# THE COMPLETE GUIDE TO SAM basic

The best BASIC guide in the history of the world
(Relion)


parts of the disk guide here in one booklet.

**Original edited on ENCELADUS diskmags by Graham Burtenshaw.**
**Booklet by Wolfgang Haller/SPC/Cologne/April 2005**

## WELCOME TO THE COMPLETE GUIDE TO SAM BASIC

There have been rather a few complaints about the SAM Coupe user manual - many people say that it's not sufficiently comprehensive to allow users to become familiar with the many features of their new computer, and the "other" manual, the Technical Manual, is considered TOO technical for a lot of people wishing to learn SAM BASIC. Heading these pleas for help, I have decided to put together the definitive guide to SAM BASIC, which will be known as the title above, and will attempt to explain both the simpler aspects of BASIC, and the more subtle details of each command. I will be treating the various areas of BASIC in alphabetical form, describing each command/function in turn.

SAM BASIC is heavily based on the Spectrum language Beta Basic, which itself is based on Spectrum Basic. Spectrum Basic has several distinguishing features - the editor, where typing is carried out in a 'command line' rather than all over the place, the structure of strings and arrays and the speed. (slow!) Other than that, it is fairly standard BASIC which would be the same on virtually any other home computer, apart from things like graphics and sound commands. The Spectrum manual, or any other BASICguide is recommended for those who are absolute beginners, but otherwise, read on!

## PROCEDURES

SAM BASIC has the powerful facility to write procedures. These are small program 'modules' - like programs in themselves - and are essential if you are planning to structure programs. Structuring programs can be tedious to actually do, but greatly simplifies subsequent de-bugging and developing.
SIMPLE PROCEDURES: Say you are writing a program, where every so often, you want the Coupe to make a sound effect - such as a rising tone. You could just type in the lines to do this each time, but this would waste time and memory and would make the program more difficult to de-bug. A much easier method is to use procedures.

```
10 DEFPROC noise
20 FOR n=0 to 50
30 BEEP .1,n
40 NEXT n
50 ENDPROC
```

DEFPROC tells the Coupe that the following lines, up to the next ENDPROC, are part of a procedure, and will not be obeyed as usual program lines. The name of the procedure (noise) follows the DEFPROC: this name can be anything you like as long as it is not the name of a SAM command. IE, you can't do DEFPROC BEEP, because BEEP is a SAM command. Lines 20 to 40 make the 'noise'. The ENDPROC tells the Coupe that the procedure ends at line 50. If you type those five lines in and RUN them, nothing will happen. To make the 'noise', you need to "call" the procedure. To do this, just type in the name of the procedure. Adding the line:

```
60 noise
```

and RUNNING the program will result in your Coupe singing to you!

```
10 DEFPROC noise
20 FOR n=0 to 50
30 BEEP .1,n
40 NEXT N
50 ENDPROC
60 FOR n=1 to 5
70 noise
80 NEXT n
```

The program should make five noises. RUN it, and you will find that only one noise is made. Why is this? It's because there are TWO variables called 'n' - one inside the procedure, and one outside the procedure. The two variables do not stay separate, and consequently the procedure ends with n=50, so the NEXT n in line 80 has no effect (because 'n' is over the limit of 5 defined in line 60 ). We can overcome this by either using different names for the variables, or alternatively, using the command LOCAL. If you insert the line

```
15 LOCAL n
```

and RUN the program again, it will work. LOCAL is followed by a list of variables, separated by commas, which are to be "local" to the procedure - in other words, the Coupe remembers the values of these variables in use outside the procedure before setting



up new ones in the procedure itself, and when an ENDPROC is reached, these values are restored. So the line

```
LOCAL a$,x,count,block$
```

would allow you to use those variables within a procedure SEPARATELY to any other variables with the same name.

```
10 DEFPROC drawline
20 PLOT x,y: DRAW 5,0
30 ENDPROC
40 LET x=50, y=50: drawline
```

The small program uses the variables 'x' and 'y' both in and out of the procedure, so they do not need to be "localised". The program will draw a small line starting at co-ordinates x,y.

```
10 DEFPROC noise num
20 FOR n=0 to num
30 BEEP .1,n
40 NEXT n
50 ENDPROC
60 noise 50
```

This program will make one "noise". Line 10 contains a DEFPROC, followed by the name of the procedure ("noise"), and then the name of a variable - num. A noise is then made so that num is the highest note played. To get a value into num, we do not use LET - instead, we put in a number when we call the procedure. So 'noise 50' in line 60 does not mean "call procedure called 'noise 50'" - it means "call procedure called 'noise', use 50 as first value". You could just as well do something like noise 5000 (which will not work because you can't BEEP at 1000!), noise -50.5 and noise note - ie, using a variable rather than an actual number.

```
10 DEFPROC noise num,times,delay
20 FOR m=1 to times
30 FOR n=0 to num
40 BEEP .1,n
50 NEXT n
60 PAUSE delay
70 NEXT m
80 ENDPROC
```

Any number of "parameters" can follow the name of the procedure, as in the example, which uses the variables num, times and delay. Call it using something like noise 50,8,5 which would make 8 noises, 50 being the highest note in each, and PAUSING 5 frames between each noise. The list of parameters must be in the same order as in the DEFPROC statement, and are separated by commas. (apart from the first one).

### *Procedure with Local, Data*

When passing variables as described on the last page, the variables are AUTOMATICALLY LOCAL to the procedure - you don't need to use LOCAL. So, in the program

```
10 DEFPROC drawline x,y
20 PLOT x,y: DRAW 5,0
30 ENDPROC
40 FOR x=0 to 50
50 drawline x,x
60 NEXT x
```

the variables x and y are kept separate in and out of the procedure. The program will draw a series of lines, to make a thick diagonal line.

```
10 DEFPROC noise DATA
20 FOR n=1 to 5
30 READ a
40 FOR m=1 to 2
```

```
50 BEEP 1,a
60 NEXT m
70 NEXT n
80 ENDPROC
90 noise 10,50,3,3,32
```

This example shows how to use the DATA function. In line 90 the procedure is called, followed by a list of 5 data items which are to be used. The DEFPROC in line 10 has DATA after the name of the procedure instead of a list of variables. When the command READ is used, as in line 30, a data item is taken from the line where the procedure was called from. So the variable 'a' becomes 10, is played twice, then becomes 50 and is played twice and so on, for all 5 values. The function ITEM (see later on) can be used as an alternative method to check if there is any more data.

### *Procedure with REF*

```
10 DEFPROC write REF a$
20 FOR n=0 to 20
30 PRINT at n,0;a$
40 NEXT n
50 ENDPROC
60 LET f$="HAMSTER"
70 write f$
```

REF is also used with procedures. When the procedure "write" is called, at line 70, the name of the procedure is followed by the name of a variable, instead of an actual variable. (Which would be, eg, 70 write "Hamster"). The REF in line 10 puts the VALUE of this variable into the variable following the REF - so that a$ becomes whatever f$ is, which is "HAMSTER"! It is then printed down the screen.

```
70 LET a$="HAMSTER"
71 LET b$="ANTEATER"
72 LET c$=a$
73 LET d$="WARTHOG"
74 LET animal$="BLACMONGE"
80 INPUT "Which variable? ";v$
90 write v$
```

NB: You can't use "write" as the name of a procedure because it's also a BASIC command. This program shows an example of how to use REF. After typing lines 10 to 60, add these lines, and RUN: you'll be asked to type in the NAME of a variable - eg, type in "b$". The procedure is then called, the automatically LOCAL variable a$ becomes whatever is in the variable that you specifiedand that is printed down the screen. So entering f$, a$ or c$ would result in the word "HAMSTER" being printed down the screen.

```
10 DEFPROC circledraw x,y,r
20 DEFAULT y=x
30 DEFAULT r=80
40 CIRCLE x,y,r
50 FILL x,y
60 ENDPROC
```

### *Default and recursion*

DEFAULT is another useful command for use with procedures. (And in ordinary BASIC). This creates a variable only if it doesn't already exist. So in the example on the right, typing circledraw 80,50,20 would draw and fill a circle at co-ordinates 80,50 and radius 20. But circledraw 80,50 would do a circle of radius 80, (line 30), and typing circledraw 30 would draw and fill a circle at co-ordinates 30,30, radius 80. Suitable for lazy people.

```
10 DEFPROC diamond x,y,s,diff
20 DEFAULT diff=15
30 PLOT x,y-s: DRAW -s,s
40 DRAW s,s: DRAW s,-s: DRAW -s,-s
```



```
50 IF size<5 THEN GOTO 100
60 diamond x,y+s,s-diff
70 diamond x,y-s,s-diff
80 diamond x-s,y,s-diff
90 diamond x+s,y,s-diff
100 ENDPROC
110 diamond 128,88,40
```

This procedure, (from the BetaBasic manual!) demonstrates RECURSION - that is, the calling of a procedure from within that procedure. A limit is used, to stop the procedure calling itself forever. Note - in line 50, a GOTO 100 is used to end the procedure rather than an ENDPROC.
That's it for procedures! I'm going to dedicate at least one page to EVERY SINGLE command and function! (Or at least try!)

## ABS

is a function, and stands for ABSolute magnitude: typing ABS n where n is a number or variable removes any minus sign that may be present. So typing PRINT ABS -9 is the same as PRINT 9, which is the same as PRINT ABS 9.
PRINT ABS (x-y*x+1) prints the ABS of the argument (x-y*x+1), ie, an argument is what the function acts on and can be just a number, a variable or an expression (a combination of calculations, such as x-y*x+1).
PRINT -ABS x will do the opposite of ABS and always give a negative number. If you want to just swap the sign of a variable, just stick a minus sign in front of it, so -(-8) becomes 8.

```
10 INPUT a,b,c
20 LET x=SQR ABS (b^2-4*a*c)
30 LET x1=(-b+x)/(2*a)
40 LET x2=(-b-x)/(2*a)
50 PRINT x1
60 PRINT "or"
70 PRINT x2
```

APPLICATIONS: There are lots, but not really easy enough to demonstrate here. Basically to keep variables positive whilst keeping their value. The program is an instant quadratic equation solver, and uses ABS to make sure that negative numbers are converted to positive ones before having their square root found. (You can't SQR a negative number).

## ACS

stands for Arc CoSine, and is used to calculate the value of an angle from its cosine. (See SIN and TAN for more detail on this). Note that the Coupe, and most other computers, use radians rather than degrees when dealing with angles. ACS gives the value in radians of the angle that has the specified cosine (in degrees). To convert back to degrees, multiply the result by (180/PI).
Example - PRINT 180/pi * ACS .5 will give 60, the angle which has cosine .5.

*SEE ALSO -> SIN, TAN*

## AND

is an extremely useful function which has several main uses. The most obvious is its use in IF statements. Eg, IF score>50 AND time<100 THEN PRINT "well done" or whatever: the command(s) following the THEN are only carried out if BOTH conditions are true. If not, the next program line is carried out. When the Coupe comes across a line such as this, it checks each condition even if the first one is false - this means that you can't use AND to see if a variable exists or not etc.
If you are testing several conditions at once, you may need to use brackets. Brackets section off a particular piece of an expression, and keep it separate to the rest of the expression. The two lines below are different, because of the use of brackets in line 20.

```
10 IF x=5 OR x=6 AND score>50 THEN ...
20 IF x=5 OR (x=6 AND score)>50 THEN ...
```

It is possible to rewrite any IF statement in a different way, which is quicker and takes up less memory, using AND.
Lines 10 to 20 in the program below will print "correct" if x=0, and "Not correct" if x=1.

```
10 IF x=1 THEN PRINT "Not ";
20 PRINT "correct"
30 PRINT "Not " AND x;"correct"
```

Line 30 does exactly the same thing. If you PRINT something like 5=5, you will get the result 1, and if you PRINT 5=3, or 5<>5, you will get 0.
The Coupe returns a 1 if a condition is true, and a 0 if it is false. So 5<>3 would give 0, "HAMSTER"="VERMIN" would also give 0 as would "AMIGA"="BRILLIANT" and "ENCELADUS"<>"BRILLIANT". Think about it!
You can't mix a string and an expresssion like this; ie, you can't do PRINT x=a$. Going back to the example program: if you type PRINT 5 AND 3 you will get the result of 5: this is because the Coupe regards any number other than 0 as being "true" in this type of situation, and if it is true, will carry out the command preceding the AND. So PRINT 5 AND 0 would give 0, (if it is false, the command is still carried out, but values are replaced by 0). If you typed PRINT "eduard shevardnadze" AND "wyautis llandisbergis"="Mrs wyautis llandisbergis" nothing would be printed, because there is a condition following the AND, and it is false. (Sorry about the spelling but in the Middle Ages people spelt things how they felt like it, so I thought I'd the same). But it you removed the "Mrs " bit, the ex-Soviet foreign minister would be printed. Now you see how the example works!

```
10 LET x=x+(IN 57342=93 AND x<255)*s-(IN 57342=91 AND x>0)*s
20 LET y=y+(IN 64510=94 AND y<175)*s-(IN 65022=93 AND y>0)*s
```

The example above is an example of how to move something about the screen. x and y are the co-ordinates, and s is the speed. In each bracket, there are two conditions to be tested. If they are both true, the contents of the bracket becomes a "1". (In effect). If either or both of the conditions are false, the contents of the bracket becomes a "0". This 1 or 0 is multiplied by s, but as anything multiplied by 0 is 0, a 0 will stay the same! These values are then added to x and y, so x increases if IN 57342=93 AND x is less than 255, and it decreases if IN 57342=91 AND x is greater than 0. (The IN function reads a byte from an I/O port. In this case, the numbers specified correspond to keyboard ports. (for keys I,O,Q,S). See IN for more detail).
NOTE: This usage of AND is slightly different from standard BASIC. If you type in PRINT 10 AND 5 on the Coupe, or the Spectrum, you will get 10 (because 5, on its own is "true", so 10 is printed). If you try it on a BBC or something, you will get 0; this is because the numbers are combined in a binary sense. You can still do this of course on the Coupe, just use the function BAND. (Binary-AND). What would this give? PRINT "ENCELADUS is ";"BRILLIANT" AND ((reader$="intelligent" OR reader$="rich") AND reader$<>"Reginald Turnill");"USELESS" AND (SAM$="kaput" OR NOT reader$="normal").

## ASN

stands for ArcSine and gives the angle (in radians) that has a given Sine.

*SEE ALSO -> ACS, SIN, TAN.*

## AT

is essential if you are planning to position text (and graphics) at various positions on the screen. AT can only follow PRINT or INPUT (I think!), and controls whereabout on the screen the printing will take place. AT is followed by two numbers.The first represents the row where the text will be printed, and usually ranges from 0 to 21. The second number represents the column, and can range from 0 to 31 in MODES 1, 2 and 4, and from 0 to 63 or 0 to 85 in MODE 3, depending upon which CSIZE is selected. System Variable 54 controls the character height, and can be used to make smaller characters, and give more rows - as in the SMALLTEXT program on ENCELADUS ISSUE 2.

AT is always followed by a semi-colon (;) or a comma (,), as the example demonstrates.

```
10 PRINT AT 6,0;"HAMSTER"
20 FOR n=0 to 20
30 PRINT AT n,n;"*"
40 NEXT n
```

If you have defined a WINDOW, the range of AT is confined to the limits of the WINDOW: if you try to PRINT at an invalid position, you will get the error report "Off screen".
Place a semi-colon after the text to be printed if you want the print position to stay the same, eg, adding a ; to the end of line 30 would make the program print one row of asterisks.



Normally, PRINTING takes place on the Upper Screen - you can't PRINT AT rows 22-24, where editting takes place. Normal printing occurs on stream #2, which is usually open to channel "S" (screen). So doing PRINT #2;"blah" has no effect because it is no different from normal. Streams #0 and #1 are open to channel "K" (keyboard), which is the lower screen - so is you type PRINT #0; AT 1,0;"HAMSTERS ARE FUN", the text will appear right at the bottom of the screen. If you print more text like this, any existing text will scroll up. If you type this as a direct command (Ie, not in a program), the text will be obliterated by the "0 OK 0,1" report, so you can prevent this by sticking a PAUSE or something after the PRINT statement.
INPUTTING normally takes place on stream #0, but by typing something like INPUT #2;AT 10,10;a$ you can INPUT on the upper screen, just as if you were printing. Remember that the inputted text is not cleared after RETURN is pressed, though. Stream #16 is open to channel "$", which is what RECORD and BLITZ use. See LIST, PRINT and DIR for some interesting uses of this! The # sign followed by a stream number can be used in any command which involves printing to the screen - so you can LIST and DIR to the lower screen, although I can't think of any use for this! If for some reason you want to, you can replace AT with CHR$ 22, which is the control code for AT. The row and column are also coded as CHR$'s, so the two lines are identical.

```
10 PRINT AT 10,10;"NEIL KINNOCK"
20 PRINT CHR$ 22;CHR$ 10;CHR$ 10;"WELSH GIT"
```

## ATN

stands for ArcTangent, and gives the angle (in radians) that has a given tangent.

*SEE ALSO -> ACS, SIN, TAN.*

## AUTO

is not a command that I use very often (in fact I have never used it), but I suspect that some people will find it useful. AUTO can not be used in a progam (well, it can, but it would be absolutely pointless) - it is only used when writing programs. Typing AUTO will list a screen full of any existing program, and then present you with a line number at the bottom of the screen: this number will be the current line number plus 10. You can then type in a program line and press RETURN, to be presented with another program line. This will continue indefinately, so to escape from the process, delete the line number and type STOP or something, or press the BREAK button. AUTO usually increments line numbers in steps of 10, but by typing AUTO 500,5 or something, you can alter this. The first number after the AUTO is the line number to start at, and the second one is the step that you want to use. The manual claims that line numbers can only go up to 61439, but in fact the limit is 65279. (which is 1 less than 255*256).
Pressing RETURN instead of typing a line when AUTOing will delete any existing line with the same number and skip to the next one. (if there is one). If you are given a number which is the same as an existing one, which you want to keep, press EDIT to bring the line down, and RETURN as normal.
This function is for some reason not mentioned in the manual - presumably because it only operates in MODES 1 and 2, and they predict you will not be using these modes much!

## ATTR

is used to find the attributes, or colour status of a "cell" in modes 1 and 2. Each pixel has its own colour in modes 3 and 4, so you can use POINT to do the same thing: in modes 1 and 2 however, POINT can only give 0 or 1. ATTR gives the value of the byte in memory that controls the colour of the colour cell at r,c - rows, columns, as with AT. For instance, the program below will find the attributes of the cell at (5,9), which will be 16.

```
10 MODE 1: PAPER 2: INK 0: CLS
20 PRINT ATTR (5,9
```

This value is calculated as follows: each bit of the number has a particular purpose as listed below. So for the value of 16 results from a red (colour 2) paper and a black (colour 0) ink.

```
BIT 0 - bit0 of ink
BIT 1 - bit1 of ink
BIT 2 - bit2 of ink
BIT 3 - bit0 of paper
BIT 4 - bit1 of paper
```

```
BIT 5 - bit2 of paper
BIT 6 - 1 for BRIGHT else 0
BIT 7 - 1 for FLASH else 0
```

BIN$ 16 gives 00010000, which can be read off using the list. In MODE 2, there are 6144 colour cells, as opposed to 768 in MODE 1, but you cannot read all of them: you can only read the first of each block of 8. You can get around this by just using PEEK. The attributes in MODE 1 start at (screen address)+6144, and in MODE 2 at (screen address)+8192.

## BEEP

This is a relic of Spectrum Basic and will probably not be used much, especially seeing that we have POW, ZAP, BOOM and ZOOM to make spot effects. It can, however, be useful for doing tunes etc, although the music program on ENCELADUS 3 is better.
BEEP is followed by two numbers: the length of the BEEP, in seconds, and the pitch of the note, in semitones. (pitch 0 is middle C). The pitch can range from -60, which gives a rather rude noise, and up to 71 which is virtually inaudible. The duration can be as long as 15 seconds, and can be as short as you like - although very short durations such as 0.0005 seconds are only effective with very high notes. BEEP is not very versatile, and much better effects can be gained using machine code, or by using the Phillips sound chip.
Alternatively, you can use the OUT command. Port 254 partially controls the microphone, and by repeatedly writing different data bytes to it, you can make buzzing noises.

```
10 FOR n=0 TO 60
20 BEEP .005,n
30 NEXT n
40 FOR n=60 to -60 STEP -1
50 BEEP 1/(n+61),n
60 NEXT n
```

The program has two sections which make odd noises, BEEP is most effective when used in a loop and with short durations. The expression in line 50 makes the duration shorter when n is higher.

```
C   D   E   F   G   A   Bb  B   C   Left-note
0   2   4   5   7   9   10  11  12  converter.
```

## BIN and BIN$

These are very useful pseudo-functions, especially if you like either writing machine code, or inspecting the ROM, SVARS, I/O ports and so on. In case you didn't already know, the Coupe, and all other computers, use base 2 for everything - just as we use base 10. This is better known as binary, and is a system of counting using only two digits. (Which can easily by represented in a computer). Numbers up to 255 can be written in 8 digits in binary. Some numbers are listed below with their binary equivalents. From right to left, the binary digits represent 1,2,4,8,16,32,64 and 128 in decimal.

```
  1   00000001
  2   00000010
  3   00000011
  4   00000100
  5   00000101
  6   00000110
  7   00000111
  8   00001000
 10   00001010
 30   00011110
222   11011110
255   11111111
```

Numbers over 255 and up to 65535 require 16 digits. All 8 or 16 digits are always written, whether they are set (1) or not. BIN$ converts a decimal number into binary. So doing PRINT BIN$ 222 will give "11011110" (as a string, not a number). SVARS 3 and 4 control what characters are to be used to print binary digits 1 and 0 usually they are 1 and 0. By changing them to, eg, "*" and



" ", you can print out graphics etc from character sets: this technique is fast and efficient and is used in the DESIGNER program on ENCELADUS 1.
BIN does the opposite and converts an 8 or 16 digit binary number into its decimal equivalent. So PRINT BIN 11011110 gives 30. If you want to use BIN with a variable, you must use KEYIN. Eg, if a$="11100010", then do KEYIN "PRINT BIN "+a$ or whatever.

