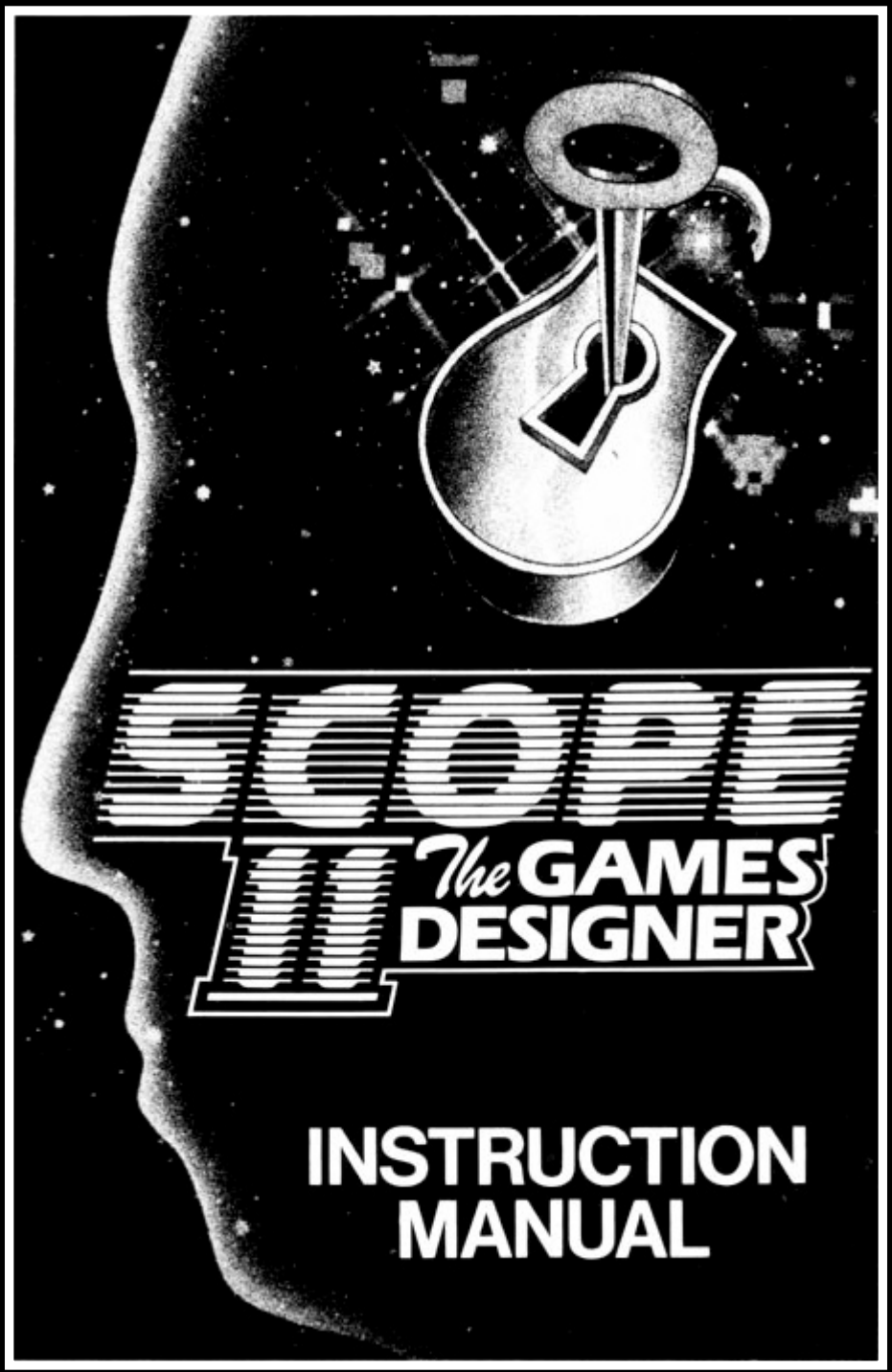

SCOPE
II
The GAMES DESIGNER
INSTRUCTION MANUAL

Program developed and written by
Allen E Pendle

Sprite Routines by P J Richards
Circle Routines by Mike Williams
'XPUT' by M Probert
Manual by ISP Marketing Limited

Introduction

SCOPE2 is a fully structurable medium level language primarily intended for the high speed handling of graphics, colour sound and animation as a fast to work with and easy to use alternative to machine code.

SCOPE2 is composed of 36 command words which are written in BASIC REM statements, before being compiled to a section of memory defined by the user for that purpose, and run by means of a USR command.

Although many SCOPE words only handle integers in the range 0 to 255, there are also several words which handle 16 bit integers in the range 0 to 65535.

