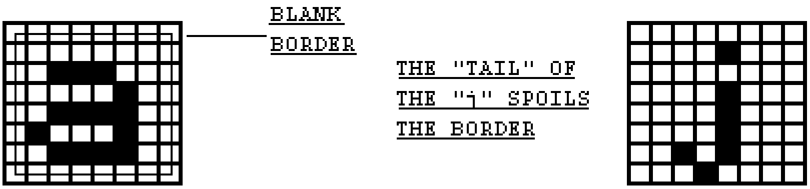
FIG. 5 - Where "FIFTH" looks for another object



ERASE

The usual way to move an object is to erase (print a space) the old image of the character and then print the character at the new position. This is all very well but is quite slow since the "FIFTH" system actually has to print two characters (the space to erase the old image plus the print of the actual character). Under certain conditions it is only necessary to print the new image as this will automatically erase the old image. The condition is that the "print as" character must have a border of paper pixels equal to the number of jumps per move of that particular object. The two diagrams below explain why the letter "a" can be moved in "non erase" mode with a jump of one pixel per move while the letter "j" cannot.

FIG. 6 - A comparison of the letters "a" and "j"



Whenever you use the PRINT or SPEED commands, "FIFTH" checks to see whether it is possible to put that particular object into non-erase mode. If this is possible then "FIFTH" will do so. All the ERASE command does is to make sure that the given object type is in erase mode, even if non-erase mode is possible. ERASE is affected by ALL and USE. Its format is:-

```
       100 REM ERASE (Name of object type)
```

You may be wondering what good this is as printing a character in erase mode uses more time and therefore slows the speed of BASIC. The answer is that there is a trade-off between the two. A character printed in non-erase mode "doesn't look where it's going" and therefore never detects a collision with another object. A non-erased character never takes part in an interact condition, therefore unless the other character is printed in erase mode and would therefore detect the presence of the first character. There is however another reason for using non-erase mode; it is explained below:-

The problem lies with the fact that the T.V. picture is being output at exactly the same time that the objects are being moved. The problem only rears its ugly head when you are moving more than a few erased characters at the same time and even so only when the characters are at the top of the screen. For the problem to reach its full extent the objects have to be moved every interrupt or 50 times a second. The first thing "FIFTH" does is to erase the character by printing a space over it. It sometimes just so happens that the ULA chip reaches that portion of the display file where the character is and outputs it to the T.V. at the exact moment. "FIFTH" did not have time to print the new version of the character so the object visibly disappears from the screen because "FIFTH" only had time to print the space over the old image. This effect does not happen with non-erased objects since they are never completely absent from the display at any time. Please note that although the object may not appear on the screen for a short period of time it is still in the display file. The answer to this problem is to try and use as few erased characters as possible and if you do use them keep them to the bottom of the screen as much as possible. Please note that you sometimes see another form of corruption where a character is only partly printed. This is caused by the ULA reaching that particular portion of the display when "FIFTH" is only part way through printing the new image of the character. (NB - Although the SPEED command may only refer to a single subscript, all subscripts of the given object type are tested for compatibility with non-erase mode.)

<u>THE "FIFTH" FUNCTION</u>

When a "FIFTH" command has a numeric parameter you can either put a BASIC variable there or a "FIFTH" function. The use of single letter variables has been explained but the use of "FIFTH" functions has not. They basically allow you to "get back" the information put in using most of the other commands. Many functions have parameters themselves, usually the name of object types. Below is a description of all 13 of them:-

NO

FORMAT: No (Name of object type)

DESCRIPTION: Returns the number of subscriptions of the given object type.

COLUMN

FORMAT: COLUMN (Name of object type)

DESCRIPTION: This function returns the column number (x co-ordinate) of the given object type according to ALL and USE. If the CURRENT of that object type is ALL then error "A Invalid argument"results.

LINE

FORMAT: LINE (Name of object type)

DESCRiPTiON: This returns the line number (y co-ordinate) of the given object type according to ALL and USE. If the CURRENT of that object type is ALL then error "A Invalid argument" results.