## BLITZ

is a command that Bruce Gordon and Andy Wright seem very proud of. It's used to replay graphics COMMANDS that have been stored using RECORD TO. The actual graphics are not stored; the commands such as PLOT or CIRCLE are stored; this takes up surprisingly little memory.
To record graphics, first type RECORD TO a$ or whatever your variable is called. Then every time you PLOT, DRAW (TO), CIRCLE, CLS, PAUSE or change colour, this will be stored in the string. RECORD STOP, CLEAR or NEW will stop this process. The graphics instructions are coded as follows:

```
DRAW        0/255 for +/-, then x, and the same for y.
PLOT        followed by the x and y co-ordinates
DRAW TO     followed by the x and y co-ordinates
CIRCLE      followed by x,y and the radius
OVER        followed by 0-3
PEN         followed by 0-1
CLS         followed by 0-1 (whole screen/window)
PAUSE       followed by 0-255
```

Using these codes, it is possible to construct a BLITZ string without actually drawing, by doing, eg LET A$=A$+CHR$ 1+CHR$ RND(255)+CHR$ RND(175) or something.
The manual claims that RECORD and BLITZ work in any MODE, but in fact it does not seem to work in MODE 3. You can RECORD in one mode, and then BLITZ in another mode, to create various effects.
If you wish to animate, say, a series of line drawings, BLITZ can be much more economical than storing each screen. This is how the TRANSMOGRIFIER on ENCELADUS 1 works, which creates some, if I may say so, quite excellent results.
XRG, YRG, XOS and YOS can be used to alter the size of the RECORDED graphic. Zenith Graphics used this in a demo on EDITION 2, and so does the Transmogrifier Demo. (on ENCELADUS 1 if you haven't found it yet). These are special variables which cannot be CLEARED. XOS and YOS control the origin of the x and y axes, and are normally 0 and 0 (bottom-left hand corner). LET xos=128 and LET yos=64 would make PLOT 0,0 plot a point in the middle of the screen (in mode 4,2,1), and you would be able to PLOT (or CIRCLE,DRAW TO etc) negative values. SEE ALSO XOS etc. XRG and YRG control the scale - usually XRG is 256, or in mode3, 512. The Coupe can get confused if you swap modes when altering these variables, so you should always check them. When XRG is 256 it means that there are 256 points which can be plotted to - 0 to 255. If you alter it to 50, 50 will be at the right hand edge of the screen. There will still be 255 pixels along, but you will have to do eg, PLOT 25.8,0 to access them. The same applies with YRG. By altering them, and then BLITZING, you can alter the size of graphics, as in this the example.

```
10 RECORD TO a$: DRAW 255,0: DRAW 0,175: DRAW TO 0,0
20 RECORD STOP: FOR n=256 TO 1968715 STEP 100
30 BLITZ a$: NEXT n
```

## CHARACTER SET

BEEP PAPER BLOCKS is not treated in sufficient detail in the user guide, and so seems a little confusing.
The CHARACTER SET is the set of letters, numbers and symbols that appear on the keyboard, along with control codes, user-defined graphics and to some extent, keywords. The character set can be broadly divided into 4 parts.
1) The control codes. The table at the back of the user guide lists these codes, which have a direct effect when printed. Eg, CHR$ 8 is the code for "backspace", and when PRINTED, as a normal character, causes the print position to backspace. Some of these codes have different functions depending on whether they are printed or produced when typing, such as the keyboard toggle function. Codes 16 to 22 control aspects of printing such as paper and pen colours. When you PRINT these codes, the ROM expects the next code to specify the pen, paper or whatever. So the two lines below do the same thing.

```
10 PRINT PEN 4;AT 5,16;"russian dwarf hamster"
20 PRINT CHR$ 16; CHR$ 4; CHR$ 22; CHR$ 5; CHR$ 16;"r.d.h"
```

Following one of these control codes with an invalid value will give an error report as would be expected.
CHR$ 14 has another special purpose, and that is to show that a floating point number follows in a BASIC program. If you use PEEK to examine a BASIC program (starts at 23755), each number will appear as 14, followed by 5 bytes. 5 bytes can store very big and very small numbers, but I won't go into that here.

2) The second main area of the character set is the numbers, symbols and letters such as "oqh5t3@%U KA}3" etc. These range from CHR$ 32 to CHR$ 127 and are arranged in a standard order known as ASCII, which means that text files from one computer can be read on another computer. The only differences are things like "`" signs, which is why some printers won't print such symbols.
The patterns for each character occupy 8 bytes, allowing eighteen trillion four hundred and forty six thousand billion different characters to be invented. This data is stored in a compressed form in the ROM, and is moved to address &5190 (20880) when the Coupe is switched on. When you load new character sets, you are just overwriting this data, although SVAR 566 holds the address of the pattern data. (minus 256). This must be below 32768, however, or else paging will mean that you will more than likely end up with blank characters. The early versions of the ROM re-loaded the character set pattern data every time you did NEW, but the new version does not - you must reset the Coupe to regain the standard font.

3) Next, from CHR$ 128 to CHR$ 168 come a set of characters which are intended for your own use. Normally, printing CHR$ 128 to CHR$ 143 gives "block graphics", which are produced as you print them, rather than being stored as pattern data. If you type BLOCKS 0, these data for these characters comes from the same area of memory as CHR$ 32-127. CHR$ 128 and 129 give the cursor, and CHR$ 130 gives the „" sign. You can then use UDG (described later) to redefine these characters as with normal characters. When BLOCKS 1 is next selected, the data is not lost. The user guide mentions a command "BGRAPHICS"; this is stuff and nonsence.
CHR$ 144 to 168 cannot be switched in and out with BLOCKS, but SVAR 635 holds the address of the start of the pattern data for these characters. Normally, it follows on from CHR$ 32 - 143, but you can POKE it to whatever you like.

4) Finally is the section from CHR$ 169 to CHR$ 255 which can also be used as user defined graphics, or a second alphabet. SVAR 637 holds the address of the start of the pattern data for these characters, and normally this is 0 - so when you PRINT CHR$ 169, you get a blob representing the first 8 bytes of the ROM. If you POKESVAR 637 with 20880, CHR$ 169 to 255 will give letters and numbers, because the data will come from the normal character pattern data.
CHR$ 192 to CHR$ 255 can be given "expanded definitions" so that when you press a key that is assigned these values, whole strings can be made to appear. This is separate to the pattern data, which determines what the characters look like when PRINTED, and will be discussed later. IE, if you define CHR$ 195 to print "my hamster has no bananas", then when you PRINT CHR$ 195, you will just get a blob, even if pressing F3 gives the message. You can't PRINT messages in this way.

## *UDG*

UDG is a function which returns the address in memory of a given character. So PRINT UDG " " or PRINT UDG CHR$ 32 will normally give 20880, because the pattern data for CHR$ 32 is stored at 20880. The function takes into account the system variables which govern where the various tables of pattern data are held, so UDG gives an absolute value that you can POKE to alter the character. EG - to make the "H" letter into a hamster picture. First find the address of the pattern data for "H". UDG "H" gives 21200: this means that the 8 bytes including and following address 21200 are the pattern data for the letter "H". Each byte consists of 8 bits which can either be 1 or 0 - paper or pen. A thing like a hamster (but more like a rabbit) is shown in the grid on the left, with each line of 1s and 0s representing the 8 bits of a byte. These numbers must then be POKED into the relevent addresses. (shown alongside them). Since the numbers are in binary (because that makes it easier to design), the function BIN must be used. You could do: POKE 21200,BIN 00110000: POKE 21202, BIN 00110000 etc but it's much quicker to do POKE 21200,BIN 00110000,BIN00110000 etc ie, the address followed by the 8 pieces of data. Alternatively you could go and use a character/UDG designer which is a hell of a lot easier and lazier.

```
00110000  21200
00110000  21201
01111000  21202
11111100  21203
00111110  21204
01111110  21205
00111111  21206
00000000  21207
```



## BOOT

is not exactly a complex command. All it does is to load in the DOS from the disk currently in the drive; the DOS is taken to be the first file on the disk. If the DOS has not been saved, then things will go wrong! Normally key F9 is used for BOOT, but typing the command in is exactly the same. BOOT also loads the first file on the disk that has "AUTO" as the first part of its name, so that if you BOOT a disk with no such files, you get the error "no auto* file". Typing BOOT1 (or BOOT213 if you're feeling rebellious) loads only the DOS and not the AUTO file.
Sometimes disks have the first filename (which will be the DOS) made invisible, or made up of invalid control codes, so that when you do a DIR1, the directory is not displayed. You can get around this by doing DIR1 "auto*" which will print up the name of the file that autoloads. Then you can MERGE this file to prevent any protection system setting up, and fiddle about with the programs. Alternatively you can use the Disk Interrogator on ENCELADUS 4.
SVAR 450 holds the page number that the DOS is located at. So to find the beginning of the DOS, do PRINT (PEEK SVAR 450)+1)*16384, and add 9 to get to the actual start of the code. The DOS only takes up around 8000 bytes, not 10000 as is popularly believed, so you can save space on disks by SAVE "SAMDOS" CODE 491529,8192. The page allocation table at &5100 to &5120 shows memory pages used by the DOS as &60: you should always BOOT before opening extra screens, or else the DOS will be loaded into a problemous address.

## BORDER

The BORDER can display any one of the 16 selected colours at any one time, and is completely independent from the current mode. Typing something like BORDER 6 will cause SVAR 587 to be updated, and data sent to I/O port 254, which is described in the table below. The table shows that 4 bits are used to access the CLUT (colour-look-up-table). The CLUT is a table of 16 write-only ports, 1 for each colour. By writing to one of them (using OUT), the colour (1 to 127) for that "paint pot" is selected. So doing OUT 504,68 would make ink 1 appear as green because port 504 controls the colour for colour 1, and the code for green is 68. You may think that because an I/O port can have between 0 and 255 written to it, you can access 256 colours! Well you can't, numbers over 127 have 127 subtracted from them. Anyway, the 4 bits in the BORDER port specify CLUT port is to be used for the border colour. So doing OUT 254,7 is (almost) the same as doing BORDER 7.

```
BORDER   port (254 dec)
bit0     BCD 1 of colour
bit1     BCD 2 of colour
bit2     BCD 4 of colour
bit3     MIC data
bit4     BEEP
bit5     BCD 8 of colour
bit6     THROM
bit7     SOFF
```

The border is more versatile than that, because by altering it quickly enough, various effects can be created. By having a dark or light border, the paper colour alters dramatically due to the contrast. Palette LINES can of course be used in the border but beware that if you have too many, jags will appear in the border.

## BRIGHT

BRIGHT is another command left over from Spectrum BASIC, and isn't really needed. In any MODE, typing BRIGHT 1 will cause the PAPER and PEN colours to be "bright" colours; what this means is that they will be above colour 8, because with a normal palette, colours 8-15 are bright versions of colours 0-7. So if you had previously done PAPER 3 and then you did BRIGHT 1, the paper would change to (8+3)=11 which is normally pink. Obviously if you have different palettes, BRIGHT could well mean "DIM" or "COMPLETELY DIFFERENT COLOUR", but the same principal applies. BRIGHT 0 makes the paper/pen colours "dimmer" by subtracting 8.
The BRIGHT which you might see in tables showing how colours are produced by the video circuitary is completely different. Out of the 128 colours available, there are 16 blues, 16 reds, 16 magentas etc, and 16 dim colours and 16 bright colours. Out of each group of 16, there are 8 dim colours and 8 bright colours. What I should be using are terms like "hues, pastals and shades" but I'm not. Unfortunately there is no quick way to calculate colours so that you can create a gradiented effect: you must pick the colours yourself by trial and error. There may or may not be a program on ENCELADUS 4 to do this for you. There is, however, a way to

calculate the luminosity of each colour and then arrange them such that when you turn the colour of your monitor/TV down a very smooth transition from dark to light grey is produced. A program describing this was on a cover tape of Sinclair User ages ago, there might be an example screen on ENCELADUS 4.

I've missed out **BUTTON!** Well I haven't got a mouse yet.

## CALL

is the most convenient way of starting a machine code routine, although there are several others. The Spectrum lacked such a command, and used USR, which was a function. Hence the program line RANDOMIZE USR 45000 which uses RANDOMIZE as a command which has no obvious effect.
CALL is used in the following way: typing CALL 45000 will exit BASIC and call the machine code routine at address 45000. If there is no routine there, the Coupe is likely to reset. Therefore the command should be used with care. When the routine has been completed, with a RET, the ROM is re-entered and the BASIC program continues. The address following CALL can be above 65535, in which case the relevant memory page is paged in. If you wish to CALL machine code in ROM1 (the one with all the graphic routines), you must either use the jump table in ROM0 or else page it in yourself. If you follow the CALL address with a comma and a list of numberic or string parameters, these items will be placed on the floating point calculator stack; the machine code can then examine these items for use in the routine. I must admit to never using this facility, so I can't comment on it much. (Not at all, in fact!). You can also do something like CALL LENGTH (0,code$) which will call the machine code contained within a string, this is convenient with relocatable code used with BASIC programs.

### *USR*

USR in effect does the same as CALL except that it is a function rather than a command. It means "call machine code at address NN and put the result into XX" where NN is the address and XX can be anything, depending on how you use USR. The result is the value of the BC register pair. So doing PRINT USR 45000 will print on the screen whatever BC is when the machine code at 45000 RETs, or you can be horrible and do things like CALL USR 45000 or CIRCLE 50,50,USR 32768.

### *USR$*

USR$ is similar to USR but is a string function; its result is the string starting in memory page A, at address DE, and length BC. Its use is not recommended if you are not looking for a string, because if the machine code does not load these registers correctly, then an infinitely long string can be printed or something equally disastrous.
It is possible to enter machine code by setting vectors such that your code is entered when, eg, an interrupt is generated, a graphic routine is called, when a certain command is encountered etc. You can crudely alter the break button's function by poking address &5AE0 with the two-byte address of an MC routine which will be called when the break button is pressed, although it'll need to do things to registers when it finishes etc. Address &5AE2 holds the address which is called afterevery frame, and you can also experiment with this.

## CIRCLE

One of my all-time favourite commands, circle is infinitely faster on the SAM than on the Spectrum, although it used to be quite fun watching the Speccy tracing out each part of the circle. SAM circles are also more accurate, which means that when you place them on top of one another, various patterns can be created. Something which you can't do is to draw ellipses. It would be nice to be able to do this by altering XRG and YRG and drawing a circle, but unfortunately this won't work. You have to write a procedure which uses SIN and COS to draw ellipses.
Conversely, you can't have circles in MODE 3 - you can only have ellipses. Using FATPIX 1 solves this, but then you're left with horrible blocky things. The only way to draw a decent circle in MODE 3 is to do it yourself using SIN and COS again.
CIRCLE x,y,r draws a circle at co-ordinates x,y, and of radius r. When r is over a certain value, the circle may wrap around onto the other side of the screen: this occurs in mode 3 as if it were in mode 4; ie, circles in mode 3 can only go up to a certain size. There are loads of things you can do with circles - draw circles in machine code by calling &142 with C and B the co-ordinates and A the radius - although I've experienced some problems with this...

See a separate entry for **OPEN #** and **CLOSE #.**



## CLEAR and OPEN

are used to fiddle about with memory. When you're writing fairly short BASIC programs, you will probably never have to use them, but if your programs start getting longer, or using machine code, you'll need to use these two commands.
OPEN and its twin (eh?) OPEN TO allocate memory pages to BASIC. Each "page" of memory contains 16K, and there are 15 on a 256K SAM and 30 on a 512K SAM.
Machines with 1Meg upgrades fitted have more. Because the Z80B can only address 64K of memory at one time, the 256 or 512K of the SAM Coupe are split up into these pages. Although in BASIC you can do something like POKE 507904,255, address 507904 doesn't actually exist; rather it is address 32768, but page 30. Only addresses 0 to 65535 can be read from or written to in machine code. Addresses 0 to 32767 contain two pages of memory, and addresses 32768 to 65535 contain another two.
Two I/O ports control whichpages are "paged" into "low" and "high" memory. Low memory normally contains ROM0 and system variables etc, wheres high memory normally contains the screen, when it is being dealt with, BASIC program or other code. ROM 1 also resides there, but does not get in the way of either BASIC or MC. (not usually). If you have a massive BASIC program, the program will be spread over several pages, and only two pages of it will be paged in at any one time. As you list the progam, other pages are paged in as necessary. That caused some of the problems in early ROMS. But you don't have to worry about any of that. All that OPEN TO does is to allocate memory pages to BASIC, by altering a page allocation table, so that the DOS and SCREENS can't use the same memory - if they did, things would go terribly wrong!
When you DIM a large array, you may be greeted with an error message - "out of memory". Provided that the array is not ridiculously large, you can OPEN some more pages. On a 256K machine you can normally OPEN up to 10 pages for BASIC use; on a 512K machine twice this amount. OPEN x opens up x MORE pages, but OPEN TO x opens or closes pages as necessary so that x pages are opened to BASIC. There is no CLOSE command. So, after doing OPEN TO 10, which is usually sufficient, you must then do CLEAR. This alters RAMTOP, which tells the Coupe where memory "ends" - though not in a physical sense. With 10 pages open, this lets you CLEAR (10*16384) = 163840. Then you can write a huge great massive whapping program which occupies many memory pages.
Doing something like CLEAR 30000 prevents your BASIC program from taking up memory so that it goes over 30000, where you might have a machine code program which could otherwise be obliterated by the BASIC. Altogether CLEAR and OPEN are very useful, but if you don't know what on earth I've been going on about, don't worry.
Sorry, there is a CLOSE command. I forgot. No, what I meant was: there is no CLOSE TO command. CLOSE just does what it sounds like, and closes memory pages.

*See -> OPEN# for CLOSE#.*

## CLS

stands for Clear Screen, and surprisingly enough, speeds up the computer. Ha! Ha! That had you stumped. CLS actually clears the entire screen, including the lower window, with the current paper and pen values. Address &5A48 (or SVAR 72) holds the current paper colour as a byte. Since each pixel in mode 3 or 4 takes up half a byte, the two halves of this byte should match - if they don't, you get magic stripey paper! SVAR 73 holds the pen colour, in the same way, do if you do something like POKE SVAR 73,185, you can then DRAW, PRINT and PLOT in striped ink.
CLS 1 only clears the current window, so that if you do it without previously defining any windows, only the upper screen is cleared. CLS# (or key F6) not only clears the screen, but also resets the colours and border etc, and is one way of getting back to normal after messing up the palette. If you happen to dislike white ink on black paper, you can easily define a key to change the attributes to something else - see DEF KEYCODE for this.
CALL &14E will also clear the screen, and CALL &151 will clear the lower screen. (So will INPUT "").

## CODE

is a function which gives the ASCII code of a given character. Eg, PRINT CODE "A" prints 65. PRINT CHR$ 65 prints "A". You can also do, eg PRINT CODE CHR$ 65 which is totally and utterly pointless, but then if you're stuck indoors on a rainy day...

*See also -> BLOCKS.*

## CONTINUE

This command has no conceivable function within a program apart from in conjunction with ON ERROR. (*See ON ERROR for an explanation*). Its main use is with dealing with errors which occur during run time of a program. For instance, if you run a program

which tries to DRAW TO 1874,1293438 the program will stop with an error such as "Integer out of range". But - typing CONTINUE will continue! Unless the error - causing command is a particular type of command such as a peripheral operation (LOAD) etc, or if ESCAPE was pressed, or if OK resulted, then CONTINUE will either do nothing, or skip back to the command after the last error - if BREAK or OK resulted.
CONTINUE may not work in a few cases where you would expect it to, but don't go trying to work out why not.
When an ON ERROR statement causes a subroutine to be called when an error is generated, CONTINUE will retry the command, whereas RETURN will carry on with the next command - so there is a definate difference, even if it does not seem so when the program actually runs.

*See also -> POP*

See -> **SIN** for **COS.**

## CSIZE

is a direct remnant of BetaBasic, but has been severely reduced in both flexibility and versatility. BetaBasic allowed characters to be displayed in sizes from single pixels up to whole screen size, and CSIZE could also be used with GRABbed strings to string or enlarge them. While this second feature is rather extravagent, the former was extremely useful. Now, on SAM, CSIZE lets you do two things - make the letters big or small, and make them a bit spaced out. Hardly versatile. I started writing a printing routine that allowed different sizes to be specified a while ago and may well put such a routine on a future issue of ENCELADUS.
The lack of CSIZE options is perhaps due to the demands of the different modes available on the SAM - it's easier to print characters in modes 1 and 2, because each byte is set out in the same way as the pattern data is stored. CSIZE 8,9 is the default selection when SAM is switched on: one extra line per row is allowed to avoid ascenders and descenders meeting; CSIZE 8,8 is generally more useful because it gives more rows on screen and spaces the rows more pleasingly. New character sets can easily be designed to allow different CSIZEs to operate efficiently - see the SMALLTEXT program on ENCELADUS 2.
CSIZE x,y lets you specify the x and y dimensions of each character. In the horizontal direction, characters can be either 6 or 8 pixels wide: in the vertical direction, characters can be either 8 or 16 pixels high, and can be spaced out in various ways. Eg, CSIZE 8,15 gives 8x8 characters spaced at 15 pixel interval rows.
SVAR 54 and SVAR 55 hold the height and widths respectively of the characters, and since you can poke these addresses with any values bewteen 0 and 255, it may well be more useful to poke them directly rather than use the CSIZE command. When a new CSIZE is selected, however, the size of the window is automatically adjusted.
This means that if you do POKE SVAR 54,16 to give double-height characters, the lower window will appear in the middle of the screen, because it is still at row 22 - but as there are only 11 rows, this wraps round to the top of the screen again.
The SVARS between 56 and 63 need to be altered to adjust the size of the windows if this technique is used.
If you want to display very large letters, then it is best to design a set of large letters using some kind of art package, GRAB each letter into a string with a meaningful name, and then write your own routine to display and possibly animate each letter. This is more useful than using CSIZE 8,16 because although CSIZE can make characters larger, they become more blocky. Large characters on teletext, on the other hand are not blocky because they have the corners smoothed out. This is a feature that CSIZE lacks and that you must implement yourself.

## DATA (Restore, Read and Item)

These commands are used to read data from within a program: the data must be in an actual program. The program below shows how to use them, and is a program to change the palette colours. The RESTORE statement in line 10 is followed by a line number, and tells the Coupe where the first data line is located. If you only have one lot of data, this is not really necessary, because if you RESTORE to a line which is lower than the first actual data line, this makes no difference. However, if you have lots of data, RESTORE should always be used. There then follows a loop which READs a number from the data list into the variable "a", this value is then used to alter the palette.
The data is held in lines 60 and 70 and consists of any number of numbers separated by commas. If you ran this program, it would in fact stop with the report "OUT OF DATA". This is due to the fact that there are only 15 data items, whereas the loop in line 20 instructs 16 data items to be read. The way to cure this is to just stick another data item on to the end of line 70 (or line 80 - it makes no difference).

```
10 RESTORE 60
20 FOR n=0 TO 16
30 READ a
40 PALETTE n,a
50 NEXT n
60 DATA 0,1,32,48,64,88,98,120
70 DATA 111,80,20,35,36,71,42
```

READ can be followed by a list of variables that are to be read to. So if, for example, you did READ a,x,y,b(n),a$ or something similar, five data items would be read in one go, and the next data item would come from the sixth item along.

```
10 DATA "hamster","gnu","woodlouse",a$
20 READ A$
30 PRINT A$
40 GOTO 20
```

This program would read in strings from line 10. It would print the names of three things, and then print "woodlouse" again, because the "a$" as a data item means: the contents of a$.
If the string quotes were removed from the data items, READ A$ in line 20 would give an error, because the Coupe would read them as the names of variables. READ A would give "variable not found". But, READ LINE A$ would work just as if the string quotes were present. It is a way of avoiding typing string quotes all the time.
ITEM is a function which shows what the next data item is - ie, whether it is a string (1), a number (2) or nothing (0). PRINT ITEM will give one of these three values, and is a convenient way of checking when all the data has been read.
The address in memory of the current data item is given by the three bytes at address &5A8A. The first byte gives the page, and the second and third bytes are the high and low bytes which give an offset plus &8000.
READ and DATA can also be used in a slightly different context in procedures - please see ENCELADUS 3 for the CGTSB part 1.
Please see ENCELADUS ISSUE 3 for explanation of DEFAULT.