The handling of colour and sound is somewhat different to the approach taken by BASIC, but the relevant sections within this manual will explain this fully to you.

As far as has been possible, SCOPE words have been made to resemble their BASIC counterparts so as to aid programmers, many of whom will already be familiar with BASIC, but not very familiar with the various other languages available for the Sinclair Spectrum.

However, with the visual similarity, so the comparison ends, as the SCOPE versions are translated into pure machine code which of course runs very much faster than BASIC.

The code generated by the SCOPE operating system is not dependent upon SCOPE being resident in memory in order to run, so any program written with SCOPE will run totally independent of SCOPE. You can if you wish write a game using SCOPE, and give a copy to a friend to play even if he does not own a copy of SCOPE.

One last word with regard to machine code, SCOPE was not written as a means of learning machine code, but because of it's more primaeval approach to programming (!) than BASIC, it does have certain similarities to machine code, and as such may be used if you so desire, as a means of progressing from BASIC to machine code, if only to realise a better understanding of how and why your computer works.

Loading instructions

Side A – SCOPE Program LOAD " "

Side B – Sprite Designer LOAD "Designer"
Demo Routine 1 LOAD "Shapes"
Demo Routine 2 LOAD "Mr ISP" CODE
Demo Routine 2a LOAD "Multi"
Demo Routine 3 LOAD "Landscape"
Demo Routine 4 LOAD "Titles"
Demo Routine 5 LOAD "Blip"

All demonstration routines with the exception of "Mr ISP", when loaded type RUN and ENTER or LIST to examine.

Syntax

The basic syntax format for SCOPE instructions is quite simple and follows the following rules:-

1) All instructions must be typed in BASIC REM statements.

2) All key words are immediately followed by a semi-colon ";"

3) All operands are separated by a comma ","

4) All instructions with the exception of "PUT" and "XPUT" and those which do not have any operands must be terminated in a colon ":"

5) All key words must be typed in lowercase letters "a" for example and not "A" the format shown in the dictionary should be followed exactly.

Any syntax error will cause the error report "SYNTAX ERROR" and the line number in which the syntax error occurs.

Whilst every care has been given to error checking, remember that if you ask SCOPE to do something silly, like jump to an undefined label, it will carry out your instructions and perhaps result in an undesirable effect, or crash. It is recommended that you save a copy of your program before even compiling it, just in case!

Memory Organisation

Since SCOPE translates the program you write into machine code, you must first nominate an area where this machine code will be written. The machine code written by SCOPE, will be in two parts, the first will be the main program, and the second part will be composed of any "routines" which are written.

The easiest way of imagining this segregation is to imagine that you are writing a program in BASIC, and every time you write a sub-routine to be called by a "GO SUB" instruction, that you write the sub-routine at line numbers above 5000, for example, and the main program at line numbers below 5000.
The area in which SCOPE will write it's routines must be nominated at the start of a SCOPE program as well as nominating where the main program will be written. The instruction to nominate these areas is "org;" which is an abbreviation for "organise" because it instructs the SCOPE translation program to "organise" the subsequently written machine code to the nominated areas of memory. The format for "org;" is as follows;

10 REM org;x,y:
Where "x" refers to the memory address at which the writing of the main program will commence, and "y" refers to the memory address where the writing of the routines will commence.

There is no restriction on whether y>x or x>y, so long as they are not the same, though please notice that if x and y are the same, then an error report will NOT be given, and your program will be translated and rewritten to the same address for both the program and it's routines, which, should your program contain any routines would have disastrous effects!
The only limitations on where you may nominate your program to be written are summarised in the Spectrum's memory map below:-

Address	*Useage*	
0-16383	ROM	(NOT AVAILABLE)
16384-22527	DISPLAY FILE	(NOT AVAILABLE)
22528-23295	ATTRIBUTES	(NOT AVAILABLE)
23296-23551	PRINTER BUFFER	
23552-23733	SYSTEM VARIABLES	(NOT AVAILABLE)
23734-23754	CHANNEL INFO	(NOT AVAILABLE)
23755-37999	BASIC RAM	
38000-39999	SCOPE PROG	(NOT AVAILABLE)
40000-59999	BASIC RAM	
60000-65467	SCOPE OP SYSTEM	(NOT AVAILABLE)
65368-65535	U.D.G. AREA	

As you can see, the best area to organise your SCOPE program to be written is between addresses 40000 and 59999, leaving the lower free area of BASIC for the BASIC program's REM statements.