SCREEN

FORMAT: SCREEN (Name of object type)

DESCRiPTION: Returns the code of the character that the given object type will be printed as. Use the CHR$ function to get the actual character.

ATTR

FORMAT: ATTR (Name of object type)

DESCRiPTION: Returns the colour that the given object type is printed in. It is given in the same format as you would get from a normal ATTR function - see page 116 of the Sinclair BASIC programming manual.

DIRECTION

FORMAT: DIRECTION (Name of object type)

DESCRIPTION: Returns the direction (0 to 15 inclusive) of the given object type subject to ALL and USE. If the current of the given object type is ALL then error "A Invalid argument" will result.

CURRENT

FORMAT: CURRENT (Name of object type)

DESCRIPTION: Returns the current subscription of the given object type. If the current is ALL then 0 is returned.

```
MASK
FORMAT:       MASK (Name of object type)
DESCRIPTION:  Returns the colour mask for the given object
              type. It is used for PAPER or INK 8 etc. When
              converted to binary any bit that is set signifies
              that the corresponding bit from the actual
              colour (returned by ATTR) is not taken from that
              byte but from what was already on the screen.

VELOCITY
FORMAT:       VELOCITY (Name of object type)
DESCRIPTION:  Returns the delay  in 1/50ths of a second between
              successive moves of the given object type
              according to ALL and USE.

JUMPS
FORMAT:       JUMPS (Name of object type)
DESCRIPTION:  Returns the number of pixel jumps an object will
              make every time it is moved subject to ALL and
              USE. Error A if the current of the supplied
              object type is ALL.

LIMIT
FORMAT:       LIMIT
DESCRIPTION:  Returns the number of the line at which the limit
              service routine is sited. If this is bigger than
              9999 then this signifies that no limit routine
              is to be called.

INTERACT
FORMAT:       INTERACT
DESCRIPTION:  Returns the line number of the interact service
              routine if there is one. If this is bigger than
              9999 then this signifies that no interact service
              routine is to be called.

STATUS
FORMAT:       STATUS (Name of object type)
DESCRIPTION:  Returns a 1 If the object is enabled or 0 if
              disabled. Subject to ALL and USE but if the current
              is ALL then error A will be produced.
```

BREAK KEY DISABLE

"FIFTH" allows you to disable the BREAK key from within a program. "POKE 65239,1" disables it while "POKE 65239,0" enables it again. As well as offering more program security this is a useful solution to the problem that you sometimes press BREAK by mistake during a program.

HINTS AND TIPS

(a) Saving application programs
When you write a program using "FIFTH" you will have to save "FIFTH" as well as the BASIC program in order for it to work on re-loading.
To SAVE "FIFTH":- SAVE "Data" CODE 61030,4506
(This also saves the user-definable graphics)
Remember to VERIFY.
To LOAD "FIFTH":- LOAD "Data" CODE 61030,45O6
(This also restores the UDGs)
Save "FIFTH" immediately after the recording of the BASIC program. Have the BASIC program auto-run and at the entry point incorporate the line:-

```
CLEAR 61029: LOAD "Data" CODE 61030,45O6
```

which will lower RAMTOP and then load "FIFTH".

(b) The use of CONTINUE
You can usually use the CONTINUE command to restart program execution after an error, intentional or not, occurs. When using "FIFTH", however, this is not possible. To avoid this problem it is best to organise your program structure into small subroutines and to make correspondingly great use of the GOSUB statement. You can then test each subroutine individually - minimising the amount of bugs.

(c) The use of RANDOMIZE
In most "FIFTH" programs, line 20 is "RANDOMIZE USR 61030". You may think that the USR function always returns the same number for the seed of RND so that random numbers always start from the same point. This is not so since a "random" number is always returned by the USR function.

(d) The use of "FIFTH" REMs
A REM statement containing "FIFTH" commands must always be the last statement on that particular line. If there are more statements then they are ignored.

INDEX