## DEF FN

is a rather powerful command, although it is not used a great deal. It in effect lets you define your own function, in much the same way that procedures let you define your own commands. However, the definition of a function can occupy only one program line. Although there is a way to get around this, described later.
A DEF FN must take place within a program, although

```
10 DEF FN a(x,y) = SQR (x+y)
```

it can be used from outside the program. If the line to the left were typed in and RUN, nothing would happen because it is a DEFINITION and not a command. To use the new function "a", do something like PRINT FN a(2,3). Note that the name of the function is not limited to one letter names, as on the Spectrum.
The DEFFN is followed by the name of the function, and then, if any parameters are needed, these parameters in brackets. Just as in procedures, these variables are LOCAL to the function. Then there is an = sign, and then the actual definition of the function - which in this case is simply to find the square of two numbers added together. If no parameters are needed - ie, no brackets - then the function acts as if it were a variable defined through LET, and is rather pointless.
You can also do string functions, such as DEFFN double$(a$)= a$+a$ which will double a string when called using, eg, PRINT FN double$("hello").
The way of getting a fuction to be made up of several lines is simply to make use of several functions, as demonstrated in this program. FN total gives a number which is made up of the total of several other functions. In case you're wondering, this function is completely useless.

```
10 DEF FN total(x,y)= FN a(x)+FN a(y)+FN b(x,y)+FN
   a$(chr$ x)
20 DEF FN a(n)= SIN (n+n)
30 DEF FN b(x1,y1)= TAN (x1+y1) MOD 360
40 DEF FN a$(x$)=x$+CHR$ (30+CODE x$)
```

```
10 DEF FN f(n)=VAL(("1" AND NOT n)+("n*FN f(n-1)" AND n))
```

The function above is supposed to calculate factorials (what you get when you calculate n*(n-1)*(n-2)*(n-3).....(1) ) by using recursion within a function. I haven't tried it on the SAM and so don't know if it works, but I wouldn't recommend trying something

like this yourself! Even more disastrously, never make the mistake of doing something like the function below which calls itself for ever! Doing this on the Spectrum resulted in a refined error report, but the SAM, being crude, merely resets itself in a desperate attempt to resolve things as its logic circuits overload. (well, that's what they say in films).

```
10 DEF FN a = FN a
```

I just finished DEF FN, so that means that I'm now on (gets out User's Guide, turns to glossary, flicks to page 139)...

## DEF KEYCODE

I might as well do DEF KEYCODE and KEY together, since they're closely related, and they are not explained terribly well in the manual.

The command KEY is followed by two values. The first of these is a number which describes a particular key-press on the keyboard - such as the key which normally gives "G", the same key + SHIFT etc etc. This must be between 0 and 279, according to the book, but I can't find a key position of 0, and CNTRL+A gives 280, which is allowed. To find the value of any key-press, use the "map" on page 180 of the User's guide. Unfortunately, the keys have got their numbers in, but their normal values are left out. This means that, eg, to find the value of CNTRL+G, you need to count how far to the right of "A" the "G" key is - that's 4 keys - and then count in 4 keys from "A" (which is easy to find) on the map, and eventually see that this square has 34 written in small friendly letters inside. Any of the three shift keys add values. The CNTRL key has a value of +210, which means that pressing it on its own gives nothing, but pressing it with "H" gives 210+34=244.

Of course it's easy to find keys like the RETURN key on the map because it's a massive great corner slab, and you get to learn the values of some of the keys after a while.

Once you've found the value of your key, you need to think about what you want to come out of it. As a case study, we'll say that we want CNTRL+G to give the number 1 when pressed, and we want semi-colon to print up "HAMSTERS ARE GREAT" on the screen every time it's pressed. Looking at the map shows that CNTRL+G has a value of 34+210=244, and that the semi-colon key has a value of 20. (When I say value, I mean map position). The second value that must be entered in a KEY command, as I was saying, determines what character code will be printed by that key. There's a detailed discussion under -> *character set*, but the CODE function quickly shows that the number "1" is character code 49. If we therefore type in KEY 244,49 <RETURN>, CNTRL+G will print up "1". This is an example of a "simple" key definition, and we could continue the whole process to completely muck up the keyboard. Since you can also redefine the DELETE, RETURN keys etc, it's possible to not only muck up the keyboard but to present an unsavoury "hacker" with no option but to press the reset button. In my younger days, I was of the belief that by using the KEY command, I could get Spectrum games to work which used a joystick that I didn't have. Of course, the KEY command only affects the values of keys used by the BASIC editor, and in machine code, things are completely and utterly different.

KEY posn,x. If x is less than 192, the key "posn" will simply print up CHR$ (x) on the screen, and x can be a control code (try experimenting with that one!), or will produce CHR$ (x) in response to a GET or INKEY$ command. If, however, x is between 192 and 254, these characters can be "expanded", so that instead of printing single characters, they print whole strings. The command to deal with this is DEF KEYCODE.

DEFKEYCODE is followed by a number which must be between 192 and 254. This determines the "character code" which will produce the expanded definition in question. The function keys normally use some of the 62 available definitions, and many of the other keys when pressed with CNTRL use them to print up keywords. Going back to the case study, we found the map position of the semi-colon; 20. We can now choose an expanded definition number - how about the first available one, 192. (If you re-define an alreay used one, the original definition will be overwritten. So you can alter all the function keys). If you desire the key to print up a string when pressed, and nothing more, use the syntax:

```
DEFKEYCODE x,s$
```

where x is the "character code" and s$ is the string. In this case, the string is "ANTELOPES ARE GREAT", so typing in DEF KEYCODE 192,"WILDE BEAST ARE GREAT" will produce the desired result. Although you'd find that every time you pressed the ; key, the words BISON ARE GREAT would flash momentarily in the editing area, and then an error report would appear. This would be because the string "RUMINANTS ARE THE BEST" would have been "typed into" the computer, and RETURN pressed. To get round this, the string needs to be ended with a colon - DEF KEYCODE 192,"MY FATHER IS AN IMPALA:". This will definately have the desired effect, although what use you would put it to is unclear. (It doesn't matter whether you type a space in DEF KEYCODE - I'm just being inconsistent).

```
10 KEY 60,200
20 DEF KEYCODE 60: CLS#: BORDER 1: CIRCLE 50,50,50: FILL
   50,50: ZAP: BOOM
```



The lines above would set things up so that whenever key "w" was pressed (position60) a circle would be drawn and a noise made. Pressing W (key position 60+70=130) would just print an upper case W as per usual. This way, you can get keys to type in several commands at once and enter them. (F8 just "types in" LOAD "" CODE ). Other variations of the syntax are:

```
DEF KEYCODE x,"string:"  - produce "string" in editing
                           area
DEF KEYCODE x,"POW:"     - produce "POW" in editing area
DEF KEYCODE x,"POW"      - make a noise like velcro being
                           ripped
DEF KEYCODE x:           - erase expanded definition for
                           CHR$ (x)
DEF KEYCODE ERASE        - should erase all definitions,
                           but doesn't work.
```

I find that defining even a small number of expanded definitions provokes "Too many definitions". Perhaps writing a small loop to erase all existing definition would get round this, as in the first program below.

```
10 FOR n=192 to 254: DEFKEYCODE n: NEXT n
```

The second program line randomizes every key, and I'm sure it will provide hours of fun for all the family.

```
10 FOR n=1 to 280: KEY n, RND(255): NEXT n
```

## DELETE

is a very simple command, but when writing and combining programs, it can be EXTREMELY useful. All it does is to delete parts of programs, which may not sound all that useful, but if the DELETE command didn't exist, you would have to go through every single line you didn't want, deleting it manually. As I used have to do on the Spectrum.

There's nothing else really to say about it. DELETE n TO m will delete line n to m, DELETE n TO will delete the whole program from line n, and DELETE TO n will delete the whole program up to n. If, in a program, a line DELETEs itself, an error message will probably result, but no permanent damage will be sustained. (Why did they change 'Statement lost' to 'Statement doesn't exist'? Sounds much worse!)

## DEVICE

changes the current "device" to another "device". That's a rather simplistic explanation, but effectively that's all it does. A "device" can be either disk, (1/2/ramdisk), tape or network. Not that I know anything about networks. (We are currently considering linking our SAM to a C64 using the midi/network ports - which should be interesting...) The commands LOAD, SAVE, MERGE and VERIFY will then work accordingly, and the specific commands such as DIR, COPY etc, will only work if the correct device is selected. When selecting T for tape, the T can be followed by a number which determines the speed of tape operations. This is peculiar syntax, because it's acceptable to do something like LET a=112: DEVICE ta which is the same as DEVICE t112. Very odd. SVARS 6 and 7 can be used to find out which device is selected - they contain the current device letter (N/d/t) and tape speed or disk number. Typing something like SAVE "d1:CAECUM" will save to disk even if you've selected tape previously, but, for me anyway, doesn't, although because I've seen it used in numerous programs by other people, I assume that it ought to.

Saving and loading routines for tape are contained within the ROM, but the counter - part routines for disk or network are contained within the DOS, apart from the code to BOOT a disk.

Don't think there's anything more to say about it...

## DIM

Many years ago, when I first had my spectrum, I saw the keyword DIM on the grey rubber "D" key, and was of the belief that it dimmed the television picture. Naturally, BRIGHT got it back again. And of course, PLOT thought up a plan, and the functions such as VAL and LEN, SIN and PI were other computers' names or strange adjectives. And you can guess what I thought CAT did.

Anyway, that was about 2000 years ago. DIM, in fact, is an abbreviation of DIMENSION, which "makes" an array, an array being a type of variable. The difference is that while a simple variable such as one called a$ can only hold one value - such as "Crested Guinea Fowl" - an array can hold many different pieces of information.

Just as variables can be either string or numeric, so can arrays. Typing in the line 10 DIM a$(10) would fiddle about with the computer's memory so that A$ would have space for 10 variables. But, their length is fixed. (If it wasn't, things would be slowed down immensely with the amount of memory juggling that would be required). In this examples, we would be given space for 10 variables, each of length 1. Which isn't really much use: it would be more suitable to do something like DIM a$(10,20), which would give us 10 variables of length 20 each. When a DIM is carried out, each array element is "blanked". To get at one particular "subscripted variable", ie, one part of the array, we could do PRINT a$(5), which would print the 5th "a$" string. (Not very good usage of terms, I know, but never mind).

```
10 FOR n=1 to 10
20 READ a$
30 LET a$(n)=a$
40 NEXT n
50 DATA "cerebellum","medulla","myelin"
55 DATA "parasympathetic ganglia bulb"
60 DATA "frontal lobe","sacral","lumbar"
65 DATA "thoracic","cerebrospinal fluid"
70 DATA "loaf"
```

Provided that a$ had previously been "dimmed", and RUN was not used, because RUN clears all variables, this program would assign parts of the nervous system to all 10 elements of the array (a$).
Note that an array and a simple variable can co-exist with the same name. Note also that the name of an array is not just restricted to one letter names - "a$" could be replaced by "nervous$".

```
10 FOR n=1 to 10
20 PRINT a$(n)
30 NEXT n
```

If the program left were used to print all 10 subscripted variables, you would find that some of them had had the ends chopped off - this is just because a$ was previously dimensioned to hold 10 strings of length 10 each - and some of the strings in the first program contain more then 10 characters. When this happens, just the first 10 characters are used. So, eg, PRINT a$(4) would give "parasympat".
PRINT a$(3) would give "myelin " - ie, "myelin" followed by 4 spaces, because the length of each string is ALWAYS as dimensioned, in this case 10. The function TRUNC$ can be used to overcome this. PRINT TRUNC$ a$(4) would chop off trailing spaces. This is only a small string array - a big one might be DIM a$(10,24576), which would accomodate 10 MODE 3/4 screens.
NUMERIC ARRAYS are similar to string arrays except that the name of a numeric array is never followed by a "$" sign. A numeric array with the same dimensions as a string array takes up more memory than the string array, because each element of a numeric array can be a floating point number (which can be practically any number), and which takes up 5 bytes, whereas each element of a string array can only be a character with a code of between 0 and 255. (requiring 1 byte).
There can be any number of dimensions to both a string array and a numeric array. For example, doing DIM hamster(10,5,6,20) would dimension an array which you could picture of consisting of 10 books each with 5 pages, each of which contains 6 lines, and 20 letters on each line. (Very strange books). This would, however, take up a lot of memory, even for these small dimensions, because the length of the array is multiplied by each extra dimension. The same principle applies to string arrays - eg, you could do DIM shifnal$(10,5,6,20), although an array of that size could be acceptable where a similarly sized numeric array would not. There are full details of the way that the data in arrays are stored in the technical manual.
If you DIM a string array - eg, DIM a$(10,20), it's possible to print each subscripted string by just typing, eg, PRINT a$(5). Typing PRINT A$(5,2) would print ONLY the second letter in the fifth string. This only applies to string arrays, and then only to the last dimension.

### *Length*

The function LENGTH is related to variables and array.

DIM b$(46,22) would DIMension an array of 46 strings, 22 chars long each.
PRINT LENGTH (1,b$) would give 46, 46 being the 1st dimension.
PRINT LENGTH (2,b$) would give 22, 22 being the 2nd dimension.
PRINT LENGTH (3,b$) wouldn't work because for one thing there are only two dimensions to the array, and for another, you can't do this anyway.
PRINT LENGTH (0,b$) would give the address of the first byte of the first element.



DIM b(46,22) would DIMension a numeric array which would be like a filing cabinet containing 46 x 22 drawers. (Just like in the Usbourne computer books).
LENGTH would work the same as described above, except that you'd have to put brackets after the name of the array - PRINT LENGTH (1,b()) would be required. Unlike the Spectrum, the error report "variable not found" has been "improved" on the SAM, because it actually tells you WHICH variable was not found, and if it was an array, the SAM sticks a pair of brackets () after the name of the array.

### DIR

stands for DIRECTORY and simply displays the disk directory. Although Bob Brenchley was upset that it isn't called CAT which stands for CATALOGUE, not that there's any difference. DIR on its own prints a quick directory, and DIR1 (or DIR2) prints a directory which file information. DIR! also prints a quick directory, although the reason for this will become apparant in a moment.
If you demand the DIR of a disk, and you get the error "invalid colour" or something,after printing the first filename, which is something stupid like "F9 TO LOAD", you can easily use a program such as the DISK INTERROGATOR on ENC 4 to get around this problem, or use the "selective DIR", where only filenames beginning, containing or ending in certain names are listed. EG, to get a list of screen files, you could try DIR 1;"*.$", which would find all files ending in ".$", which is a commom screen filesuffix, or DIR 1;"s*", which would find all files such as "screen1", "screen2" etc. And so on. And to get around many protection systems, simply BOOT1 the disk (to stop it autorunning), possibly from another disk, and do MERGE "auto*", which will load the loader program without running it. Or you could load the menu thing, remove the disk and select a program - in many cases you'll then be plonked into BASIC. What would be nice is if you could DIR from a certain file onwards...
With normal SAMDOS, it's very easy to get the directory into a string. First pick a stream (let's say 10), so then do CLOSE #10: OPEN #10;"$". This opens stream 10 to the "$" channel, which is used when RECORDING TO strings. Next type RECORD TO a$ (or another variable) and then DIR #10;1 . Wait a bit, RECORD STOP and then PRINT A$ will give the directory. If you want a short directory, it'll be necessary to do DIR #10;1! - and that's why the exlamation mark is used. It's also in case you want to do, eg, DIR 2! - ie, get a short directory of drive 2 whilst in DEVICE d1.
The directory normally occupies the first 4 tracks of a disk, and since each directory entry takes 256 bytes, each track contains 10 sectors and each sector can hold 512 bytes, this means that you can have 4x10x2 = 80 files per disk. Usually. These 256 bytes are used to store the file type, length, start, name etc, but most of it hold the "sector address map" for the file. This tells the Coupe exactly which sectors are used by a file. If you ERASE a short file, and then SAVE a longer file, some of the data will be put where the old short one was, and the rest of it will probably go elsewhere. The sector address map makes this possible. A bit address map is calculated by the DOS when a file is SAVED, by combining the sector maps of all existing files on a disk, to see whether there is space for it.
If you use MC and DOS hookcodes to save to disk from MC, a "mini" directory of the file in question must be set up, called the UIFA. This is pointed to by IX and holds details of file length etc etc. Details in technical manual. 8 bytes within each directory file are spare, and these can be used to extend filenames, or save extra information about the file, although there is no way to use this from BASIC.

### DISPLAY n

simply displays screen n, which is not necessarily the same as the screen being worked on. A screen needs to have been OPENed before it can be displayed, obviously. SCREEN n selects a screen to "work on", which is not necessarily the same as the one being displayed.
When you reset the Coupe, or type DISPLAY on its own, the command SCREEN automatically also does a DISPLAY, so that the displayed screen is always the same as the current screen. It's only after a DISPLAY n command has been executed that the two become separate.
The current screen, selected by SCREEN n, is the one used for printing, graphics etc etc, and if you print or draw onto a screen which is not being displayed, then it will not be evident that this activity is going on. The commands can therfore be used to "hide" graphics activity and display it when it is finished. Whilst the SCREEN command alters a system variable (&5A78), which is used whenever DRAW etc decide which memory page to use, the DISPLAY command alters another system variable (&5A77), and also alters the I/O port 252, which determines which memory page is to be used by the video circuitry. So it's easy to DISPLAY closed screens just by writing to this port. But that's another story altogether...
When you OPEN a screen, it also updates the Page Allocation Table at &5100 - which tells the Coupe what each memory page is used for, a screen page being &C0 - and also the screens list (SCLIST) at &5ca0. This is 16 bytes long and contains information on memory page (bits 0-4), and screen mode (bits 5-6) for each screen. If you then select a screen with SCREEN n, the system variable

SCPTR, at &5C9D is updated so that is points to the position in the SCLIST table of the current screen (absolute address). So a round-about method of finding the memory page of the current screen would be to do

```
PRINT (PEEK (DPEEK &5c9d))BAND 31.
```

Much quicker do to IN 252 BAND 31, but this only shows the page of the DISPLAYED screen.
On a 512K Coupe, 16 screens are available, and by drawing different stages of animation on each screen, and then using DISPLAY to flick between them, smooth animations can be produced. Every time you deselect a screen, ie, you choose another screen with SCREEN n, a block of system variables from &5A34 to &5A6E is stuck at the end of the screen data, in its memory page. These contain information on pen/paper colours, CSIZE, windows, xrg/yrg etc etc, and allow each screen to have its own attributes. It was very noticable with the old ROM, and still sometimes evident, that this can be a source of problems, particularly with palettes, which are also stored with each screen - ie, one screen's attributes would work their way into another screens. You should therefore use DISPLAY regularly to reset the displayed screen.

## DO-LOOP

A DO-LOOP structure is very similar to a FOR-NEXT loop, though is rather simpler. The three program segments listed below are essentially the same except that they each "end" in a different way. Whenever the interpreter comes across a "LOOP", it jumps back to the nearest DO, and starts again from there. It would go on forever, if it were not for the three qualifiers which will "end" a loop.

```
10 DO
20 GET a$
30 PRINT a$;
40 LOOP UNTIL a$=" "
```

```
10 DO UNTIL A$=" "
20 GET a
30 PRINT a$;
40 LOOP
```

```
10 DO
20 GET a$
30 EXIT IF a$=" "
40 PRINT a$;
50 LOOP
```

The simplest being EXIT IF, demonstrated in the last example, which will skip to the statement or line after the next LOOP $ if the condition following it is true. In this typewriter program it would end if you press space, which would not be printed. (NB: use EXIT IF rather than GOTO 60 or something, because it knocks the DO line off a storage area which would otherwise get confused. And don't confuse it with END IF, which is completely different).
Both DO and LOOP can be followed by either UNTIL or WHILE, which pretty much what they sound like they do. IE, a loop will LOOP or be DONE UNTIL a condition is true, or a loop will LOOP or be done WHILE a condition is true. LOOP UNTIL a$=" " is the same as LOOP WHILE a$<>" ", ie, a WHILE condition is true when an UNTIL condition is false. In the first 2 programs here, the space will be printed, which may be undesirable. DO UNTIL CODE INKEY$: LOOP waits for a key-press.

## DPOKE and DPEEK

are double byte forms of POKE and PEEK which might as well be discussed here as well.
POKE puts a number between 0 and 255 (a byte) into a memory ADDRESS. This address can be between 0 and 540671 on a 512K machine, or 0 and 278528 on a 256K machine. But POKing an address below 16384 has no effect because these addresses usually make up the ROM, which cannot be changed. (When one says a 512K Coupe, that's 512K RAM - the total amount of memory is 544K). The byte you put there will stay there until either the computer is switched off, or another routine alters it - for example, if it is part of the screen display, a BASIC program etc etc.
PEEK is a function and shows the value of a byte in a particular address. Eg, POKE 30000,24: PRINT PEEK 30000 would print 24 on the screen.
Actually, it's possible to POKE an address with a value from -255 to 255, but negative values are subtracted from 256. What's the point of that? Not much unless you're writing MC progs from BASIC and you want to do a backward DJNZ.



Because a byte can only store a number from 0 to 255 (work out why), two bytes together can store numbers up to 65535, in base 256 if you like. The second byte, (the most significant byte) stores the number divided by 256. The first byte, (the least significant byte) stores the number modulo 256. Ie, the number 550 would be stored as 38,2. (38 + (2*256) = 38+512 =550)).
DPOKE pokes two address with two bytes, then, and you use it like this - DPOKE address, word. With "word" being a number from 0 to 65535. It's the equivalent of POKE address, word-INT (word-256)*256: POKE address+1,INT (word/256), and as you can see is a damn sight quicker. DPEEK is the double byte equivalent of PEEK and returns a number between 0 and 65535. DPEEK (address) and DPEEK (address+1) use parts of the same number, and therfore give completely different results.

To store numbers over 65535, there are three ways.

1) Carry on using base 256, such that the "really significant byte" would hold the number/65536.
2) The first byte holds a page number. Each page contains 16384 bytes. This is the method used by the ROM for storing certain addresses.
3) Use Floating-point form, which allows both hugewapping gigantic numbers and tiny minute microscopic numbers to be stored to within a certain degree of accuracy in 5 bytes. And I don't understand it properly.

## DRAW

is a pleasant command which draws straight and curvy lines. It has several variations:

```
DRAW x,y      - draws a line x pixels right and y pixels up.
DRAW x,y,z    - draws a line x pixels right and y pixels
                up and turning through angle z
DRAW TO x,y   - draws a line that ends at the point (x,y).
DRAW TO x,y,z - same as DRAW TO, but turns through angle z.
```

In the first two, if x or y is negative, then the lines goes left or down.

```
10 LET x=0,y=0,p=50,q=50
20 LET x1=p-x,y1=q-y
30 LET d=SQR (x1*x1+y1*y1)
40 LET c=INT (d+.5)
50 LET a=s*x1/d, b=s*y1/d
60 FOR f=1 TO c
70 LET x=x+a, y=y+b
80 PLOT x,y
90 NEXT f
```

This program shows one way of drawing a line using PLOT. Which doesn't sound very useful seeing as we've got a DRAW command, but you can alter it to give dotted lines etc.
The system variables at &5a41 and &5a42 hold the current graphics position (variables x and y in the program - p and q are the point to draw to).
DRAW TO x,y,z or DRAW x,y,z turn through a curve, whose "curvosity" is z radians. There are (pi) radians in half a circle, and so doing DRAW 50,50,PI would draw half a circle. DRAW 50,50,PI/2 would draw a smaller section of a circle, and DRAW 50,50,2*PI would give an error. But values greater than 2*PI will work. Negative values will make the curve "go the other way." So, to get a semicircle thing, you could type PLOT 50,50: DRAW 50,50,PI: DRAW TO 50,50. Very nice.
There are hundreds of things you can do with DRAW; it's used all over the place. If you do FOR n=0 to 255: PLOT 128,88: DRAW TO n,0: NEXT n to "sweep" a line across the bottom of the screen, a pattern will result, because gaps get left between neighbourouring lines; this is always true, and you should remember it for the rest of your life.