It will be assumed henceforth that you choose to organise your SCOPE program to start at address 40000, and your SCOPE routine area 10000

bytes further on at 50000, thus leaving approximately 14K to write your SCOPE program in, and 20K for it to be translated to.

Thus your first line of program would read;

10 REM org;40000,50000:

Or whatever line number you choose to use.
Notice the semi-colon after the key word "org", the comma separating the two operands (40000 and 50000) and the colon terminating the line.
Please not the last line of any SCOPE program must always be exit.

e.g. 200 REM exit;

Compiling & running

Having written your SCOPE program into REM statements, you need to compile it into it's final super-fast machine code format.
This is done by simply entering as a direct command;

PRINT USR 60500 and press ENTER

After a short delay, which will vary according to the length of your program, either a number will appear at the top of the screen, or a SYNTAX ERROR message will be displayed at the bottom of the screen.
If the error message is given, it will also display the line number where the syntax error was discovered, and you will require to check and correct the line in question.
Having corrected the error, attempt compilation again by typing in the direct command as before.
Do not be surprised if another error report is given, as this error message will only report the first syntax error that is discovered, and there may be more throughout your program.
The most common causes of syntax errors are;

1) Spelling a key-word with a capital letter to begin.
2) Leaving the colon off the end of the instruction line. Remember, it is only PUT and XPUT which do not require a terminating colon.
3) Not putting sufficient operands into a PUT or XPUT instruction.

Having finally compiled your program, you will see a number at the top of the screen.
This number refers to the first free Byte from which any subsequent program will be written.
It is important to realise that this is the first free Byte, and not the last one used, as should you require to use SCOPE to write a series of machine code programs to be used as sub-routines from a BASIC program, then you will ORG; the next SCOPE program to start at the number shown at the top of the screen.
To actually RUN your compiled and super-fast machine code SCOPE program, type as a direct command or execute as a program line;

LET L = USR first org address

Saving

Having spent many enjoyable hours writing a program with SCOPE, you will obviously wish to save the compiled program to cassette.
This may be simply done using the Sinclair Spectrum's save bytes facility, thus in the following example it will be assumed that you have compiled your SCOPE program to the addresses 40000 and 50000.

1] The first step is to find out how much of memory your compiled program uses up. This will be the number printed to the screen if no routines have been written. Otherwise enter as a direct command PRINT PEEK 60484 +256★PEEK 60485.

2] The next step is to save your program by;

SAVE "name" CODE 38000, N

Where N is the address given by the PEEK command or the address printed to the screen, *Less* 38000. 38000 is where the SCOPE dedicated routines are stored and you must always save from this address.
Your compiled program will be then save to cassette, and may be reloaded, without SCOPE present in the computer, by;
LOAD "name" CODE

Handling Numbers

Often it is very useful to store a number, so that it's value may be changed in relation to it's existing value rather than to a new and definite value.
Such a system is known as a "variable", and within BASIC variables are used in almost all programs from the simplest to the most involved.
So it is with a program written in SCOPE.
However, SCOPE variables differ slightly from BASIC variables, the most significant difference being that SCOPE variables may only be "named" by one or two character names.
SCOPE also, however has two types of variable, these are "vars" and "bvars".
First, vars.
A var may hold any value between 0 and 255 inclusive, notice that negative numbers are not provided for.
The name of a var may be any letter of the alphabet in either upper-case or lower-case format, each being treated as individual, thus "a" is not the same as "A".

The word format for initialising a var is as follows;

var; name, a var or a number:

Thus for example;

```
20 REM var;a,10:
30 REM var;B,a:
```

Will set both var “a” and var “B” to hold the value 10.
A var may be increased in value by way of the word “inc” which follows the following format;

inc;var,a var or a number:

Where the second operand indicates the value for the first operand to be increased by.

For example;

40 REM inc;B,10:
50 REM inc;a,B:

Will increase the value of both the vars “a” and “B”, but notice that var “a” will be increased in value by 20, as opposed to the increase of 10 applied to var “B”.

As may be expected, a var may also be decreased in value, and this is supplied by the word “dec” which follows the following format;

“dec;var,a var or a number:”

Notice that any overflow of the values 0 to 255 will result in a “wrap around” value being put into var. Thus; a var holding a value of 255, and increased in value by 1 will resultantly hold a value of;

255 + 1 = 256 but an overflow occurs, and 256 is subtracted leaving 0

Likewise, if a var holding a value of 15 is increased in value by 250 then the resultant value will be;

15 + 250 = 265
265 – 256 = 9

This is known in mathematical circles as “modulo arithmetic”, where the vars operate in modulo 256.