## DUMP

You need two things for this to work - the printer dump software, and a printer. It's the equivalent of COPY on the spectrum, and simply DUMPS the screen display onto a dot matrix printer; it works in any mode. There are a whole range of system variables associated with this - listed in the Users guide - such as the size you want to dump to end up, which part of the screen you want dumped etc. You can also change the message sent to the printer before and after each row, if the normal message does something that you don't want it to. The rountine is small and fast, and in MODE 3 or 4, simply prints any point that isn't 0 as a

point. In many cases this works perfectly OK, but in others, the screens turns into garbage. The routines in FLASH! are slower, but are also clever because they work out the luminosity of each SAM colour and print it "shaded" accordingly.
There MIGHT JUST be, but probably not, a program somewhere on this disk to DUMP special hi-res screens, which use memory separate from screen memory. If there isn't, get inspired and do it yourself.
There will more than likely, though, be a STAR LC200 colour dump routine somewhere on ENC 5, which, while rather basic, can give fairly good results. More details on this elsewhere. DUMP CHR$ searches through the screen and sends to the printer the codes of any text in the current char, set on the screen, which are then printed in the printer's current font/pitch setting. A fairly clever command actually, which must be rather difficult to write.

For **ELSE, ELSE IF** and **END IF** see **IF.**

For **END PROC,** see the section on -> *procedures* at the start of this booklet.

For **EXIT IF,** see **DO,** which is a few pages back.

## ERASE

is an extremely simple command which surprisingly, erases files from disks. It only erases their directory entries, so by using a program like the Disk Interrogator on ENC 4, they can be recovered.

## EXP

is a mathematical function which calculates exponential numbers. It's followed by a number, and EXP 1=e (the e is meant to be written strangely), and e is a constant which starts off 2.718... (similar to PI, which is a constant used for various purposes and is a non-recurring decimal).
EXP n means e ^ n, ie, e to the power of n. What I don't really know is what you'd actally use it for (apart from drawing patterns with).

## FATPIX

is an extremely stupid command (sorry if I've offended anyone), but I NEVER use it. As the technical manual elegantly puts it, in mode 3, "thin" pixels are used, because there are twice as many of them horizontally as there are in mode 4. The horzontal scale is therefore doubled, to 512. (see XRG). So if you have a program which plots a shape using PLOT, CIRCLE or DRAW, it will look squashed up in Mode 3. By typing FATPIX 1, XRG is halved, and the pixels become "fat". I don't really see the point of this, unless it was just to utilise a useful system variable or something, because doing LET XRG=256 has virtually the same effect, and if I may say so, I wouldn't want to have fat pixels.

## FILL

is a lovely command, and also one that I would hate to have to program. The SAM version is very fast, but you can watch it in operation carefully by using the "slow-downer" on ENCELADUS 5. All that it does is to FILL in shapes with colour, which may sound simple, but to program it is horrible. (I make it sound as if I programmed SAM BASIC, which I didn't).
FILL x,y starts filling in the current pen at co-ordinates (x,y). It will colour all points that are the same as the starting point until a boundary is reached. If you use FILL USING a$;x,y, a 16x16 pixel pattern is used to fill. This is superior to the fill function in FLASH, because in FLASH!, the fill patterns are monochrome only, although this is to reduce problems in modes 1 and 2. The easiest way to get a pattern is to GRAB it. EG, to get a stripes pattern, you could use the program below.

```
10 FOR n=0 to 16 STEP 2: PLOT 0,n: DRAW 16,0: NEXT n
20 GRAB pat$,0,16,16,16
30 CLS: CIRCLE 50,50,50
40 FILL USING pat$,50,50
```

Alternatively, to fill something with a load of junk, you can either type in FILL USING "sfh2ot8fs";0,0 or something along those



lines, or do FILL USING MEM$(0 to 1000);0,0. (If string is too long, then just the first part is used, and if it's too short, then part of the previous pattern is used.
Care must be taken when using FILL, because if just one pixel is missing from a shape boundary, the colour will leak out into its surroundings.
Every time FILL is used, a "scratchpad" area is created in the computer's memory, and this causes the delay before the FILL starts. If you are FILLing several shapes, each of which has the same background, a "1" can be placed after the FILL - eg, FILL 100,10,1 and the same scratchpad will be used. Although sometimes this behaves strangely...
What I sometimes do is get a REALLY complex shape on the screen (such as loads and loads of random dots) and then FILL around it, to see how the Coupe copes, sometimes using the slow-downer. Watching it FILL mazes, or spirals etc is very interesting. It's possible, if the shape is REALLY REALLY complex, to make the Coupe crash in confusion, but this is cruel. You will observe that it FILLS in, eg, one side of the shape at a time, and then goes back to all the bits that it missed. The other way of FILLING is to use the "flood" method, where every part of the shape is simultaneously filled, but this is less fun to watch. FILL in mode 3 works on half of the screen at a time, so you usually need two FILLS. (If it didn't, more memory would be needed to make the "scratchpad"). Definately one of my favourite commands, and the sign of a very good BASIC.

## FLASH

only works in modes 1 and 2, and acts like PEN or PAPER. If you print something using FLASH 1, it will "flash", for some unkown reason. The computer doesn't flash it, as with flashing palettes, but the video circuitry handles it. Attribute bytes in modes 1 and 2 have bit 7 set if they are flashing. It's not possible to change the speed of FLASHING colours.

## FOR - NEXT

loops are a fundamental principle of BASIC and are a more advanced version of DO - LOOP. The three programs below demonstrate uses of FOR-NEXT. When you do a FOR statement, a special variable is set up, which can be separate from a similarly named normal variable. In these three examples, it's "a", though it can be longer in length. (You normally see "n" used, this just stands for "number"). The variable is followed by a START and a LIMIT, and every time a NEXT is met, the program loops back to the line or statement after the FOR, while the variable is increased. This continues until the variable equals or exceeds the limit, in which case the statement or line after the NEXT is dealt with.

```
10 FOR a=1 TO 600
20 BORDER (a MOD 15)+1
30 NEXT a
```

```
40 FOR a=32 to 255
50 PRINT CHR$ a;
60 NEXT a
```

```
70 FOR a=1 to 500: NEXT a
```

The three examples "overleaf": the middle one prints the character set, the bottom one is a simple pause, and the top one flashes the border. The top one has "a" going from 0 to 600, but as the border can only be 0-15, MOD is used. ("a" still goes up to 600, but the MOD divides it down for the BORDER command).
STEP can be placed after the limit as in FOR n=0 to 50 STEP 2. This does what it suggests, and determines the step at which the FOR variable will alter. In this case,it would increase as such: 0,2,4,6,8.....46,48,50, and obviously the step can be any value, including fractions and negative values, in which case the variable will decrease.
Just a point - say you want to inspect an area of memory, for example the first two lines of the screen display. Some people would type

```
FOR screen_display_counter=507904 to 507904+(128*2):
.....:
.....:
NEXT screen_display_counter
```

Well this is completely and utterly stupid, and you should follow my example and do something more like FOR n=507904 to 898219442 which saves a lot of time (and memory). If it's for your own purposes, ie, not to go in a program, there's no point in working out the limit precisely.

## free

PRINT free shows simply how much memory is left for BASIC. FREE is a special variable, so you can't do, eg, LET free=5. Its value will depend on the size of your program and variables, RAMTOP, and how many memory pages are open to basic.

## GET

gets a key into a string. Unlike the BBC, which uses LET a$=GET$, the Coupe uses GET A$, which is much better. In this case, the Coupe will wait for the user to press any key (apart from ones like SHIFT), and will put that into the variable "a$". Ie, if you pressed "F", a$ would become "F". GET automatically accounts for whether SHIFT/SYMBOL is pressed, and if you redefine keys, as described earlier, these new definitions will take effect. (But expanded definitions will just appear as, eg, CHR$ 192, which is a blob). When you press a key for GET, the Coupe also makes the key-press sound which it does in BASIC. (Well, it's meant to, but mine doesn't, not until I poke the system variable for it).
GET followed by a numeric variable - eg, GET a, gets a number according to the pressed key. Pressing "1" gives 1, "2" gives 2 and so on, and pressing either "A" or "a" gives 10. So this makes it useful for using in menu systems where there are more than 10 options. EG, you could use the program fragment below, which GETS a key value, and then jumps to the relevant part of the program. (Line 1000 for "0", 1500 for "5", 2100 for "b" and so on).

```
10 GET a
20 GOTO 1000+a*100
```

You can also use it as a type of typewriter.

## GO TO, GO SUB and LABEL

These are all related commands and perform the simple task of jumping from one line of a program to another. In a BASIC which allows procedures, such as SAM BASIC, it is possible to write programs without using any of these, by the careful use of structured programming, but if, like me, you can't stand structured programming then you'll be using them all over the place.
GO TO can also be used to start a program without clearing the screen and variables which can often be useful.
GO TO n jumps to line "n" in a program, and if "n" doesn't exist, then the nearest line to "n". If "n" is beyond the end of the program, the computer will stop with the OK report. Lesser BASIC do not allow jumps to non-existant lines, which can be a realpain. When a GOTO is used, two SVARS are used for the LINE and STATEMENT to jump to. These are &5C42 (2 bytes) and &5C44 respectively. Poking the former has little effect but by poking &5C44 we can force a jump to a particular statement within a line. So lines 10 and 20 both jump to line 50, but doing so from line 20 has a totally different result.

```
10 GO TO 50
20 POKE &5C44,3: GOTO 50
30 ...
50 POW: STOP: NEW
```

### *Gosub*

GO SUB is short for "go to subroutine" and acts as a kind of poor man's procedure. When a GO SUB is encountered by the BASIC interpreter, the line no with the GO SUB in is placed on a STACK so it can be recalled later. GO SUB is followed by a line no. as with GO TO, and this line no is then jumped to. So far it's the same as GO TO, but if GO SUB is used, a second command - RETURN - can be used to jump back to the line containing the GO SUB. The small program below demonstrates this, and line 20 shows that RETURN can jump back to a particular statement in a line. Rather than type in the graphics commands each time, the SUBROUTINE at line 50 can be called each time instead. Although this program is rather meaningless.If you keep everything neat, each RETURN should jump back to the right GO SUB... however if you mess things up by using GO SUBs within subroutines (recursion, which is perfectly alright) too many times etc, the stack may overflow and things will start to go wrong. The command POP unstacks the last value on the stack into a variable. So, in a subroutine, RETURN could be replaced with POP x: GOTO x.

```
10 GO SUB 50: INK 2: GO SUB 50: OVER 1: GO SUB 50
20 FOR n=1 TO 10: GO SUB 50: NEXT n: STOP
30 ...
40 ...
50 CLS: CIRCLE 50,50,50: FILL 50,50: RETURN
```



### *Label*

LABEL is simply used to give parts of programs names, especially for use with GOTO and GOSUB. I'm not an unintelligent person, but it took me ages to "figure out" how to use it as the manual is a bit ambigious. The two programs below demonstrate simply the principle: in this case using LABEL is a bit pointless because it is most useful in large programs. Anyway, all it does it to allow GOTO and GOSUB to use NAMES rather than numbers. You could also do this by just doing, eg, LET x=30: GO TO 30, but this would not allow for renumbered programs etc. If you have a variable with the same name as a LABEL, they will overwrite the actual LABEL. So, eg, adding 5 LET yo=60 to the second program would result in line 60 being jumped to.
LABELs are integral to the actual program, so when a program is renumbered or whatever, they will still be in the correct relative positions.

```
10 GO TO 30
20 STOP
30 PRINT "YO!"
```

```
10 GO TO yo
20 STOP
30 LABEL yo
40 PRINT "YO!"
```

## GRAB and PUT

These are probably two of the most "feature" commands available in SAM BASIC and make it easy for animation and graphics work to be carried out from BASIC. This is something which other computers lack, and seems to constantly amaze non Coupe owners!

### *GRAB A$,x,y,w,l*

The above shows the syntax of GRAB. "A$" represents the name of a variable which will be used to store an area of the screen. x and y are the co-ords of the top left corner, w is the width in pixels and l is the length in pixels. The x co-ords are only accurate to two pixels (four in mode 3), but this speeds things up considerably. When the GRAB command is entered, the area of the screen specified as described above is stored into a string - in the form of a couple of characters to show the width & length, followed by the raw data taken from the screen. It's possible to compress GRAB strings using a similar technique to screen compressing, but I haven't yet tried that because it would mean fiddling about with variable addresses etc.
If the area specified is too large (more than about 1/3 of a screen), the GRAB will not be allowed. This is because the ROM GRAB routine places the data in an 8K area of memory: the screen is 24K in size. To get around this you can just make two GRABS, each storing a half (or less) of the relevant area. An equivalent method to store certain area is to use MEM$ and POKE - see these commands for details.
If you want to use sprites in M/C, the same ROM routines can be used. The actual GRAB and PUT routines will not run any faster, but the linking which provides the co-ords, movement etc, will be much faster and so very fast and smooth animations can be produced. The ROM routine is located at 0136H and grabs the area specified by the C and B registers (x and y co-ords, 0 at the top), with E giving the width in bytes (ie, half the width in pixels) and D the length in pixels. On exit, DE points to the start of the block, and BC gives the length. The start of the block will usually be 57344 dec (65536-57344= 8K), so if you're going to use several sprites, you will need to move the sprite to somewhere else. If they're small enough they can just be moved to eg, 60000, in the same memory page.
BASIC will deal with addresses of GRAB blocks because they are in variables, so you can ignore all of the above! You can thus store the equivalent of about 16 mode 3/4 screens with a 512K Coupe in GRAB blocks. If you have MasterBASIC you can save them to RAMdisk and compress them, allowing even more to be conveniently stored. If you want to store a horizontal slice - eg, from x co-ords 0 to 255 - remember that this will be GRAB a$,0,y,256,l - ie, 256 rather than 255. The same applies to vertical slices. In CSIZE 8,8 with YOS = 0, store a quarter screen with:

```
GRAB A$,0,175,64,191
```

YOS and YRG and CSIZE can get a bit confusing!

PUTwill in due course place a GRABbed block back on the screen. You can try PUTting other types of strings, but the Coupe isn't as stupid as it looks and will usually refuse.

### *PUT x,y,a$*

x and y are simply the co-ordinates of the top-left corner to place the block and A$ is the name of the variable. Remember that GRAB doesn't store the colours of blocks, and you should therefore make sure that the palette is the same as the screen used for the GRAB unless you specifically want different colours. However you can use PUT in conjuction with OVER or INVERSE - eg, PUT INVERSE 1; OVER 2; 80,80, A$. Just experiment with these to see how they work.
PUT can also use masks. A mask is a version of a sprite or graphic which determines which part of the sprite should actually be placed on the screen, and which pieces should remain "transparent". Using a mask allows complex graphics to be place on the screen without a big square blanking out everything else. A PUT mask must be the same size as the accompanying graphic. The user's guide gives a short program to make a mask from a sprite (using FILL, GRAB, PUT, INVERSE), but generally any PEN 15 pixels in the mask will cause the corresponding pixels in the sprite to be placed on screen, and any PEN 0 pixels will remain "transparent" in the sprite. It's actually a bit more complex than that, but you get the idea.
In M/C, the PUT routine is located at 0133H. As with GRAB, the C and B registers hold the x and y co-ords. HL points to the start of the data. The A register determines which method of PUTting will be used (XOR, OR, AND etc - similar to OVER, INVERSE in BASIC), or =5 to use a mask. In this case, HL' points to the address of the start of the mask.
The demo a few pages ago (and running from a disk) uses the mask option in MC. This enables the object to move over the background without leaving a trail or making holes everywhere. To stop flickering, the program loops until the TV scan-line (read from I/O port 504) has just past the current y co-ord.
PUT with a mask in BASIC uses the syntax: PUT x,y,a$,m$ where M$ is the mask. There are various demos of this in various places, so if you don't understand any of this, just look at these programs. The normal variety of PUT is used all over the place. As a GRABbed variable is saved with a program, there is no need to GRAB a graphic every time a program is run - it only need be GRABbed once. That is, provided you don't use RUN or CLEAR which will delete all variables.
GRAB and PUT only work in modes 3 and 4, but by poking SVAR 64 (as explained elsewhere), you can use them in any mode - though obviously this won't have much benifit in modes 1 or 2.

### HEX$

is a useful little function that converts decimals to hexadecimal numbers. Since the Coupe will accept hex numbers in BASIC (by prefixing them with "&"), the only real use of HEX$ is to find out for yourself what a dec number equals in hexEg, PRINT HEX$ 67321 will print up 0106F9 on the screen - to get it into a variable you must do something like LET A$=HEX$ 32768. Even though the result is a "number", it may contain letters, and this is why a string variable is used to store it.
To use the HEX>DEC conversion, just do something like PRINT &0106F9 - this would print up 67321, and the result is a normal numeric variable. See ENC 3 for details of BIN and BIN$ which have similar functions.

### IF

is a conditional instruction (it says in the book). It is used to test conditions and do certain things depending on the result. It is, however, possible to write a program requiring conditional testing but not using IF by the careful use of things like: PRINT "true" AND var=4 and so on, but IF is obviously more convenient, but another disadvantage is that is tends to slow down programs quite a lot.
The condition following the IF is ultimately either true or false. If is it false, then the next line is jumped to. (NOT the next statement). First of all a check is made to see if there is an ELSE in the line in which case this is jumped to. If the condition is true, then the command(s) following the THEN which follows the condition are executed.
The actual conditions can consist of comparisons - such as = (equals), > (greater than), < (less than), >=,<= (greater/less than or equals) and <> (doesn't equal) or can simply be the name of a numeric variable - in this case, the result is TRUE is the variable is not 0. Otherwise the result is FALSE.
In the program below, three POWs will occur. (lines 50, 60, 80). All the other conditions are false.

```
10 LET c=0, d=1, e=5, f=100
20 IF d=e THEN POW
30 IF c>d THEN POW
40 IF f<e THEN POW
50 IF f>=d THEN POW
60 IF d THEN POW
70 IF c THEN POW
```



```
80 IF d<>e THEN POW
90 IF NOT e THEN POW
```

The second program line shows how more than one condition can be tested at once by using AND. When this is used, the command(s) following the THEN are only executed if BOTH conditions are true. In this example, though, if the first is true, then the second must also be true, and vice versa. Of course any number of conditions can be tested:

```
IF a=5 AND b=10 AND a$="wildebeast" AND c THEN STOP
10 IF d<e AND e>d THEN POW
```

for example. Similar to AND is OR - this does what is suggests and the total condition is TRUE if one OR the other component conditions are true. Eg:

```
IF p$="Labour" OR p$="L Dems" THEN NEW
```

The Coupe will reset if it finds either of the two conditions to be true...wouldn't you?

NOT can be used to negate a condition: ie, the condition preceded by NOT will only be true if it is false. Understand? An alternative to NOT would be something like

```
IF (a>=b)=0 THEN STOP
```

which would only STOP if a was less than b... Or you could do IF NOT NOT a>b which would carry on as normal. If you use several ORs and ANDs then brackets may be required...

```
IF (a OR b OR c OR d OR NOT e) AND ((a$="hamster" OR
a$="russian dwarf hamster") OR (a$="wildebeast" AND
name$="Frank")) AND (a AND b AND b>a) AND ((f OR g) AND
h=5) AND (a*b+b^(c/2)+PI/(a^.023)-SQR e)=42 AND (NOT d OR
d=5 OR d=a OR d=VAL "3") THEN ZAP
```

And so on. All the above, from after the IF to before the THEN could be put into a variable (such as con$), and then rather than type in all that several times you could simply do IF VAL con$ THEN ZAP each time which would have exactly the same effect but would be slightly slower. In fact you can INPUT a condition and use VAL to see if it is true or not.

ELSE can be used to deal with false conditions.

```
IF a=5 THEN ZAP: ELSE STOP
```

In this example, if a=5, then the computer will "zap", else it will stop. The command(s) after an ELSE can include more IF's, and things can get a bit complex if you use too many IFs and ELSEs. You can easily do an ELSE IF as a kind of OR, or you can use long IFs!
These consist of an opening IF - such as IF A$="hamster" some commands can then follow on the next line(s). If the condition is false - ie, a$ does not contain "hamster", then these lines will be skipped until an ELSE IF or END IF is found. The latter will end the session. An ELSE IF acts as another IF and will contain a second condition to check. The program below illustrates this.

```
10 IF A$="hamster"
      15 PAUSE 50
      20 POW
25 ELSE IF a$="bird"
      30 FOR n=1 TO 20
      35 BEEP .1, RND (30)+30
      40 NEXT n
45 ELSE IF a$="lion"
      50 BEEP 3,-50
55 ELSE IF a$="rainforest bird"
      60 ZOOM: ZOOM: ZOOM
70 ELSE IF a$="frog"
      80 ZAP: PAUSE 10: ZAP
85 END IF
90 STOP
```

(The last two lines wouldn't fit on the screen...bah!) The program, will, to the best of the Coupe's ability, make noises to simulate particular animals. (It's quite good actually - press SPACE to hear). The POW in line 20 for the hamster is the hamster scuffling about in a bit of sawdust.
The long IF is ended by an END IF. The line following the END IF is jumped to when a true condition has been fulfilled - eg, when the hamster's noise has been made, line 90 is jumped to. If a$ does not equal any of the creatures specified, no sound will be made, but an extra ELSE after line 80 could deal with any extra values of A$.

## INK

I use INK all the time instead of PEN but look under PEN for that.

## INKEY$

There are three methods of reading the keyboard in BASIC - and they are to use INKEY$, GET or IN. GET is explained elsewhere. INKEY$ is a function which returns a single character; this character comes from the keyboard and can be a symbol or shifted character if SHIFT/SYMBOL/CNTRL is also pressed. Only one key at a time can be read - if several keys are being pressed then the result will alternate between the two.
Unlike GET, the normal INKEY$ does not wait for a key to be pressed and therefore can be tested to see if a key is being pressed. The first two programs below show ways to loop until a key's pressed.

```
10 DO
20 LOOP UNTIL INKEY$<>""
30 DO
40 LOOP UNTIL CODE INKEY$
50 PRINT INKEY$ #0;"_";CHR$ 8;
60 GOTO 50
```

If the INKEY$ function is followed by a stream number (as in the third program), INKEY$ will appear to wait for a key to be pressed - although in fact it is producing CHR$ 0's which when printed like this have no effect...although they do when you do something like PRINT CHR$ 0. The program also places a cursor after the current character and a backspace code. In fact INKEY$#0 can be used as a faster version of GET...
The disadvantage of using INKEY$ is that only one key can be read at once and hence diagonal movement can not be implemented: one way to get round this is to use the IN function. This is also a bit quicker than INKEY$.

## *IN*

is similar to PEEK, except that rather than reading the contents of a memory address, it reads an I/O port. There are, in effect, 65536 of the little blighters and some have particular purposes, although because of the way they're arranged most of them repeat themselves. Not all can be both read and written to, and some give different results depending on whether they're read - using IN, or written to - using OUT. Writing to I/O ports allows the user to control hardware, and it is the only way to do so! Ports exist to enable the controlling of the disk drive, the printer, external interfaces, the sound chip, memory paging, the CLUT, the MIDI portsthe BORDER, EAR and MIC and so on. There are equivalent ports to read the status of some of these devices but obviously it's a bit difficult to read the status of, eg, the sound chip. But's it possible to see if the printer's ready, and that sort of thing, as well as more complex things requiring external input.
PRINT IN address is one way to use IN: its result is a value between 0 and 255. "address" can be between 0 and 65535. Ports below 256 are "base ports" and adding 256 to their addresses will give other ports which have the same effect. It doesn't work quite this way though, and I don't quite understand it...

## *I/O ports*

For instance, the CLUT's base port is 248, which itself controls colour black. Ports 504, 760 and so on (248+256, 504+256) control the other colours. In BASIC, to use them you could just do, eg, OUT 1016,44 to turn colour 3 to palette colour 44. However it will immediately flick back to the previous colour because the ROM uses a table of colours which it uses to update the CLUT, and so the command OUT 1016,44 must be performed repeatedly. This also occurs in M/C unless interrupts have been disabled using interrupts. The instructions would be LD C,248 : LD B,3 - so BC=1016 and then OUT (C),A. The same commands, but using IN A,(C) could be used to read a port. The CLUT, incidentally, cannot be read - this is why the table is needed.



```
65534  cursors
65278  SHIFT - V
65022  A - G
64510  Q - T
63486  1 - 5
61438  0 - 6
57342  P - Y
49150  RETURN - H
32766  SPACE - B
```

The table over to the left shows which ports can be used to read the keyboard. In MC, they can also be used, although there there are other methods. Reading these when no keys are being pressed will give a result of 95. Bits 0 to 4 are each "controlled" by a different key in that half row. So key F alters bit 3 of port 65022, key A is bit 0 of the same port. When A is pressed on it's own, IN 65022 would give 94 - when S is pressed it would be 93, D would give 91 and so on. Pressing A and S at once would give 92. These can therefore be used to read the keyboard, as in the next program...

```
10 LET x=0,y=0
20 DO
30 PLOT x,y
40 LET x=x+(IN 57342=93)-(IN 57342=91)
50 LET y=y+(IN 64510=94)-(IN 65022=93)
60 LOOP
```

This primitive program moves a point about the screen controlled by the keys QSIO. It's easy enough to adapt it to test if the point will fall off the screen and so on.
In M/C, to test if SPACE is being pressed, one way would be as below, in which case a RET would be performed if SPACE was pressed - regardless of whatever else was being pressed.

```
LD BC,32766
IN A,(C)
BIT 0,A
RET Z
```

If the instruction BIT 0,A was replaced by CP 94, the RET would only be performed if SPACE was pressed but not at the same time as either B,N,. or SHIFT.

Since I seem to have started talking about OUT over the page, I might as well give a list of the most important I/O ports...

Ports 254 and 249 can be used to read the keyboard as described above but actually it's a bit more complex than that. Use Port 249 to see if, eg, the function keys are being pressed.
When written to, Port 254 controls the border (bits 0,1,2,5), the microphone (bit 3) the "beeper" (bit 4) which are used with SAVE to cassette. Bits 6 and 7 control MIDI through operation and the screen disable function respectively. By setting this bit (like OUT 254,128) the screen will go "off" and the computer will speed up a little. It will usually reset when a key is pressed.
Port 253 is concerned with MIDI-IN and MIDI-OUT which I won't dwell on.
Port 252 is a read/write port and controls the video paging. Bits 0 to 3 determine which memory page (0-15) is used whilst bit 4 is set to select the extra RAM bank for 512K machines. Bits 5 and 6 control the video mode (0-3). Bit 7 is more MIDI rubbish.
Port 251 pages "high" memory - ie, controls which pages should be placed at addresses 32768-65535. Bits 0 to 4 control the page and bank, as before, and if bit 7 is high then the Coupe uses external memory. (Like the 1Meg). Bits 5 and 6 allow more than 4 colours to be used in mode 3. In this mode, each screen pixel occupies a quarter of a byte, and these two bits allow for four colours. Another two bits come from Port 251, allowing 16 colours. As they are normally reset, the four colours used in mode 3 are mode 4's colours 0,1,2 and 3. But if, eg, you did OUT 251,96, then they would refer to mode 4's colours 12,13,14 and 15. Obviously this isn't an awful lot of use, but by changing port 251 quickly in M/C, the screen can be split into several parts - a bit like palette LINEs. It also provides another way to access more than 4 colours in MODE 3 without using the PALETTE command.
Port 250 is similar to Port 251 and controls the paging of "low" memory - 0 to 32768. Bits 0 to 4 are the page and bank as usual. If bit 5 is set, then RAM replaces the ROM - so alternative ROMs could be used. If bit 6 is set, the Coupe's second ROM (the one that deals with all the graphics etc) replace RAM in addresses 49152-65535. Finally, if bit 7 is set, write protection of any RAM which has replaced the ROM is enabled - so you really can install your own ROM. When SAM>Spectrum utilites place the Spectrum ROM into the Coupe's low memory, they use this bit to write protect the Spectrum ROM. But by unsetting it (and you can do so whilst in

Spectrum emulation) you can POKE the operating ROM, which could be interesting. (There's no way you can POKE the Coupe's ROM!)
Port 249 when read gives information on the status of various interrupts and isn't of much use, and can also be used to read certain keys. When written to, it controls the line interrupt - but apparently this doesn't work...well I can't get it to.
Reading Port 248 gives the co-ords of any connected light-pen - if there's no light pen then it is constantly updating with the scan. Port 248 gives the x-coord, and port 504 (248+256) gives the y-coord or the current scan line, and this can be used to synchronise graphics etc, as described earlier. Port 248 controls the CLUT when written to - as explained before. Bits 0,1,2 and 4,5,6 control the levels of red green and blue - so two levels of each are possible. Bit 3 adds luminosity to the colour when set. The first three RGB bits give 8 possible colours. In combination with the second three RGB bits, this gives 8x8=64 colours. Each of these can be "dim"or "bright", so 64x2=128 colours which is what you've got.
Reading Port 255 gives the currently displayed attribute which I don't really see the point of (there isn't really one - just the video hardware) and when written to controls the sound chip. Port 511 (255+256) must first be written to with the register you want to alter, and then Port 255 with the data. Eg, OUT 511,28: OUT 255,1 to enable the sound chip.
Ports 224 to 231 control disk 1, and ports 240 to 247 disk 2 if fitted. Details of how to use these are in the technical manual. To completely destroy the inserted disk do something like the thing below, which will certainly make some worrying noises if nothing else.

```
10 OUT (224+RND (7)), RND (255)
20 GO TO 10
```

Ports 232 to 239 control the printer which I'm not sure about. The technical manual contains details of all of these along with examples of how to use them.
The Spectrum uses the same ports, and this is why the Spectrum emulators can work just by using a copy of the Spectrum ROM.

## INPUT

is one of the simplest ways of extracting numbers or strings from the user. Its simplest form is: INPUT x where "x" can be the name of any variable. If "x" is a numeric variable, the Coupe will stick the cursor used in BASIC in the bottom left hand corner of the screen. A number can then be typed in, and when RETURN is pressed, the variable "x" takes the entered value. As well as simple numbers, expressions can be entered, which the computer will evaluate before assigning their result to "x": Eg, you could type in "2*3+5" or something, which would make the variable "x" = 11. If "x" is a string variable, then a couple of string quotes ("") will appear and a string of any length (within memory of course) can be entered.

```
10 PRINT "What's your name, eh?"
20 INPUT name$
30 PRINT "I have always despised the name ";name$
40 INPUT "And how old is ";name$;", hmmm?"
50 INPUT age
60 PRINT "It must be awful to be ";age;", ";name$
```

So this small program demonstrates how to use INPUT. The inputted strings or numbers are printed in the colours etc of the the LOWER SCREEN - which is the bit of the screen where you type in BASIC commands and so on. So CSIZE selections etc also have effect on INPUT, as it is very closely related to PRINT, except for the obvious difference that it extracts some information. It is possible to INPUT a "null" string by just pressing RETURN, but you are not allowed to do this with numbers.

```
10 INPUT "Tell me your name ";name$
20 INPUT "How old are you ";age
```

To make things more friendly, INPUT can print messages before the cursor - the first part of the program on the other page could be replaced by the lines below, which print the prompt in the lower screen, and the cursor pops up after them.

Several items can be input in one go, for instance: INPUT a$,b$,a,b,c$(4).

In this example, two strings then two numbers would be input, then a third string, which would replace the fourth string of the array c$. Because each variable in this example is separated by a comma, the different inputs would be separated by a TAB - ie, 8 or 16 columns depending on the value of a SVAR. In Mode 4,(or 1 or 2), two will fit across the screen; then the lower screen scrolls up and the next input occurs at the left margin...and so on... If the commas were replaced by semi-colons then the inputs would appear right next to each other. You can also use any other separators used by the PRINT command.



If you don't like the string quotes which appear when INPUTting a string variable you can use LINE - as in: INPUT "What's your cat's name? "; LINE cat$ . This will get rid of the string quotes. It also means you can't delete them and type in the NAME of a variable as you could otherwise. You can always do this with numeric input because there are no string quotes to delete - so you could type in "age" <RETURN> if the if the variable "age" had previously been set up.
If you want the INPUT message to include variables in it, you must use brackets. The program below would not work. It would ask for the user to input their name twice and then ask rudely "how tall are you?".

```
10 INPUT name$
20 INPUT "Tell me, ";name$;" how tall are you? ";h
```

Each item in an INPUT statement is separated by a comma/semi-colon etc, and if the first character of an item is a letter - not a " or a number - then the Coupe assumes that it the name of a variable that is to be input.

```
10 INPUT name$
20 INPUT ("Tell me, ";name$;" how tall are you? ");h
```

Either of the two line 20's below is the correct way to deal with it: brackets tell the Coupe that the 20 INPUT "Tell me, ";(name$);" how tall are you? ";h following variable already exists and is not to be input. Either the whole message can be bracketted or just the variables that you want to be printed.
Just as with PRINT, you can use AT with INPUT, but if you want to use the whole screen you must select stream #2: Eg,

```
INPUT #2; AT 10,10; "Why? "; LINE a$
```

Which will print the message at co-ords (10,10) (y,x) and input "A$" afterwards. When you do this, the message & input are NOT cleared when RETURN is pressed as occurs with normal INPUT which uses stream #0.
Another way to use the whole screen would simply be to redefine the lower screen using SVARS 60 - 64 and then do a normal INPUT, as in the following example:

```
10 POKE SVAR 60,31,10,10,12
20 INPUT "Got a hamster? (y/n) ";a$
30 IF a$="n" OR a$="N" THEN PRINT "Shame"
40 POKE SVAR 60,31,0,22,23
```

This would give the same result as the last example. It's longer, but the area would be cleared afterwards. Line 40 resets the lower window to its normal place.
You can change the co-ords and size of the lower window any time if you feel like it... There, got that out of the way.

## INSTR

is a function (short for IN-STRING if you're a bit dim) which searches one string for another. This can be useful in database type programs, for checking strings to see if they contain certain characters, for scanning memory or a whole host of other possible uses. If f$="I" have four gazelles, a woodlouse and a pig" and g$="woodlouse", then typing PRINT INSTR (f$,g$) would give the result 25. (I think). This is the postion within f$ that g$ (woodlouse) starts. You could just as easily type PRINT INSTR (f$,"woodlouse"), or type out F$ inside the bracket.

LET x=INSTR (f$,"wood") would assign 25 to x, again, but the "wood" could be part of another word - it could be "wood", but it could also be "woodlouse", "firewood", "woodpecker", "redwood" and so on. To check for the word "wood" on its own, you'd do LET x=INSTR (f$," wood "). Ie, put spaces on either side of the word in question.
Once you'd found one occurence of the target string, you might want to see if there are any more occurences. The Satanistic program below shows one way of finding all occurences of a string. When INSTR is used with a number before the first string ("n" here), this number is used for the start position within the string

```
10 LET a$="Well hello, I'm Bjorn from Norway, and I live
   in a place called Hell. I have a collection of sea-
   shells, but boy, it's absolute hell trying to find
   them here."
20 LET n=1
30 LET a=INSTR(n,a$,"hell")
40 IF NOT a THEN GO TO 80
```

```
50 PRINT a$(a TO)
60 LET n=a+1
70 GO TO 30
80 STOP
```

If no number is specified then it's taken to be 1 - the first character; the start. If the target string is NOT found, then INSTR returns 0, which is what line 40 checks for. Otherwise that part of a$ onwards is printed and the start position altered, and the search repeated. There are 3 occurences of "hell" in a$. Not 4 - because INSTR is case-specific and "Hell" is different from "hell".

## INT

stands for INTEGER, and rounds fractions down to whole numbers. When rounding fractions normally, you go to the next highest integer if the fraction is equal to orgreater to a half, and to the lower integer if it isn't. (So 4.4 would be 4 but 4.5 or 4.8 would be 5). However INT is slightly different in that 4.2, 4.5 and 4.8 all equal 4 when "INTed". And -4.2, -4.5 and -4.8 all equal -5 when INTed.
To get a "proper" INT, you can just add 0.5 to the fraction, as in the DEF FN below. Line 20 would print up 5, but the normal INT would have given 4 (4.6).

```
10 DEF FN round(x)=INT (x+.5)
20 PRINT FN round
```

Many commands, such as PEN, DRAW, PRINT AT etc will round fractions themselves, and others such as BEEP and obviously mathematical functions use fractions accordingly. (PAUSE 1.5 is actually slower than PAUSE 2 because 1.5 has to be rounded first!). Therefore whether you need to use INT depends on the command to be used.
To test for odd numbers - IF (n/2)=INT (n/2) THEN... will succeed if "n" is even.

## INVERSE

works in all modes and simply reverses the pen and paper colours - either permanently or temporarily. To see if INVERSE is selected do PRINT PEEK SVAR &5A4B which will give 255 for INVERSE 1. INVERSE can have different effects in different modes, and in different OVER modes, and with text/graphics, so I can't really explain it here. Eg: PRINT INVERSE 1; "Hello there" or INVERSE 1: CIRCLE 50,50,50

## ITEM

```
10 DATA "hamster","wildebeast","pigeon","harvestman"
20 PRINT "I am a flying squirrel"
30 DO UNTIL ITEM=0
40 READ b$
50 PRINT "No! I am a ";b$;", honest."
60 LOOP
```

The function ITEM gives information on the next DATA item and can be used, as in the example, to check whether there are any more items or not. The function ITEM returns 0 if there are no more items, 1 if the next item is a string and 2 if it is a numeric expression. If another DATA line was inserted at line 15 no more items would be printed, because ITEM gives 0 when there are no more items in the CURRENT DATA LINE. So it would ignore the DATA in line 15. ITEM can also be used in the same way in procedures when a list of data is passed to the procedure.

## KEYIN

This is something which is fairly simple to program (I would imagine) but has many thousands of uses. The Users Guide explains that KEYIN "enters a string as if you had typed it yourself" which isn't really very clear, because surely you would have to enter the string in the first place for KEYIN to enter it for you...
What it is actually trying to say is that KEYIN can be used within a program to make the computer think that the User has typed something in - and this can be used to add new program lines, or carry out certain command sequences etc etc.
Probably one of the main uses of KEYIN is to enter DATA statements, which is what the program below does. The string which



follows KEYIN can either be a variable (as in KEYIN a$) or an actual string (ie, KEYING "10 REM"). This string must be a valid BASIC command or line. Ie, it can be anything that can be typed into the normal BASIC editor without invoking the beep. If the string is invalid (ie, if you do KEYIN "hohkl" or KEYIN "BORDER s$") then you'll get "Not understood". If the string is valid, but causes an error, (like PLOT 3000,3000) then that error message will appear as normal, unless you have used ON ERROR.

```
10 FOR ad=65536 to 81920 STEP 10
20 LET b$=STR$ (ad-65536)+100
30 LET c$=b$+"DATA"
40 FOR n=ad to ad+9
50 LET a=PEEK n
60 LET c$=c$+STR$ a+","
70 NEXT a
75 LET c$=c$(TO LEN c$-1)
80 KEYIN c$
85 NEXT
90 STOP
```

So the DATA making program simply goes through memory, ad and makes up DATA statements of 10 numbers each. Line 75 chops off the last comma of the string, otherwise it would be something like "100 DATA 1,2,3,2,1,2,3,15,122,111," which is invalid.
KEYIN is quite a "slow" command and so this program would take a fair while to run. You should be careful when using it, as it is quite possible to delete parts of program, or make other mistakes which can have tragic consequences.
Another use (though rather naughty) of KEYIN is to get stubborn commands to run, particularly with the old ROM, but there's no point in dwelling on that.
Something else is to use KEYIN to "make" command which are normally impossible. For instance, when you use the BIN function, it MUST be followed by an actual binary number. You can't do "BIN VAL "10001001". With KEYIN you can. The two lines down on show how to do this. The string for KEYIN here is actually made from two strings - "PRINT BIN" and the binary number string. Any number of strings can be combined in this way. In this example, the binary number string is defined in line 10, but in practise you could generate it from various data.

```
10 LET a$="11010011"
20 KEYIN "PRINT BIN "+A$
30 GO TO 10
```

Another example: say you have a load of procedures, and you require the user to type in the name of the procedure to call: normally this would be impossible! KEYIN works!

```
10 INPUT LINE A$
20 KEYIN a$
```

This little program would suffice, although obviously you'd need to check that an actual procedure name had been typed. There are loads and loads of uses of KEYIN, I'm sure you can of lots more than these few. (I can't though...)

## LABEL

is rather confusingly explained in the manual (rather like everything else) and in fact I think I have already gone through it with GOTO and GOSUB. So read that.

## LEN, LENGTH

LEN stands for LENGTH (hey!) and will return the length of a string. So doing LET frank$="Yo!" and then doing PRINT LEN frank$ will print 3.
Or PRINT LEN "" gives 0 and PRINT LEN STR$ INT 34.33 does in fact give 2. If you object to using LEN you could always do...

```
PRINT DPEEK (LENGTH (0,A$)-2)
```

which has exactly the same effect. LENGTH (0,a$) gives the address in memory of the first character in A$. The two bytes before that are the high and low bytes of its length, and the bytes before that are the string's name.
If you have an original Spectrum manual you will probably be aware that the last few chapter contain all kinds of interesting bits

and pieces like this which allow you to spout out pointless little bits of info like that. I think I already went though this with you...
It's a very useful function which will either return the dimensions of numeric arrays or the addresses in memory of variables.
DIM x(15,36) will set up an array "x" of size 15 x 36. LENGTH will enable you to find out its size if you are stupid enough to forget, or if you find a variable which you have no idea as to its size. So PRINT LENGTH (1,x()) gives 15, and PRINT LENGTH (2,x()) gives 36.
There are brackets after the variable name to indicate that it is a numeric array. (I don't know why, because LENGTH won't work with normal numeric variables!). DIM A$(15,36) would return 15 after doing PRINT LENGTH (1,a$) and so on.
AS you know, an array can have any number of dimensions (you could do DIM a(3,2,4,2)) but LENGTH will only bother about the first two. Try LENGTH (3,a$) and you get an error, which is rather unfortunate.
In MASTERBASIC, the lengths of arrays can be altered, without destroying them, and it is in these cases that LENGTH becomes rather invaluable. See the MASTERBASIC manual for more details on that.
LENGTH can be replaced by several DPEEKS and PEEKS, in the same way as LEN can be, although this is rather pointless and unnecessarily long. However it would enable youto find the sizes of the 3rd etc dimensions.
The first number in brackets after LENGTH is either 1,2 or 0
0 is a special case, and will return the ADDRESS IN MEMORY of the variable. So if you reset the Coupe, and ran the program

```
10 LET a$="hello"
20 PRINT LENGTH (0,a$)
```

you would get a number somewhere in the region of 26000. As you defined more variables, each one's address would be higher than the last one.
This is very useful for storing machine code routines in strings, and there was a detailed discussion about this in ENCELADUS 5. Bear in mind that the addresses of strings do not stay the same all the time; they move about as other variables are made, etc, so LENGTH (0....) will not give a "permanent" value. You should use the command every time you need to act upon it, if you see what I mean. I don't think I need to go into much detail here...

## LET

Just in case you didn't already know, the Coupe allows "multiple LETs". So you can do something like LET a=2,b=5,A$="4" whereas on the Spectrum you would need a separate LET command for each.
Some BASIC don't insist on using LET (eg, the above example could be typed a a=2: b=5: A$="4"). I personally don't like this approach if only because using LET makes it clear what you actually mean. (Ie, "a=5" means that "a equals 5", which it doesn't, but "LET a=5" means what it does.)
You could use KEYIN to, eg, set up 26 variables a,b,c etc and to give them values. This program sets each to 0, but of course it could be much more complex.

```
10 FOR n=0 TO 25
20 LET A$="LET "+CHR$ (65+n)+"=0"
30 NEXT n
```

A brief bit about variables: Out of LET a=489322 and LET a$="489322", A$ will take up much less memory than a. Doing this with lots and lots of variables can give significant memory savings. To convert A$ back to a number, you'd simply use VAL A$. (as in PRINT 2+VAL A$) etc. MASTERBASIC has special commands to make this sort of thing easier.

## LIST

I should hope you all know what LIST does. It lists the program. LISTing a long program in mode 4 is surprisingly slow; much better to use MODE 1. Since most program lines are not all that long, a faster way is to use MODE 3, but to set up a window so that only half the screen is used. Even better, LIST in a window only a couple of characters wide!
SAM's list is different from the Spectrum LIST in that program lines are reserved 6 characters of each line, with long lines being indented. This makes programs much easier to read.
In addition, LIST FORMAT 1 and LIST FORMAT 2 can be used to indent lines even further. In these cases, colons separating statements disappear, and each statement is given its own line. The only difference between LF1 and LF2 is in the number of columns that certain commands indent lines.
FOR-LOOP, DO-LOOP and DEFPROC-ENDPROC all indent the section of the program that they occupy, so that the beginning and ends of loops can be easily identified - although I must admit that it's never actually been of much use...



IF, ON ERROR and ON also ident their own lines and ELSE and EXIT IF will "undent" a loop's indentation for their own line. (Ok?)
LIST followed by a number will list the program from that line onwards.
LIST followed by two numbers separated by "TO" (eg, LIST 20 TO 100) will list the program section between the two numbers.
LIST n TO n will therefore print up just one program line. To find the end of the program, do something like LIST 65000, which will more than likely produce a blank screen. Pressing the UP cursor should take you to the last line of the program. (The Spectrum 128K had all sorts of strange key combinations to do things like this automatically).
If a listing scrolls endlessly without stopping, it is because SCROLL CLEAR has been selected. Cancel this with SCROLL RESTORE.
Listings can contains all sorts of control codes which produce various effects when LIST is used. Most obvious are colour controls, which can either highlight certain lines, or hide them, by setting their ink to the same colour as their paper. Also, REMS can contain AT codes. (You can make these by doing something like KEYIN "10 REM "+CHR$ 22+ CHR$ 10+CHR$ 10). When these lines are listed, the listing will suddenly jump to whatever AT position was chosen, which would confuse the lister no end. There are loads of other things like this, and ways of making programs difficult to list, but this is not the time or place for such discussions. Eg, the line numbers can be POKED so that all line numbers are the same! It would then be almost impossible to edit them...