As a further example, consider the folowing;

A var is assigned with the value 12, and then decreased in value by 18, the resultant value is;

12 – 18 = – – 6
– 6 + 256 = 250

If you suffer any difficulty with grasping this principle, draw yourself a circle, and number it from 0 at the top, around to 127 at the bottom and up to 255 at the top left, just before the 0.
Now, place a pointer to the value held by the var, for example the number 12, and move on the number of placed required, thus to subtract or decrease the value move anti-clockwise and to increase the value move clockwise.
Thus, with your pointer on 8 count around anti-clockwise 18 places, starting the count as “1” on the 7. Notice how you end up on the number 250!
Vars are one of the most versatile and useful features of SCOPE, and have uses ranging from loop counters to fluctuating co-ordinates and in reading the keyboard, but more about these features later!

We now move on to the big brother of the humble var, the "bvar" or "big-variable".

A bvar may hold any value between 0 and 65535 inclusive, and operates in a modulo 65536 form.
A bvar may be named by a letter of the alphabet, but it must be a CAPITAL.
The format for the word to initialise a bvar to a value is;
bvar;name,a number:

Notice that bvar;name;bvar: is NOT allowed.

Thus in the following example both bvar "A" and bvar "B" would be set to the value 123;

```
20 REM bvar;
30 REM var;B,123:
```

As with vars, a bvar may also be increased or decreased in value, the formats being;

add;bvar,a var or a number:

To increase the value of a bvar by a value not exceeding 255, and;

minus,bvar,a var or a number:

To decrease the value of a bvar by a value not exceeding 255.

Notice that as with vars, if the value held within a bvar overflows the range 0 to 65535 then a "wrap around" effect is experienced, only in this case 65536 is either added or subtracted as the case may be.

Bvars have the unique capacity for displaying their value on the screen, this is achieved by the word "num" which follows the following format;

num;a var or a number,bvar,a var or a number,a var or a number:

The first operand specifies the attribute value for the number held in the second operand to be displayed in at the line and column co-ordinates as defined by the last two operands respectively.
Thus;

```
20 REM bvar;A,12345:
30 REM num;56,A,10,10:
```

Will print the number "12345" at co-ordinates 10, 10 in black ink on white paper.

Finally we come to SCOPE's random number system. As anyone who writes games will tell you, random generated numbers are an essential element.
This word, "rnd" follows the following format;

rnd;a var,a number, a number:

It assigns to the specified var, a value which is a random value between 0 and 1 less than the first number, and to this is added the second number.

20 REM rnd;a,10,0:

will assign var "a" with a value which is between 0 and 9. This is equivalent to the BASIC statement;

"LET a = INT(RND★10) + 0"

Colour & Graphics

Throughout this manual, colours are referred to as a combination of the PAPER and INK colours, combined with the BRIGHT and FLASHing state of the characters in question, this colour being the "attribute" value, and is derived from the table below;

Ink	*Black*	*Blue*	*Red*	*Magenta*	*Green*	*Cyan*	*Yellow*	*White*
Paper								
Black	0	1	2	3	4	5	6	7
Blue	8	9	10	11	12	13	14	15
Red	16	17	18	19	20	21	22	23
Magenta	24	25	26	27	28	29	30	31
Green	32	33	34	35	36	37	38	39
Cyan	40	41	42	43	44	45	46	47
Yellow	48	49	50	51	52	53	54	55
White	56	57	58	59	60	61	62	63

Notice, that to add extra brightness to a displayed character, 64 is added to the attribute value, and for FLASH, 128 is added, thus for both BRIGHT and FLASH add 192 to the value in the table above.

In circumstances which require only a single colour number, such as the border colour, the value used in SCOPE programs is the same as the values used from within BASIC.

The two words with relevance to colour, are "bdr" and "chg".
"bdr", which is an abbreviation for "border" is used to change the colour of the border, just as "BORDER" is used from BASIC.
The format for the word "bdr" is as follows;

bdr;a number or var:

The value of the number or var may be any value up to 7
Thus, to change the border colour to red, you could use the following short program;

```
10 REM org;40000,50000:
20 REM bdr;2:
30 REM exit;
```

Or, alternatively, you could use a var, and then the program would like this;

```
10 REM org;40000,50000:
20 REM var;a,2:
30 REM bdr;a:
40 REM exit;
```

The word “chg”, which is an abbreviation for “change” changes the permanent attributes, and clears the screen so as to implement them. This means that it changes both the paper colour, and the ink colour, though the change in ink colour will not be noticed until your program is listed.
The format for the word “chg” is as follows;

chg;a number or var:

The value of the number or var should be taken from the table above. Thus, to change the screen colour to blue you could use the following program;

```
10 REM org;40000,50000:
20 REM chg;8:
30 REM exit;
```

Notice that the screen changes to blue, but the ink remains black.
If you change the value in line 20 to, for example 15, then the paper will become blue, and the ink will change to white.