## *LLIST*

is exactly the same as LIST except that it lists to the printer. (The "L", by the way, stands for "line printer"). Listing programs to printers may cause some problems with line lengths, double spacings and the like. There are several SVARS which help. SVAR 14 contains the right-hand text column for printers. (normally 79). SVAR 15 contains the character code sent to the printer after CHR$ 13. Poke with 0, if you get double spaced lines. You may also have to fiddle about with your printer margins, I don't know... It took me ages to get satisfactory output, and in fact I still don't know what I did! In fact both LIST and LLIST can be followed by a stream number, so an equivalent to LLIST is simply LIST #3. You could open stream 4 to binary output (do OPEN #4,"b") and then LIST #4 if you wanted. (LLIST prints to channel "P"). It is easy to LIST to a string! Just do something like

```
RECORD TO G$: LIST #16: PRINT G$
```

(Stream 16 is the stream to which things go when they are "recorded"). Unfortunately it takes ages and some keywords appear as blobs, but you can't have everything. There are loads of other things similar to this that you can do.
You could cope without LIST, because the SAM (and Spectrum) produce listings when you press RETURN, which of course scroll with the cursor keys.

## LN

is a function to calculate natural logarithms. Without going into much maths... The natural logarithm of a number is the power that you must raise "e" to in order to get back the number you first thought of... ("e" is an infinite non-recurring decimal). The function EXP is used to do the opposite - EXP a gives e ^ a.

Common logarithms use base 10 rather than base e.
So the common logarithm of 1000 is 3, because 10 ^ 3=1000.
Similarly, the common logarithm of 500 is around 2.7, because 10 ^ 2.7 sort of =500.
Natural logarithms are the same except that they calculate to base e.

It's easy to calculate logarithms to any base. Simply divide the natural log of the number by the natural log of the base.
Eg, we know that 2 ^ 8 = 256; say you wanted to see what to raise 2 to to get 2048... PRINT (LN 2048/LN 2) gives 11, and you can check that 2 ^ 11=2048.

## LOAD.....

I don't want to dwell on this because there's plenty of information in the disk drive book and all over the place. But I'd better go though all the variations....
I'll assume we're using disk.....

LOAD "name" loads in the program called "name". Using tape, LOAD "" would load the first program on the tape, but obviously this does not apply when using disk. If the program is loaded, any previous program and variables are cleared. The program will NOT be loaded if its length is found to be too long (if RAMTOP is too low or if it is a 512K only BASIC program etc). In such cases do something like OPEN TO 10: CLEAR 163840 which should free up enough memory.
To stop a program auto-running, you can use MERGE instead of LOAD, or you can more simply do LOAD "name" LINE 64999. Since there is unlikely to be a line at line 64999, the program will not run. (the LINE tells the Coupe where to run the program from).

Of course, to make programs more difficult to break into, just make sure that line 64999 (or whatever the highest line number possible is) exists!
LOAD "name" SCREEN$ will load in a screen to the relevant memory position, and deal with setting palettes, screen mode etc. If the file is CODE, but not a SCREEN$, it will still be loaded, but will probably not be anything useful.
With MasterBASIC, compressed screens are still classed as SCREEN$ in the directory. Screens compressed with ENCELADUS compressors appear as CODE files, and should not be loaded using the SCREEN$ command.
LOAD "name" CODE will load in a code file to the address it was saved from. Of course it is possible to override this start address by doing LOAD "name" CODE start where start is this address.
You may want to load in a screen without altering the palette. You could do LOAD "name" CODE 507904 or LOAD "name" CODE 245760 (depending on whether you have a 512K or 256K SAM) which would only load in the screen data.
LOAD "name" DATA A$() or LOAD "name" DATA a() will load in a variable or array. I don't fully understand what you can and can't do here, so won't go into it!
Basically you can save a specific variable (say a PUT block, hamster$) and save it with SAVE "hammy" DATA hamster$, and then load it back into a variable with a different name. Eg, LOAD "hammy" DATA B$. What I'm not sure of is when you're meant to use brackets - I think if you are saving or loading an array.
Saving a BASIC program will also save any variables present, and these will be loaded back with the program. Therefore you only use the above to save certain, specific variables.
To save just a load of variables, simply delete any program, and then SAVE as if you WERE saving a program.
I'm sure you all know about doing LOAD "d3: blah" to load from different devices, etc etc, so I'll leave that for now!

## LOCAL

I have already explained LOCAL, LOOP, and LPRINT in one place or another.

## MEM$

is a bloody marvellous function which will assign huge hunks of memory to strings. This makes it easy to store, eg, graphics in strings, and then poke them back into place. The program below puts the entire screen into a string. Line 10 finds the start address of the screen.

```
10 LET sc=(1+IN 252 BAND 31)*16384
20 LET pic$=MEM$ (sc TO sc+24576)
30 CLS: PAUSE
40 POKE sc, pic$ MEM$(n TO m)
```

Line 20 uses MEM$ to bung it in a string. Line 30 clears the screen and waits for a key, and line 40 puts the picture back. The numbers in the brackets are simply the start and end addresses of the section of memory to store. They can be anything, from 0 up to 540000 or thereabouts, although you should note that you can't really store low addresses using MEM$. (Ie, the ROM). For some reason it won't work, and you'll just get a load of 0's. (Which can actually be quite useful). Of course MEM$ can also be used with INSTR to search areas of memory.

PRINT INSTR(MEM$(65536 TO 80000),CHR$ 255) would search for the first 255 byte in the area of memory from 65536 to 80000.

MASTERBASIC has a LOCN function which does the same thing, faster.

## MERGE

will sort of, "merge" in programs together. This can be used to combine several programs together. If the different programs each have totally different line numbers (like, part 1 has line numbers 0-100, part 2 from 500 to1000 etc) then there is no problem. If, however, there is conflict between line numbers, the program being merged in takes priority, and the old lines are lost. To avoid this, you can always renumber one of the programs to very high line numbers, and then renumber it all afterwards. MASTERBASIC has a special MERGE * command which is quicker.
Because MERGE stops progams running, it can used to stop and examine programs, althogh as I mentioned previously it's quicker to use LOAD "" LINE 64999.
MERGE can also be used to stop auto running CODE files, but they can be specially protected to avoid this - so it won't always work. Another use of MERGE is to merge in groups of variables.



## MOD

takes one number "modulo" a second number. PRINT 20 MOD 15 gives 5, because that is the remainder when 20 is divided by 15. It also works with fractions, of course, and negative numbers.
Especially useful for dealing with machine code type things, and also for ensuring that numbers don't exceed certain values but allowing "wrap-round". Eg, PLOT 257 MOD 256,0 would plot a point at 1,0.

## MODE

fairly obviously, changes the screen mode. MODE n, where n can be from 1 to 4. You know what each mode is like.
When the MODE command is used, several things happen: not only is the mode changed, but the screen is cleared, and several SVARS are altered to take into account the new mode - so that text and graphics commands work properly.
I've gone about this before: SVAR 64 holds the current screen mode minus 1 - so that's a quick way to find out what mode you're in - and so poking this can change the screen mode without clearing the screen, although the actual physical mode will not change. Ie, the graphics and text commands will start acting as though another mode HAD been selected, but they will not give proper results, because each mode has the screen laid out differently. Thus each mode has its own text and graphics routines. There are lots of SVARS holding paper and ink colours etc, I won't go into that!
There's a table at &5CA0 which is 16 bytes long and holds the mode and page number of each BASIC screen. By POKING it, you can effectively open screens, or alter the modesof screens without clearing them.
Bits 5&6 of PORT 252 hold the mode of the currently displayed screen. Use OUT 252,n to alter it, if you know what you're doing! (How the MODESWITCHERS work).

## NEW

on the SAM is actually a lot less harsh on the Spectrum, and I still haven't decided (after 2 years) whether I like it or not!!!
NEW on the SAM really only clears the program, variables and screens. On the Spectrum, virtually everything was cleared, including the char.set, system variables, sound - the lot!
Things like the interrupt-driven modeswitchers will survive NEW, and this may give some odd results which could worry people considerably - of course resetting the Coupe is MUCH more drastic than typing NEW.
Actually, when you NEW, although you lose any program that was present, the program ISN'T ACTUALLY ERASED. It's still there! It's just that the SVARS which control the length of the program, etc, are set to zero, and the first part of it wiped. Therefore, as long as you don't mind losing the first 100 bytes or so of your program, it ought to be somehow possible to restore old programs, but as of yet I haven't worked out how to....any ideas??
Some things are lost when you NEW, and some aren't - so streams are closed, but key definitions are kept, as are things such as SCROLL CLEAR and BLOCK selections.

## I covered **NOT** when I did AND (in great detail) back...

## And **NEXT** is mentioned with FOR.

## ON

is a useful little command that I never use.
Essentially it lets you choose from a selection of statements, and executes one of these statements depending on the value of a variable. (In the Users Guide it goes on about using it to GOTO or GOSUB, but I don't really see its point...)
The program down there is a very simple example. It asks for a number, and then makes a sound effect based on this number. In line 20, the POW is the second actual statement in the line, but when using ON, statements are counted from the ON. So the POW is statement #1, ZAP is statement #2 and so on. ZOOM is statement #4.

```
10 INPUT a
20 ON a: POW: ZAP: BOOM: ZOOM
```

Very simply, the statement corresponding to the value of a is carried out, and then the next line is considered. So if a=3, a resounding "BOOM!" will echo out around the room and wobble the furniture. And if a=1, a deafening "POW!!!" will be emitted. If the value of a does not correspond to any of the statements (like if it's 0 or more than 4), then none of the statements will be executed.

There is no way you can get each statement to do more than one thing, because a statement's a statement!! So the only way to do something complex is to make each statement in the ON command a GOTO, GOSUB or procedure call.
These could all be very useful. Another possible use is that of a menu program - the variable "x" could be the number pressed on the menu screen, then:

```
ON x: LOAD "prog": LOAD "game": LOAD "crap": LOAD"hamster":
LOAD "thing" etc etc.
```

You can't use string variables with ON, but you could look at the CODE of the first character or something. You should be able to replace the name of a variable after the ON with an expression, but I wouldn't risk it if I were you as it seems to be somewhat unpredictable.

## ON ERROR

This is a very useful command which stops errors stopping programs! Not so long ago I maintained that if you needed ON ERROR you were just a hopeless programmer, since it allows you to ignores any errors. However the main use is to TRAP errors rather than prevent them - ie, still act on them, but in a way not so disruptive to the actual program.
ON ERROR is followed by a command, and this single command is executed whenever an error occurs.
At it simplest, you can have ON ERROR NEW at the start of a program, which will NEW the computer when an error occurs. (ESCaping or pressing the BREAK button is also counted as an error and so would result in a NEW. To get round this in other peoples' programs: just keep pressing the BREAK button quickly)
However this is very crude and of little use. Much more purposeful to run a routine when an error is experienced. To do this, follow the ON ERROR command with a GOSUB or GOTO. I'll explain the difference in a moment...

```
10 ON ERROR GOSUB 50
20 PLOT RND (500),RND (500)
30 GOTO 20
40
50 RETURN
```

These two programs plot points randomly, but will plot some that don't lie on the screen - this would normally invoke an "Integer out of range" error. Line 10, of the first program, though, has stated that line 50 should be "gosubbed" if an error occurs, and this is what happens. There could be a long routine at line 50, but for this example, there isn't. There is simply a RETURN. Since the GOSUB effectively occured at line 20, the RETURN jumps to the statement after the plot. There isn't one, so line 30 is executed.

```
10 ON ERROR GOTO 50
20 PLOT RND(500),RND(500)
30 GOTO 20
40
50 GOTO 10
```

The second program says that when an error occurs, line 50 should be jumped to. Now. When RETURN was used in the first example (this would also happen if you used a procedure), ON ERROR was re-activated by the RETURN. (It is switched off for the error handling routine). However this does not happen with the second program, and so the original ON ERROR command has to be re-executed. In a real program, you can have lots of ON ERROR's throughout the program, just to make sure that it is not disabled without your knowing.
If you replaced the RETURN with CONTINUE in the first program, the statement which caused the error (the PLOT) would be attempted again (maybe the error handler alters something which makes the error line valid), although ON ERROR would still be disabled.

```
10 ON ERROR GOTO 50
20 INPUT f$
30 LOAD f$
40 GOTO 2000
50 PRINT "Disk error!"
60 PAUSE
70 GOTO 10
```



One of the most common uses for ON ERROR is to trap disk operations. Demonstrated by this program. Any error caused by the DOS (file not found, no disk, sector error etc) will cause a jump to line 50, which informs the user, waits for a key, and then tries again. There are lots of variations of this. (Eg, the error routine could go back to another part of the program rather than try loading the file again, or could offer the user the choice, etc).
You should make sure you know what you're doing with ON ERROR! Whenever an error occurs, three special variables are created. (They're special because they overwrite any other variables of the same name). These are ERROR, LINO and STAT, and contrary to what I just said, are only produced when an ON ERROR trapped error occurs. They can then be used by the error handling routine accordingly. In the example below, a file is loaded from disk.

```
10 ON ERROR GOTO 50
20 LOAD F$
30 GOTO 2000
40
50 IF error=87 THEN GOTO 10
60 GOTO 10000
```

If no disk is inserted, error 87 is generated. Line 50, the first line of the error handler, checks for this, by looking at the magic variable "error". If it sees that this is what happens, it tries again to load the file. If some other error occured (such as FILE NOT FOUND etc), then it gives up, and jumps to another part of the program.
LINO and STAT hold the line number and statement within that line where the error was caused. Thus the error handler can deal with several errors from different lines and sort them out accordingly.
I think I've mentioned most of this in various other commands...

## STREAMS, Open Streams

There are 16 streams, and each can be open to one of 5 (I think...6..or more maybe) channels, each of which does "something different". These channels each have an initial, to represent them, and they are:

S: Output to the normal, upper screen.
K: Output to the lower screen (the editting line)
P: Output to the printer, with tokens etc expanded and so on
B: Binary output to the printer - what you send is what the printer gets
$: Output to a string - as in RECORD TO .

There are also channels to do with the disk drive, which I won't go into.
I'm not sure about using the Network facilities of the Coupe (does anyone?!)
As I said, each of the 16 streams can be open to any of these channels, and then reading from or writing to these streams will have an effect on whatever channel it is opened to.
Some of the streams are always opened. Stream #16 is always open to channel '$'. Streams #0 and #1 and always open to channel "K", and stream #2 is always open to channel "S". Stream #3 is open to channel "P" when you switch on the Coupe, but is often changed. The others are initially closed.

But you can open them with the OPEN command! (Hurrah!)

OPEN #4;"B" would for instance open stream 4 to the 'B' channel. This channel can be used to send data direct to the printer, bypassing the ROM printer routines. Then to use it, you could do PRINT #4;"HAMSTER" or indeed LPRINT #4;"CEREBULLUM" or LIST #4; or DIR #4 etc.
INPUTing from streams is a little less useful; you can INPUT on the upper screen by doing INPUT #2;A$ or something.
Quite often programs contain lines like...PRINT #0;"I wonder what time is it", or PRINT #1; AT 0,0; INK 4;"It's five past five" etc. Since I said that streams #0 and #1 are always open to channel "K", these print commands will operate on the bottom two lines of the screen (the editting area).
These streams can be altered though... normal screen output (PRINT, LIST etc etc) goes to stream#2, open to channel "S". Before you can alter a stream you must close it - do CLOSE #2. Then do OPEN #2;"B", and all text output will go to the printer. Alternatively, CLOSE #3: OPEN #3;"K", and all LPRINTs and LLISTs will go to the editting area! etc etc.You couldn't do that on the Spectrum.
Redirection to channel "$" is also possible, for listings and DIRs to strings. I think I've already covered OPEN. (Can't remember where! but I'm sure I have).

### *OPEN SCREEN*

The Users Guide contains a double entry for OPEN SCREEN; how intriguing. Actually I think I've done this as well! (In DISPLAY or something).
Anyway, OPEN SCREEN simply opens up a new display screen. You can have a total of 16 on a 512K machine, although this depends on how much memory is avaiable. There are a couple of tables in with the SVARS, as well as the important PAGE ALLOCATION TABLE at &5100.
When you open a screen, these tables are altered. When you then do a DISPLAY n or a SCREEN n, to select a screen, these tables are checked, and if they contain certain values, then display is switched to the new screen. Otherwise you get the error "Invalid screen number." When you close a screen, the tables are again altered.
Briefly, the table at &5100 is 32 bytes long, and it contains a byte for every memorypage. This byte is: 0 for an unused page, 255 for a non-existant page. (as on a 256K Coupe - pages 16-31 don't exist). 64 for a page used by BASIC, 192 for a screen page, 32 for a "utilities page" and 96 for a DOS page. So the Coupe can keep track of what bits of memory are used for what, and this enables it to prevent screens, the DOS, etc, from being overwritten, which is pretty rather helpful.
Since a screen occupies two 16K pages, each page is 32768 bytes below the last. When you switch on the Coupe, the screen occupies the two pages near the end of memory; the DOS when loaded takes up the two underneath the screen. So SCREEN 2 will be situated 65536 bytes below SCREEN 1 rather than 32768 bytes, because the DOS is in the way. After that, SCREEN 3 would be 32768 bytes below SCREEN 2 and so on. Screens work their way down memory (ie, screen2 is below screen1), and so obviously the more screens you OPEN, the less memory you have available for BASIC. Before OPENing a screen, CLOSE it first! The reason for this - you can close a closedscreen, but you can't open an opened screen.

### OR

is dealt anywhere in the first parts. (I think - probably...)

### OUT

is dealt with IN, where there is a detailed discussion of ports...

### OVER

is rather difficult to explain, which is one reason why I didn't bother with it last time. It's effect varies depending on which mode you are in, what colours you are using, and so on. The early ROMS seemed to have a bug which meant that OVER did not operate correctly in MODE 3, and in fact I'm still wary of doing so.
OVER is a command similar to PAPER, PEN and INVERSE; it controls the way that graphics and text are placed on the screen. I will assume we are dealing in MODE 4. In this mode, OVER has more use with text.
Normally, if you PRINT in a character square which already has something in, the printed character will overwrite what was there before. (Ie, if you do PRINT AT 10,10;" " on a picture, a little block of paper will appear in the middle of the picture. If, however, you do PRINT AT 10,10; OVER 1;" " then nothing will appear to happen. (You could just as well do OVER 1: PRINT AT 10,10;" "). When OVER is in force, graphics are COMBINED with what's already on the screen, rather than OBLITERATING them, which is the usual effect. For instance, you can make a print "SAM Coupe" in a phoenetically correct sense by doing this:

```
PRINT "saam coope";CHR$ 8; OVER 1;"'"
```

The CHR$ 8 acts as a special MAGIC character which backspaces when printed, so that the apostrophe is printed OVER the "e" of "Coupe". Hence a little accent. So far, this all happens in each mode. However, in MODE 4, if you change colours between selecting OVER, then all sorts of things will start to happen.
What actually happens when OVER is "on", is that each pixel of colour is XORed with each pixel on the screen, which means that each bit within a pixel of colour is only retained if the bit of the existing pixel is of the reverse status....

```
PIXEL ON SCREEN: 0110 (colour 6)
PIXEL PLOTTED:   1100 (colour 12)
PIXEL PRODUCED:  1010 (colour 10)
```

The example above makes this a little clearer: each pixel consists of 4 bits, which gives a possibility of 16 colours. (as you know). (See explanation of BINARY). In the PRODUCED PIXEL, bits of the two pixels to be combined which match (ie, 1+1 or 0+0) result



in a 0 in the corresponding bit of the produced pixel. You can test that by printing something in mode 4, pen 6, and then the same thing in pen 12, OVER 1. You get the same thing in pen 10.
If you print different things in OVER 1, then you will get a mess - the various pixels will combine as explained. The next example shows why when you print something in the same colour, in OVER 1, you get something in the same colour, in OVER 1, you get a blank space - because EACH pixel matches, so the resulting pixel is all 0's - colour 0.

```
PIXEL ON SCREEN:  0101 (colour 5)
PIXEL PLOTTED  :  0101 (colour 5)
PIXEL PRODUCED :  0000 (nothing!)
```

In MODE 1, what ink you have selected makes no difference, because the pattern and colour data are stored separately. You get the same effects that you do when you use matching inks in MODE 4.
That deals with printing - although you really should experiment a bit...

### *Over with grafics*

With graphics, exactly the same applies, except that this time. Because the second time you draw something in OVER 1, the original data disappears, OVER 1 can be used to superimpose things over graphics and them remove them - as shown by the example.

```
ON SCREEN: 0101 | ON SCREEN: 0011
PLOTTED  : 0110 | PLOTTED  : 0110
PRODUCED : 0011 | PRODUCED : 0101
```

The ON SCREEN pixel for the second column comes from the PRODUCED pixel of the first column. Plotting the same thing as the first time restores the original pixel.

```
 10  LOAD "hamster" SCREEN$
 20 LET x=40, y=40
 30 LET x1=50, y1=30
 40 DO UNTIL INKEY$=" "
 50 OVER 1
 60 FOR n=0 TO 1
 70 PLOT x,y: DRAW 0,y1
 80 DRAW x1,0: DRAW 0,-y1
 90 DRAW -x1,0
100 NEXT n
110 OVER 0
120 LOOP
```

This program loads in a great picture and then flashes a box over it - the purpose of which being to select a bit of it, for some reason, but I didn't put in all the lines to deal with that...
You can do all kinds of things like this, or just use OVER 1 to generate some strange patterns.
PUT also responds to OVER, and behaves in the same way - so you can PUT blocks down on the screen without obliterating the surrounding area.
The SAM Coupe also allows OVER 2 and OVER 3. As explained, OVER 1 gives XORing - bits are set if bits do not match.
OVER 2 gives ORing - bits are set if either bit is set.
OVER 3 gives ANDing - bits are set if both bits set. See table.
(I included an INVERSE result to fill up space).

```
PIXEL    | 1100 | 0101 | 1111 | 1010 | 0000 | 0001
XOR 1010| 0110 | 1111 | 0101 | 0000 | 1010 | 1011
OR  1010| 1110 | 1111 | 1111 | 1010 | 1010 | 1011
AND 1010| 1000 | 0000 | 1010 | 1010 | 0000 | 0000
INVERSE | 0011 | 1010 | 0000 | 0101 | 1111 | 1110
```

These all have various effects. I think that's enough; why not try some out???!
(Just finishing OVER): You can find out which OVER has been selected by looking at PEEK &5a4c. PRINT PEEK &5a4c will give 0,1,2 or 3. PEEK &5a4a contains the OVER status for printing, which seems to be a bit "different".

## PALETTE

is probably one of the most "well-known" SAM command, and it has quite a large number of variations.
I won't go on about "paint pots" etc, since you should know all that - but very briefly, the Coupe can use any 16 out of 128 colours on the screen at once. The PALETTE command selects which one of the 128 colours you want to assign to one of the 16 "pots". (Sorry!)
When you switch on, some values have already been assigned (good job too - otherwise everything would be black and you wouldn't be able to see anything!). These have been chosen to give normal and "bright" versions of black, blue, red, magenta, green, cyan, yellow and white. (don't ask about bright black..) Just like the Spectrum.

PALETTE pot,colour
Where pot is the number (0-15) of the pot you want to change, and colour (0-128) is one of the 128 colours to put in the pot. Say you hate blue, because you're a demented socialist, so you decide to replace it with a nice bright red - you know that colour number 42 is red, and "pot" number 1 is usually blue. PALETTE 1,42 would do the job.
To find out what each of the 128 colours looks like (so you know that 42 is red in the first place), you can either look at the chart thing in the Users Guide or type in the little program it gives you to display all 128 colours at once. After a while you get to learn them all anyway, and they are quite ordered. You may think (as I used to) that if you displayed all 128 colours in order, you would get a nice "spectrum" effect, but due to the way they're organised, you don't. Originally, the SAM only had 64 colours, but the design was changed (quite late on) to double this. (More details elsewhere in CGTSB - see IN or OUT).
In MODE 1, things are a little confused because as well as having pots 0-15, you can also select BRIGHT, which will select another pot (sort of) - see BRIGHT. You can also have magic paint pots which flash (probably radioactive); and you can set up these by doing something like PALETTE 7,34,127. This will make colour from pot 7 (which is the colour used upon switch-on) flash between colours 34 (red) and 127 (white). The speed of the flashing is controlled by SVAR 8. Eg, do POKE SVAR 8,1 to give yourself an epileptic fit. POKE SVAR 8,255 to give the slowest possible change. So you'll be typing away, and the colour will CHANGE after you've forgotten all about it; give yourself a heart attack as well.
Now the SAM Coupe allows you to have more magic pots of paint (sorry, non-toxic wax crayons) which change colour at certain vertical positions on the screen. These are set up like this:

```
PALETTE 7,64 LINE 80
```

This would make ink from pot 7 change colour at line 80, which is about half the way up the screen. In the top half of the screen, then, it would be...whatever you'd selected before, and in the bottom half it would be colour 64. (green).Colours can also flash from specific lines - eg, PALETTE 0,34,99 LINE 80.
In CSIZE 8,8 the LINE can range from -16 to 174. Not 175 as you might expect; the Coupe can't change colour there because what's the point of changing on the very first line displayed?
The advantage of using palette "lines" like this is that it becomes possible to "draw" across the border, something which is normally impossible. Ok, so you can't actually put pictures there, but on the Spectrum, you had to resort to very fiddly machine code routines to do anything with the border.
Also, palette lines can be used to act as simplistic backgrounds for programs, which can not be "damaged" by graphics moving across them. They do, however, require extra processor time, so the program can be slowed down if you have lots of palette lines.
Note - my term "palette line" doesn't really mean much; it's my own phrase.

### *Palette lines*

PALETTE lines will survive CLS, but not CLS #. CLS # includes the command PALETTE, and when you type PALETTE on its own, all colours are restored to their default values. Type this if you ever get into trouble and can't see what you're doing. Tap the BREAK button first to clear the editting line though.
The colours for the "pots" are stored in memory starting at &55d8. 16 bytes from there correspond to the 16 pots. Then there are four bytes (for mode 3 colours I think). Next (at &55d8+20) is a second set of 16 colours, which refer to the "flashing" partner of each colour. In actual fact, all colours are always flashing: it's just that the flashing partner of each colour is usually set to be the same as the normal colour, so you don't see any flash. When you set up a flashing colour, only one of the 16 sets is altered. (Incidentally - the FLASH command for MODES 1 and 2 is entirely separate). At (&55d8+40) there is either a 255, which is there if you have no palette lines. If you do, then each palette line occupies four bytes:

1) line where the change takes place, but with 0 at the top (spooky!).
2) Colour (0-15) to change.
3) Colour (0-127)to change to.



4) Colour (0-127) to flash between.

There can be up to 128 of these, and a 255 marks the end. You don't really need to know any of that!
If you want to store a palette, you can either save it to disk, or store it in a string.
DISK: Type SAVE "palette" CODE &55d8,50 - 50 bytes is usually long enough, but if you have palette lines, then you will need to increase it, otherwise they won't load back. Up to 600!!

## PAPER

is used to select which paint pot (not again!) is to be used as the background for PRINTING. (not graphics).

PAPER c
Where c is between 0 and 15. PRINT PAPER 15; INK 0;"My hamster Keith" will print in black ink on white paper. (with a normal palette). That's a temporary paper (it only lasts for that PRINT), but permanent papers can also be chosen simply by typing PAPER 7 or whatever.
PRINT PEEK &5a48 will show you what paper has been selected, but the result will not be between 0 and 15, but between 0 and 255. Say you get 51 - in hex (do PRINT HEX$ 51), this is 33. The two digits match; they're both 3, and this is the pot being used for the paper. You can POKE this system variable with a number where the nibbles (halves of a hex number) do NOT match - eg, POKE &5a48, &30. In this case, the paper will be striped!
PEEK &5a30 gives the PAPER colour of the lower screen. (editting area).
In MODE 1, PEEK &5a45 can be treated like an ATTR value. (see there).

## PAUSE

is a wonderful command which turns the little blue feet of your Coupe into real paws and makes it scuttle away. Ha! Ha!
In actual fact, what it DOES do it that is makes the poor Coupe freeze, to death. This is useful for either waiting a set amount of time before doing something, or for slowing a program down.

PAUSE n
n can be any positive number at all, and represents a number of frames to wait for. Every second, the screen displays 50 frames (60 in N America). Therefore, PAUSE 50 will wait for 1 second, PAUSE 400 will wait for 8 second and so on. Likewise, PAUSE 25 will wait for half a second, and PAUSE 1 will wait for a fiftieth of a second. (When the SAM "waits", it does nothing except carry out key scans, palette changes, etc, for the specified number of frames, OR until any key is pressed. On a TV, the picture quality usually changes slightly during a PAUSE - usually improves).
PAUSE 0 is a special case which will wait for ever and a day. (But can still be interrupted by pressing a key). On the Coupe, the 0 can be left off, and the command PAUSE on its own will do the same thing. Handy, eh? You could also do GET k, which also waits for a key to be pressed.
I said that the value following PAUSE can be anything, but in fact it does only go upto 65535. And I have a feeling that PAUSE 0 is not "eternal", because I have often left a program containing a PAUSE 0 in it and then gone away; when I've recurred, the program has mysteriously moved on, past the PAUSE 0.....
How long does PAUSE 65535 last? 65535 frames will take up 65535/50 seconds, which equals 1310.7 seconds, which is 1310.7/60 = just under 22 minutes. This is the same length of time taken before the SAM turns off the display (if this is enabled).
PAUSE 1 is also a rather special case, because it simply waits until the start of the next frame, and so can be used to "synchronise" graphics with the TV frame. If you have an animation, or graphic display, which flickers a lot (in particular, it may "disappear" from time to time - this only applies to small graphics, especially those created with DRAW or CIRCLE), then try inserting a PAUSE 1 somewhere in the loop; this often slows things down very slightly to give a more acceptable display. If you want to wait for a time regardless of keypressing, use something like FOR n=1 TO 500: NEXT n. You have to figure out the numbers yourself...

```
10 BORDER 1: PAUSE 1
20 BORDER 2: PAUSE 1: GOTO 10
```

Something like on the Spectrum gives good border effects. Not quite on the SAM though, because the timing is different. (quicker).

## POKE, PEEK

I do not know if I need to explain these, since I have done so with other commands.
Basically, the SAM's memory contains 557000 addresses or so. (on a 512K SAM). An ADDRESS is simply a "box" which can hold a

number between 0 and 255. The 557000 comes from 32 "pages" of 16384 bytes each, plus 2 ROM pages of 16384 bytes. Put another way, each address can accomodate 1 byte, so 1000 addresses will let you store 1000 bytes.
The 2 ROM pages contain the data needed by the SAM to operate BASIC, load disks, etc etc etc. The rest is all RAM, which can hold whatever you want it to. (You cannot overwrite the ROM). However, there has to be somewhere for the screen, and somewhere for the DOS, but I'm not going to go into all that here. (If you are interested, see IN and OUT).
PEEK is a function which reveals the contents of an address, and POKE will put a value into an address. Eg, POKE 40000,52: PRINT PEEK 40000 will give you 52, because you just put it in! Addresses around 24000 are all system variables - I keep referring to them - which are USED by the ROM, but can be written over. I think I've covered other associated commands/info elsewhere...
Be careful when using POKE, as you can crash the Coupe. (But can't cause "damage").

## PEN

has been done (I think). It is used in the same way as PAPER and refers to the foreground used with text or graphics. Do PEN n, where n is between 0 and 15.
INVERSE 1 will reverse pen and paper. INVERSE 0 will reverse them back again.

## PI

is another numerical constant, equal to 3.141592653589..... it goes on for ever and ever. I'm sure are all familar with it. If you divide the circumference of a circle (perimeter) by its diameter (width), you always get PI, no matter what the size of the circle. Therefore it can be used to do various things with circles, such as find their areas, and so on, and is also used in various other mathematical situations. PI is actually written as a little thing like a lower case "n" with the top bar expanded, but it's impractical to use this on the computer; the word PI is used instead. PI on the Coupe gives the first 7 decimal places of PI.

## PLOT and DRAW

There are 49152 pixels on the screen - 49152 little dots, each of which can be any one of 16 colours. (Selected from a palette of 128 of course...). In MODE 3 there are double the number - 98304, because the resolution is higher.
The command PLOT is followed by two numbers, which specify the x-coordinate and y-coordinate at which to set a pixel to the current ink colour.
In MODEs 1,2 and 4, x can be between 0 and 255; and y between -16 and 175. (In CSIZE 8,8). In mode 3, the x resolution increases such that x can be between 0 and 511.
PLOT x,y will not only set the specified pixel to the current colour (or temporary - do PLOT INK 5; 50,34 etc) but will also change the current graphic position.

DRAW also does this, and it means that you can draw a line right in the middle of the screen, rather than one jutting out from the bottom left hand corner. A lot of other computers (BBC, C64 etc etc) use a different set of commands for drawing lines and so on, but you don't need to worry about that, do you!

```
10 PLOT RND (255),RND (175)
20 GOTO 10
```

This tiny program will plot pixels at random to give a fascinating picture. (There doesn't seem to be much more for me to say...)
The current x and y co-ordinates are stored in system variables, at addresses &5a42 and &5a41. You can do PRINT PEEK &5a41 to get the y co-ord, but must subtract it from 175, because the SAM actually likes to think of y co-ord 0 as being at the top. Do PRINT DPEEK &5a42, because the result can be over 255. While I'm at it, and so that I don't have to do it later, I may as well explain.

## XOS, YOS, XRG and YRG (PLOT scaling variables)

These are something which the Spectrum did not have (though BetaBasic did), and are special variables which control the scale and origin of the screen. Special variables - don't actually use them as variables, because if you do LET xrg=20, you will be affecting the screen, although XRG will stay as 20.
Normally there are 256 dots across the screen in MODEs 1,2 and 4, or 512 dots in MODE 3. (Sorry, I mean PIXELS). Doing PRINT XRG will print 256 or 512, depending on which mode you are in, but you can change them: if you type LET XRG=512, in mode 4, then doing PLOT 255,0, will plot a point half way along the screen. Normally the dot would appear on the right-hand edge.



Doing LET XRG=512 gives you 512 pixels across the screen, and the same applies the y-axis. (Normally, YRG=192). If you RECORD a graphic to a string, you can BLITZ it back in different sizes by altering XRG and YRG. In mode 3, set XRG to 256 in order to draw mode 4 graphics at the same scale.
Usually, swapping between modes 3 and 4 causes XRG to be doubled or halved, but sometimes this seems to go wrong - there appears to be a bug - so you should always set XRG after changing mode. More often than not, things will be fine, but (in my experience certainly) things can be a little unpredictable.
There are some system variables containing XRG etc, but I can't find them. (helpful!) But you don't need them anyway.
XOS and YOS are more special variables, which control the origin of the x and y axes. Normally, XOS = 0, which means that PLOT 0,0 will plot a point on the left-handedge of the screen. If XRG = 256, and you do LET XOS=128, then the origin will move to the centre of the screen. PLOT 0,0 will now put a dot in the middle of the bottom of the screen, and to plot on the left-hand edge, you'd do PLOT -128,0. Exactly the same applies to YOS.
As with XOS, YOS is usually 0, but negative y-coords are still allowed. Things are complicated a little by the fact that the y origin is altered by CSIZE, although YOS and YRG will not change. PLOT 0,0 will plot a point on the left, and at the top of the editting area. The height of the editting area, in pixels, is 2 times the current height of characters, so in CSIZE 8,8 it is 16 pixels high. Since there are 192 pixels up the screen, and y co-ord 0 starts 16 pixels from the bottom, it follows that the very top pixel will have a co-ord of (191-16)=175. In CSIZE 8,9, the top pixel is (191-18) =173, and in CSIZE 8,16, it's (191-32)=159.
Doing PRINT PEEK &5a5c will print up the height in pixels of the editting area. Because of this change of top y-coord, commands using the y-axis - like PALETTE LINE, as well as DRAW etc, may need to be altered accordingly.
You shouldn't need to worry about this if you always select CSIZE 8,8 at the start of a program (which is what I do), in which case PLOT 0,175 will plot at the top of the screen, and PLOT 0,-16 will plot at the bottom.
It may seem strange starting 16 pixels up from the absolute bottom, but I assume it is to allow compatibility with Spectrum BASIC which was the same - except that you couldn't plot in the editting area at all.
Anyway, if you select CSIZE 8,8, and then do LET YOS=-16, then PLOT 0,0 WILL plot in the VERY bottom-left-hand-corner of the screen, and the very top will be accessed with PLOT x,191. This may or may not be more convenient...
Setting XRG and YRG to small values makes it easy to plot mathematical things - like one of those graph plotting calculators. (Good aren't they?) The bottom program plots a sine wave using normal XRG,YRG, XOS and YOS values, and you can see that it requires a lot of multiplying etc to scale the sine wave up on the screen.

```
10 LET xrg=2*PI, yrg=2*PI
20 LET xos=xrg/2, yos=yrg/2
30 FOR n=-PI+.01 TO PI-.01 STEP .05
40 PLOT n, SIN n
50 NEXT n
```

```
10 FOR n=-PI TO PI STEP .05
20 PLOT 128+n*40,88+SIN n*50: NEXT n
```

The top program sets XRG etc so that line 40 is very simple. This would be more advantagous in a longer program.

## *Point*

POINT goes nicely with PLOT and returns the colour of a pixel on the screen. Very simple. In mode 4, a pixel can be any one of 16 colours (but out of a palette of 128, REMEMBER!?), and typing PRINT POINT (x,y) will return a similarly scaled number betwixt 0 and 15, indicating the colour of the pixel at co-ordinates x,y.
In mode 3, a number between 0 and 4 will result, but in modes 1 and 2, only a "0" or "1" is possible. This is because, as explained elsewhere, the colour data in these modes is stored separately. To access this colour data, you have to use ATTR (see there) which gives a number between 0 and 255. POINT in MODES 1 and 2 gives 1 if the pixel is set to the current pen colour, and 0 if it's set to the paper colour. The program below will work in mode 1, and will print the absolute colour of a pixel at co-ords (x,y). A good programming exercise...

```
10 LET x=0,y=175
20 LET a=ATTR (x/8,(175-y)/8)
30 LET p=POINT (x,y)
40 LET i=a-8*(INT (a-8))
50 LET pa=INT (a/8)
60 PRINT (i*p)+(pa*NOT p)
```

Line 20 gets the colour of the character square that (x,y) lies in, and line 30 sees whether it's ink or paper there. Lines 40 and 50 extract the ink and paper values from the ATTR value, and line 60 prints either the ink or paper value depending on the value of p. Incidentally, you can guess what I uses to think PLOT and POINT meant.. (YEARS ago!)

## POP

When you execute a GOSUB, call a procedure or start a DO-LOOP, the line number which held the calling statement is placed on a STACK, so that the Coupe knows where to jump back when it reaches a RETURN, ENDPROC or LOOP.
This "stack" is simply a reserved area of memory, and since you can GOSUB from within a procedure, etc etc, it "grows". Every time you "cancel" a GOSUB/procedure/loop, the stack shrinks again as the last value is removed, and jumped to.
If you do too many GOSUBs etc, or keep forgetting to RETURN (you can use GO SUB instead of GO TO for a certain number of times) then the stack will run out of room and you get an error: "Basic stack full".

POP is a command which takes off the last value from the stack, and if you follow the POP with the name of a numeric variable, then it will be put into this variable. You can use POP to get the stack back to normal.
Eg, if you get the above error and can't really see where it came from, type this line in a suitable place.

```
9999 POP: GOTO 9999
```

You couldn't type in DO: POP: LOOP because the POP would cancel the loop! Alternatively, you can use POP to purposefully end a procedure /loop/ subroutine, if you want to jump somewhere OUT of it and avoiding returning to the calling statement. Admittedly it is rarely used, and it's more than likely you'll never have any use for it, but it's there all the same.

## PRINT

Usually PRINT operates on the main area of the screen, but you can print in the editting area with PRINT #0;"Welcome to Rwanda".
You all know what PRINT does (I hope); it prints numbers and characters on the screen and has two basic forms: PRINT n and PRINT S$. Ie, you can print numeric expressions, or strings. If a string to be printed is absolute (my terminology - if you actually type the string, rather than using a variable) - then the string must be in quotes. (Fairly obviously).
Although you probably take it for granted, PRINT will automatically scroll the screen when necessary and deal with windows, and so on, which really is quite clever. It can print anything you throw at it and much more as well!!
PRINT SEPARATORS: One PRINT command can print several strings/numbers, each is separated by a punctuation mark as follows:

| | |
|---|---|
| ; (semi-colon) | - print next item right next to the last. |
| , (comma) | - print next item at next TAB position. |
| ' (apostrophe) | - print next item on next line, at left margin |
| Totally unrelated | - look, a big "O": () !! good, eh?! ()()()()() |

Usually, the "comma" separator has the same effect as pressing TAB when editting BASIC; that is, it moves 16 characters to the right, and if that will take the print position off the screen, then onto the next line. If you do POKE &5a2F,1 then TAB will move 8 chars instead of 16. POKE &5a2f,0 to restore normal status. Things can get complex if you use a lot of print separators, so you should make sure you know what you're doing!
When a PRINT statement ends, the print position is set to the start of the next line,unless the PRINT statement is terminated with a ; (semi-colon), in which case the next PRINT statement will carry on where the previous one finished.
There are a couple of system variables which hold the current print position, but unfortunately I can't find them, which is very sad, because they're very useful!
You can also use AT and TAB with a PRINT statement, which are QUALIFIERS. AT can be used to print at a specific location - eg, PRINT AT 10,10;"Ahoy!" to print in the middle of the screen. TAB can alter the column to print at in the current row, and is followed by a number 0-31 in modes 1,2,4 or 0-63 in mode 3.
If you set up a WINDOW (see this command later) then the limits of AT and TAB will change to whatever the limits of the window are.
Of course PRINT statements can also contain PAPER, INK, INVERSE, OVER and FLASH statements - eg, PRINT AT 3,10;INK 5;PAPER 0;INVERSE 1;FLASH 1;OVER 1;BRIGHT 1;"Yo!" (Forgot BRIGHT!)
As explained earlier, there are CHR$ codes which have effects when printed. PRINT CHR$ 65 will print "A", because 65 is the code for "A", but PRINT CHR$ 22;CHR$ 10;CHR$ 5;"Spanner." has the same effect as PRINT AT 10,5. PRINT CHR$ 16;CHR$ 5;CHR$ 17;CHR$ 10;"Do you read the Daily Mirror?" has the same effect as PRINT INK 5; PAPER 10;"Are you a total disaster area?" CHR$



18-CHR$ 23 represent FLASH, BRIGHT, INVERSE, OVER, AT and TAB respectively, and should be followed by one or two more CHR$'s.
CHR$ 6 also acts as a "comma" TAB, and CHR$ 13 as a RETURN. (Next line). CHR$ 8,9,10,11 control the print position, allowing you to move it left, right, down and up respectively, and CHR$ 12 will delete left. These are also very useful and make a lot of things easy to do. (eh?)
In the first question string above, it would be desirable to put "Daily Mirror" in quotes, but to do this you must use a "double quote", which tells the Coupe that you really do want to print a quote rather than end the string. Eg, PRINT "Have you read ""The Collins Gem Dictionary""" will have the sought-after effect. The "triple-quote" acts as a double quote plus string terminator. If you want to print in machine code, there are a number of routines: (Yes, I know this is the CGTS BASIC, but the following is something which is very useful to know). Nevertheless, you may like to skip it.
The routine in the ROM at &10 will print the character in the A register to the current stream, and will respond to the control codes I talked about earlier. Another routine at &13 will print BC bytes starting at DE, to the current stream. To set a stream, first load the A register with the stream you require - &FD for lower screen, &FE for upper screen, &FC to output to a string. (RECORD TO..) &FB for binary output to printer, and &03 for ROM's printer output routine. Then CALL &112.
The Relion TEXT COMPRESSOR does not actually set a stream within the code, because if use one of the PRINT routines without setting a stream, the last stream used will be used again. If the uncompressor code starts at 30000 dec, then doing PRINT AT 2,0: CALL 30000 will work fine, and to uncompress to the printer, LPRINT: CALL 30000. This is more convenient than poking the code with numbers.
The routine starting at &115 will output the Ath message from a table at DE. (0 for the first message). Set bit 7 on the last char. of each message. Do not use control codes. Good, eh?

I have dealt severely with **PUT** along with **GRAB.**

## RND

is used to make random numbers, which, if you think about it, is terribly hard to do. Go on, write a BASIC random number generator without using RND!
PRINT RND will give a number between 0 and 1, but RND (x) will give a number between 0 and x; a whole number; this is generally more convenient and of more use. If you do PLOT INK RND(15); RND(255),RND(175) then you will see that the dots are very evenly spread out. HOWEVER, if you go away for a while and then come back, you should see that some diagonal stripes have formed. This is because RND is only pseudo-random. It does to some extent use the clock for part of the number generation - the no. of frames passed since you switched on the computer - but in a loop like the one above (sorry, I didn't tell you; put a DO and LOOP either side of the PLOT command) then the clock will have moved on a set amount between successive RNDs anyway, so the randomness will be reduced.
I won't go into how the random number is generated, (partly because I don't know), but I do know that it's a good deal more complex than the Spectrum's RND, which used a load of dividing by 65536 and multiplying by 75 and goodness knows what. The RND function is quite "slow", and it can be avoided - for instance, if you go through the ROM with PEEK, the bytes are fairly random - divide them by x. (RND (X).

## *RANDOMIZE*

RANDOMIZE sets a system variable (&5c76 if you must know) with a number to be used for the next RND. How pointless, you may think, when the whole idea is to be random anyway.
However, the main use for this (as far as I can see) is to "synchronise" random routines. For example, you may have a program where you wish to display a load of dots at random positions, but for some reason, you want these dots to be in the SAME places each time. You could set up lots and lots of DATA statements and read from them, but much quicker to simply use RND and do a RANDOMIZE first.
Say that you do RANDOMIZE 10 and the result is pleasing - then put a RANDOMIZE 10 at the beginning of the routine, and you'll get the same result every time. This program will draw a random shape, but I can tell you that because of line 10, it will be quite organised.

```
10 RANDOMIZE 33074
20 FOR n=1 TO 10
30 DRAW TO RND(255),RND(175)
40 NEXT n
```

Then delete the line 10 and keep repeating, and I can guarantee that you'll never get the same shape until you replace the RANDOMIZE 33074. (Or wait for about 66000 turns). The value after RANDOMIZE must be below 65536.
Incidentally in a lot of (Spectrum) programs, RANDOMIZE USR n is used a lot; the RANDOMIZE has got nothing to do with it. USR has to be preceded by a command, and RANDOMIZE "does" about the least - you don't KNOW it's done anything!

I have dealt with **READ** (with **DATA**) and **RECORD TO** (with **BLITZ).**

## REM

stands for REMark and can be used to put reminders (yep! REMinder) in programs. Just follow the REM with anything you like and the Coupe will ignore it all.

## RENUM

is a useful little command which will quickly renumber program lines. If you're anything like me, then you'll always be adding or eraseing lines, or trying to squeeze them in tight holes, or numbering lines in steps of 1 and then trying to insert lines. When program lines are in such a mess, then RENUM will go through and renumber them in steps of 10, making things much neater.
Of course if you're a structure freak then you will never need to use RENUM because you will have written the entire program out on cartridge paper
On the other hand you get people who type RENUM after entering every single line. Ridiculous, isn't it.

RENUM n TO m LINE l STEP s is the full syntax, and any of the paramters can be left out as long as they stay in that order. Eg, RENUM 10 TO 2000 STEP 20.

Not like procedures; you can omit any parameter. There APPEARS to be some kind of bug which means that complex RENUMs will not work. (Give "No room for line", even when there most definately is). I see little point in dwelling on this...
RENUM corrupts the screen, so do not use it if you have vital info. on the screen! (The good thing about being in this part of the glossary is that I've already dealt with every other command..)

**RESTORE** is in with **DATA. RETURN** is with **GO SUB.**

## ROLL and SCROLL

are lovely commands to roll parts of the screen. The only difference between them is that where ROLL will wrap the edge of the area being rolled back to the other side (so that no data is lost), SCROLL replaces it with paper. In the following, any of the commands can be used with SCROLL or ROLL. Both work only in modes 3 and 4.

**ROLL d** will roll the screen one pixel in direction d, where d:
1:left
2:up
3:right
4:down

**ROLL d,n** rolls the screen "n" pixels in direction d. Unless n=1, then odd values of n are rounded up. You'd have to do, eg, ROLL 3,6:ROLL 3 for ROLL 3,7.

**ROLL d,n,x,y,w,l** If you want to roll only a bit of the screen, then you must include x,y,w,l, which refer to the x, y-coords of the top left corner of the area. w and l are the width and length (height) in pixels. Again, w is rounded to an even value. (But "l" is not).

If you try to SCROLL the screen away by doing 255 x ROLL 3 then you may be disappointed at the speed - although it is rather inevitable. Instead you must use bigger rolls; something like

```
FOR n=1 TO 8: SCROLL 3,32: NEXT n
```

could be acceptable....
With small rolling areas, then rolling will be considerably faster; in fact you may have to insert PAUSE s to slow it down. Also pleasant (over the whole screen) are non-linear rolls (how about logarithmic rolls?!) which speed up - like:



```
10 FOR n=1 TO 22
20 SCROLL 3,n
30 NEXT n
```

The 22 is not quite right; you would have to have another, smaller SCROLL after the loop, or do something else to "n".
In order to find an exact value to replace the 22 (if you're interested), we call this value "x", and we want to move 256 pixels. So 256=x(x/2) which means that x=SQR (2*256) which gives 22.6. Not much use, that, because putting 22.6 into the program makes no difference.
In machine code, ROLL and SCROLL are dealt with by a routine at &14B. "A" register = 0 for SCROLL or 255 for ROLL. C = direction, B = no. of pixels to move. L & H = x and y co-ords of top left corner E = length in bytes, D = height in pixels. (0 at top of the screen).

You know what **RUN** does!!! Remember that it clears all variables as well...

## SAVE

I've covered most of SAVE with LOAD, since it's basically the same. (!) I mean that the syntax is the same. Use DEVICE to choose disk or tape, or ramdisk, then:

| | |
|---|---|
| **SAVE "name"** | - save a BASIC program. |
| **SAVE "name" LINE n** | - save a BASIC program to auto-run from line n upon loading. |
| **SAVE "name" CODE ad,len** | - save CODE from address "ad" of length "len" bytes |
| **SAVE "name" CODE ad,len,start** | - as above, but auto-execute from address "start". |
| **SAVE "name" SCREEN$** | - save the screen, with info. on mode and palette. |
| **SAVE "name" DATA A$** | - save the string "A$". (You cannot save a numeric variable!!!) |
| **SAVE "name" DATA A$** | - save the array "A$"; you can't have a string and array called the same thing, so the Coupe knows which you mean. |
| **SAVE "name" DATA A$()** | - saves a string or numeric array. |

The loading back of these types of files uses basically the same syntax. When you SAVE to disk, the directory is also altered to give info on various aspects of the things you saved, but you don't need to worry about that... There's a LOT of detail in the Technical Manual about directories etc. See also DEVICE and LOAD for some more information.

Have done **SCREEN** (in OPEN SCREEN, I think...)

## SCREEN$ (y,x)

works in any mode and returns the character which is displayed on the screen at co-ords (y,x). (Character co-ords: y=0-21, x=0-31).
Obviously this only works if there is a genuine printed character there - which was printed using the current font. If not, then the result is a null string. SCREEN$ will recognise user-defined graphics, but not block graphics...(don't quite see why). I haven't used this function for YEARS, so I don't really have much to say about it, except that it's really more useful in mode 1 (and on the Spectrum!) because there's no way you can use it with PUT blocks in mode 4, which you are more likely to be using in the first place. However you could use it to "read" things off the screen by building up a string with SCREEN$, across a row...though I can't think of many applications apart from maybe examining a disk directory.

**SCROLL** was covered together with **ROLL.**

## SCROLL CLEAR

is one of several commands present in SAM BASIC which really do nothing more than poke a few system variables but exist for the sake of user-friendliness. SCROLL CLEAR simply disables the "scroll?" prompt which appears when LIST or PRINT (or DIR etc) takes up more than one screen.
After SCROLL CLEAR has been typed, no prompt is given, so long listings etc will scroll themselves without giving you a chance to read them, but it looks impressive.

SCROLL RESTORE will switch on the prompt system again.

The same effect can be gained by simply POKING system variable SPROMPT at &5ABB, which is non-zero if scroll prompts are disabled.
So, eg, POKE &5abb,1 is the same as typing SCROLL CLEAR, and you can PEEK &5abb to see if a SCROLL CLEAR has already been executed. (Don't know quite why you'd want to)
Incidentally, why not try listing a long program in MODE 3 in a very small window (Do WINDOW 0,2,0,2 or something)...it really is quite fun.

## SGN

stands for "sign", or "signum", and returns the sign of an expression, giving 1 if the expression is positive, -1 if it's negative, and 0 if it equals 0.
This does have quite a few uses, although I can't quite think of any now... How about doing LET a=a*SGN a, which will make "a" positive even if it's negative?!

## SIN, COS, TAN, ACS, ATN, ASN

There are two ways of thinking about this functions; they come to the same thing, but look different at the outset. Think about a circle. (I can't draw diagrams here). Imagine constructing a triangle inside the circle, a right angled triangle, which has one point at the centre of the circle, one point somewhere on the edge of the circle,and the other point situated somewhere inside the circle so that a right angle is formed. Got it?
The angle at the centre of the circle formed by the two of the triangle's edges can also be thought of as a distance travelled round the edge of the circle, and the sides of the triangle given names in relation to this angle: "opposite", furthest away from the centre of the circle, "adjacent", the side which forms the right angle with the "opposite" side, and the "hypotenuse", which is the other side, that's opposite the right angle and is also the longest of the three sides.
The lengths of any of these sides can be found provided you know the angle at the centre and the length of any one of the sides.
The SIN of an angle is the ratio of the opposite and hypotenuse, and equals O/H. (Opposite divided by hypotenuse). Similarly, COS=A/H and TAN=O/A.
If you think of the length of the hypotenuse (which is also the radius of the circle) as being equal to 1, then the length of side O (opposite) is equal to the SIN of the angle. (I'm talking about SIN, COS etc rather than sine, cosine etc, the "proper" names, because they take less time to type).
The length of side A is equal to the COS of the angle.
The length of the hypotenuse equals the radius of the circle, or can be calculated using SIN. Ie, it equals the length of side O divided by the SIN of the angle.
TAN can be used in the same sort of way, but doesn't involve the hypotenuse. If you keep the hypotenuse the same, but alter the lengths of the other two sides of the triangle, then a circle will be traced, and this is how SIN and COS can be used to draw curves. Alteratively, increasing the angle at the centre of the circle will also trace a circle. As the angle increases from 0, the length of the Adjacent side decreases from maximum (=radius of the circle) to 0, and the length of the Opposite side increases from 0 to maximum (again, = radius of circle).

```
10 FOR n=0 TO 2*PI STEP .01
20 PLOT SIN n*50,COS n*50
30 NEXT n
```

The program uses these facts to draw a circle. "n" represents the angle and increases from 0 to 2*PI. This is because the SAM works in radians rather than degrees, and there are 2*PI radians in a circle. It would be perfectly possible to use degrees, (so line 10 would read FOR n=0 TO 360), in which case an extra line would be required, line 15: 15 LET n=n/180*PI. This converts degrees to radians, since the computer calculates sines, cosines and tangents in radians whatever.
The explanation I have given of sine etc was extremely poor and you probably knew all about it anyway. If you didn't, and you still don't understand, that's no great surprise. All you really need to know is how the program above works, because there are LOTS of things you can do like this; just messing about with various functions will draw all kinds of patterns.
That program would give an error, because the centre of the circle would be at the graphics origin, so most of the circle would lie of the screen. The program should be:

```
10 FOR n=0 TO 2*PI STEP .01
```



```
20 DRAW TO 128+SIN n*50,88+COS n*50
30 NEXT n
```

(DRAW TO just gives a solidrather than dotted circle). (Like the imaginative title above?)

ASN, ACS and ATN can be used as "opposites" of SIN, COS and TAN, and return an angle.The SIN of an angle = O/H, and so O=SIN a*H. (a=angle).

If you know what O/H is, and need to know what angle would give this value, then ASN can be used. Eg, ASN 0.5 gives 0.52, meaning that if the angle = 0.52, then O/H would give 0.5. Remember that the value of the angle (0.52) is in radians; to convert to degrees, this would be (0.52/PI*180) which equals 30. So in degrees, SIN 30=0.5, and ASN 0.5=30.

These are less likely to be useful for graphics, etc.

```
10 FOR n=0 TO 2*PI STEP .02
20 PLOT 128*SIN n*80,80
30 DRAW SIN n*30,COS n*40+SIN n*30
40 NEXT n
```

Here's another small program using SIN + COS to draw a solid ellipse type thing; as I have already said, there are in numerable "things" you can do with these functions.

SIN, COS etc are calculated by using a series of formulas, and the accuracy of the result is dependent upon how many times the formula is "repeated". Can't remember the basic formula, but you could probably write a BASIC version more accurate than SAM.

## SOUND

is the cop-out way of allowing the user access to the sound chip, but it does let you control anything you like with regard to said chip. I don't want to go into the detail of using this command - that would require an entire article, several of which have already been written by various people in various places. There are 32 sound chip "registers", each one of which controls a certain aspect of the sound. (Though not all 32 are used).
The SOUND command is followed by two numbers; the first determines which register you wish to write to, and the second is the data - a number between 0 and 255. So SOUND 3,15 means "send 15 to sound register 3", which in turn sets the volume of generator #3 to 15.
There are two I/O ports (can only be written to) which are written to by the SOUND commands; port 511 is where the address (the register you want) gets sent to, and port 255 is where the data is sent. You can therefore generate sound easily in machine code by writing a small program to read data from a huge table and send it to these ports. How you go about writing the table is another matter...
Unfortunately, I don't quite understand all of the various things you can do with the sound chip, but I can list some of the charts from the technical manual, since none of them are included in the Users Guide.
On second thoughts, I don't think I will copy out the tables. They're too complex. If you are serious about trying to use the sound chip, buy the technical manual.

The following registers are used to produce simple sound: (given in dec.)

28: sound chip enable/disable: do SOUND 28,1 to enable chip and SOUND 28,0 to silence

20: Frequency mixer: controls which generator (0-5) should be enabled. To use generators 0,1,2 and 3, do SOUND 20,(1+2+4+8). For one gen., do SOUND 20,10-5: Amplitude of generators 0-5. Each generator can have separate volumes for the left and right channel (hence giving a stereo effect), and each of these registers is split into two.

To get generator 0 at volume V, do: SOUND 0,V+16*V. Gen 1 would be SOUND 1,V+16*V Maximum volume=15. For volumes VL and VR (left+right) do SOUND 0,VL+16*VR. 8-13: Tone number for generators 0-5. Tone can range from 0-255, within the selected octave.

16,17,18: Octave numbers for generators 0-5. Each of these registers is split into 2, as with the amplitude controls. Octave ranges from 0 to 6.

Other registers control noise, envelopes and so on, but I won't go into that here. For one thing it would take too long, and besides, I don't "get" all of it...

## SQR

stands for "square root" and calculates the square root of a number, ie, finds a number which when multiplied by itself gives the original number. Ie, SQR 9=3.
Doing 9 ^ 0.5 is exactly the same, but is slightly slower.
To find the cube root of a number "n", do (n ^ (1/3)) and so on.

## STOP

is a crude command which stops (amazing) program running and exits to BASIC. As far as I know, it cannot be trapped by ON ERROR, although as it is an actual command which you'd type into a program, you wouldn't want to trap it.
When a program has STOPPED, typing CONTINUE will resume the program, from the statement or line after the one where the STOP was situated.
Some of the old Spectrum keypresses have been retained on SAM. One of these is STOP, which can be accessed by pressing SYMBOL+A. (Unless key definitions have been changed using KEY). Pressing these keys to produce STOP whilst in an INPUT will exit to BASIC, providing a means of breaking into some programs. If the key definitions have been changed in such a way that pressing SYMBOL+A does not produce STOP, then you're stuck, since simply typing STOP is no good.
This produces an error "STOP in input" rather than "STOP statement", proving that the two sequences are actually completely separate.

## STRING$

can be used to generate large sequential strings, but has a few limitations. PRINT STRING$ (100,"*") will print 100 asterisks, and PRINT STRING$ (100,"ha!") will print 100 "ha!"'s, (300 characters in all).
The first number, which represents the number of times to repeat the string, cannot exceed 255, which can be a bit of a drawback. To get 300 asterisks, you would need to do PRINT STRING$(100,"***") or whatever.
The final length of the generated string can, apparently, not exceed 512 characters, so you are still rather limited. If you needed a really huge string you could always do this:

```
LET a$=STRING$(255,"*"): FOR n=0 TO 10: LET a$=a$+a$: NEXT n
```

which would double the size of the string 10 times over...

## STR$, VAL and VAL$

I think I'll treat STR$, VAL and VAL$ all together.
Collectively, these commands convert strings to numbers and numbers to strings. Thus, LET A$=STR$ (4+5) would make A$="9". (Doing LET a$=4+5 is definately a no-no). This converting of numbers to strings makes it possible to check the length of numbers, and to see whether they contain certain numbers (how else would you find out whether 3784.337 has the number "5" in it? It hasn't...) and do all sorts of things that you can only do with strings.
VAL does the exact opposite and converts a string into a number. If the string in question does not make sense as a number (eg, if it's "hamster"), then the error "Variable not found" will be produced, since the computer will regard "hamster" to be a numeric variable. Of course, if there is such a variable, then fine, the corresponding number will be used.
If the string does not even make sense as a variable (eg,"2ifi7;sd)#2") then you'll get "Not understood", which, incidentally, is very crude compared to the elegant Spectrum corresponding complaint of "Nonsense in BASIC").
So, PRINT VAL "5+3" will of course give 8, and LET a=VAL "5+3" will make a=8. This makes it possible to "generate" numbers, or sums or expressions, since the string may contain functions and all sorts.
The two programs below both allow the user to type in the name of a numeric variable (such as "a" or "hamster"), and then display the contents of that variable.

```
10 INPUT "What variable do you want to see? ";LINE A$
20 PRINT "Variable ";A$;" has value ";VAL A$
```

```
10 INPUT "What variable d'yer want to regard? ";a
20 PRINT a
```

However, the first program, remembers the NAME of the variable as well as its actual value. The 2nd program really cheats.



VAL$ acts like VAL, except that it acts on strings and it's result is a string. One use for this is to set up a string so that it's contents can be interpreted as a string itself. Eg, LET A$="H$+B$+""hello""". Note that double quotes are needed.
A$ will then hold the string "H$+B$ etc", but doing PRINT VAL$ A$ will then evaluate that string, adding strings H$ to B$ to "hello". If either H$ or B$ didn't exist, then "Variable not found" would result.
If the VAL$ string does not make sense as a "stringular expression", then "Not understood" will appear.
The use of these three commands may not be apparent if this is the first time you have come across them, but problems quite often crop up in programs which can be solved easily by one of them. You should note, however, that VAL, and probably VAL$ as well, are prone to causing resets or crashes if you misuse them, and you should therefore check the best you can that they are only being used with valid strings. VAL, by the way, stands for "eVALuate"...
And don't go doing KEYIN "PRINT STR$ VAL "VAL$ "A$+STR$ SQR f""""" or that sort of thing as it just will not do..
SVAR can be interpreted as meaning "+23040", and was intended as an easy way of accessing system variables - ie, you could do POKE SVAR 8,2 rather than POKE 23048,2. To a certain extent it is used with common svars, but it's become a little bit meaningless.

## SVAR

Similarly, DVAR accesses the DOS variables, and is genuinely more useful, because the addresses of these variables can change, and the addresses involved are higher, so a bit of typing is saved...
Also useful are XVAR's, which only exist in MasterBASIC and access this program's special variables.

## TAB

is something I don't use a lot; it simply controls the horizontal (column) positioning of text on the screen, in the CURRENT row. The disadvantage is that you can't "go back" on yourself. Ie, while PRINT TAB 10;"Boris";TAB 20;"Pankin" will print the two words on the same line, doing PRINT TAB 20;"Pankin";TAB 10;"Boris" will split them over two lines.

## TRUNC$

will delete trailing spaces from strings, so that PRINT TRUNC$ "Mein Kugelschreiber ist kaput " will only print up to "kaput". This can be jolly useful if you have strings taken from a string array, where all the strings are of the same length and will therefore have superflous spaces in them.

## UDG

is also very useful, and will give the address in memory of the start of the 8 bytes of data which contain the pattern for characters. Since CHR$ 32 (" " - space) is the first character which can be altered, doing PRINT UDG " " will give you the start of the character data area, and it is normally 20880. The SAM differs from the Spectrum in that the normal character set is always in RAM. (On the Spectrum, in order to change it, you had to POKE all sorts of things). The character set is also stored in the ROM on SAM - otherwise you'd have blank characters when you switched on - but it is moved to address 20880. Typing PRINT DPEEK &5C36 will give an address 256 bytes below the address of the "space" pattern. (so it normally give 20624). This, in effect, would be where the pattern data for CHR$ 0 would be stored, which makes it easier for the PRINT routine etc to locate addresses. (Eg, CHR$ 65 would be 20624+65*8).
To change a character - say "A". Just do something like :

```
POKE UDG "A",0,43,255,44,2,2,132,88,0
```

The eight numbers represent the eight pixel lines of the character, and range from 0 to 255. You may find it easier to use binary: to define "A" as a solid triangle, do POKE UDG "A", BIN 10000000,BIN 11000000,BIN 11100000,BIN 11110000,BIN 11111000, BIN 11111100,BIN 11111110,BIN 111111111.

## **UNTIL** is described elsewhere.

## USR

The SAM has two main ways of calling machine code from BASIC: CALL and USR. CALL simply..."calls" the machine code in much the same way as GO SUB calls a BASIC subroutine.

USR is a function rather than a command, and has a result: this result is the value of the BC register pair (so ranges from 0-65535) when the machine code ends.
So, if you had a M/C program at address 30000 that went:

```
LD  BC, 52112
INC BC
RET
```

and you then did PRINT USR 30000, you would get 52113 printed up.

USR$ is a string variant, and its result is a string which comes from address DE, in page A, of length BC. If these registers are not set to the start of a string upon the machine code exit, then USR$ will try to print a nonsense, or infinately long string, so care must be taken.

I haven't seen a use of USR$ yet...

## VERIFY

I can't remember if I have covered VERIFY before or not...I think I have...

## WINDOW

is a very nice command which doesn't seem to be used as much as it could be.

WINDOW left,right,top,bottom
The syntax is shown above. Each of the four numbers refers to an edge of the window, given in character blocks. The size of the window therefore is dependent upon which mode you are in, and when you change mode, your window is destroyed. You can, however, set up windows on different screens.
What you can't do is set up more than one main windows, as you could on Beta BASIC. However, the lower window (where commands are typed) can be made any shape you like, and placed anywhere, and since it can be printed to with PRINT #0 or PRINT #1, it can act as a second window.
In MODE 3, CSIZE 6,8 you can do WINDOW 0,84,0,21, which is the size of the normal screen. In MODE 4,2,1, CSIZE 8,8 this is reduced to WINDOW 0,31,0,21. Using short letters (CSIZE 6,6) would give more rows, but WINDOW will not allow this;you have to poke the system variables directly.
Try setting up some window (do WINDOW 5,15,5,15) and listing, print, DIRing, etc - it's quite interesting, but beware that if the window is small, and you try to edit listing (pressing the up cursor in particular) the computer may crash.

```
&5a38 - upper window rhs column (31)
&5a39 - upper window lhc column (0)
&5a3a - upper window top row    (0)
&5a3b - upper window bottom row (21)
```

The system variables listed can be used to examine, and change, the size of the upper and lower window. In particular, the second set of svars, which control the lower window, can be very useful. Apart from making the lower window different sizes and altering its position, you can banish it!

```
&5a3c - lower window rhs column (31)
&5a3d - lower window lhs column (0)
&5a3e - lower window top row    (22)
&5a3f - lower window bottom row (23)
```

Poking its top row with a number greater than the lowest row in existence (eg, do POKE &5a3e,50) will usually make the lower window disappear. You can still type in it, but you won't see anything being typed.
By poking &5a38, you can set up "magic" lines which wrap round themselves! In mode 4, do something like POKE &5a38,100, and then LIST a long program. Lines which are longer than 31 chars long will wrap round on the same line, because the computer will think you're using a 100 column screen! Etc.

I've covered **XOS, YOS, XRG** and **YRG** in some other place.



## XMOUSE, YMOUSE

need no explanation, and XPEN and YPEN are a bit meaningless. Therefore I have completed the COMPLETE GUIDE TO SAM BASIC.

*PS: A mouse needs a driver to work succesfully, and a good driver has also an explanation how it works....*

The end of the guide may seem a strange place to say it, but it is quite likely that you don't have the start: this guide was not supposed to be a complete BASIC "tutor", but more a discussion of the specific commands and functions available to the SAM programmer, and a few details which you won't find in any of the manuals. I hope you have found it useful, inspiring and thought-provoking. At times we laughed, and yes, at times we cried. But together we got through this thing and now here we are, at the end, with everything behind us, full of wonder and excitement at the secrets that have been revealed, and bubbling with anticipation of what may be to come.
Well the thing is, there won't be anything else to come because I can't think of any more tutorials to write. So, if you have any ideas, let me know. And if you have any specific programming problems, come to me, and if there's enough response, I'll have a section where I can answer such queries. Until then, then, see ya! (how crud)