We now move on to the Sinclair Spectrum’s excellent graphics.
Please notice, that SCOPE treats the Spectrum’s low resolution graphics and UDGs as though they were standard characters, and as such are not pertinent to this section.

The Sinclair Spectrum has the capacity for high resolution displays in the form of “PLOT”, “DRAW” and “CIRCLE”. SCOPE retains all these, and has even improved the “CIRCLE” command to give greater flexibility.
Notice though that “INVERSE” is not catered for directly, as colour combinations must always be specified.

First “PLOT”, the SCOPE word to illuminate a single pixel is also called “plot”, and follows the following format;

plot;colour,column,line:

Notice that the colour, line and column operands may all be either a number of a var, thus, to illuminate the pixel in the centre of the screen we could use the following;

```
10 REM org;40000,50000:
20 REM plot;56,128,88:
30 REM exit;
```

If your screen is already in white paper, with black ink, you will notice a tiny dot appear in the centre of the screen when you run the above program.
But, let us assume that you wish to plot the central pixel in “INVERSE”, SCOPE does not have an “INVERSE” command, but this is not difficult to immitate;

Change line 20 to read;

```
20 REM plot;7,128,88:
```

And now run it.

On it's own, "plot" is not a very useful command, however it is essential when using "DRAW", as it provides a reference point for the start of line drawing. Speaking of "DRAW", SCOPE's word to "draw" a line is also called "draw" and follows this format;
draw; colour, + or – left or right, + or – up or down.
Notice please, that all the operands may be either numbers or vars. The "+/– right or left" operand indicates firstly with the + or – sign whether the line is to be drawn to the right + or the left –, and secondly by how many pixels. The third operand, operates in a similar fashion, to draw up + or down – and by how many pixels. Notice that the direction operands must be prefixed by either a + or a – sign. Drawing of the line takes place starting at one of four possible positions;

1] The bottom left corner of the screen, co-ordinates (0,0) if no Pixel has been previously plotted.

2] From the co-ordinates of the last Pixel Plotted.

3] From the end of the last line draw using either "draw" or "globe"

4] From the bottom right corner of the last character which was "xput"ed to the screen.

You may assign a starting point for a line created using "draw" in one of two ways;

1] Plot the appropriate Pixel using "Plot".

2] Poke the co-ordinates of the starting point into the Sinclair Spectrum's system variable "co-ords", addresses 23677 (column) and 23678 (line).

Okay, time for an example of "draw". The first example will link the two points (0,0) and (30,20);

```
10 REM org;40000,50000:
20 REM draw;56,+30,+20:
30 REM exit;
```

This example will draw a line from the far left of the screen to the far right, across the centre of the screen;

```
10 REM org;40000,50000:
20 REM plot;56,0,0:
30 REM draw;56,+255,0:
40 REM exit;
```

A little more interesting is this, which quarters the screen;

```
10 REM org;40000,50000:
20 REM plot;56,128,175:
30 REM draw;56,0,–175:
40 REM plot;56,0,88:
50 REM draw;56,+255,0:
60 REM exit;
```

The drawing of arcs by use of a third draw parameter is not catered for by SCOPE.

Now to "globe", SCOPE's word for drawing circles!
"globe" follows the following format;

globe; column, line, radius:

Notice that once again any or all of the operands may be either a number or a var.
"globe" draws a circle, at three times the speed of the ROM's routine, around a central point indicated by the line and column operands, and of a radius specified by the radius operands.
Type in the following example, and you will get the idea;

```
10 REM org;40000,50000:
20 REM globe;128,88,50:
30 REM exit;
```

Okay, compile it and run it.
Great, a circle appears in the centre of the screen, no big deal.
However, now change line 20 to read thus;

```
20 REM globe;128,88,100:
```

Now try it.

The colour will always be the prevailing permanent colours as set by CHG;
There is one final word in this section, "over" which operates in the same manner as it's BASIC counterpart.
The format for "over" is as follows;

over;0 or 1:

If a 1 is present, then the "OVER" condition is set, where as the presence of a 0 resets the over condition.

To appreciate this try the following;

```
10 REM org;40000,50000:
20 REM over;1:
30 REM globe;128,88,50:
40 REM globe;128,88,50:
50 REM exit;
```

Notice how the circle is drawn and then undrawn?
This is because the "over" establishes the screen display to display all characters and to plot all pixels etc in an XOR manner.
For the benefit of those of you who are not familiar with logical operators, XORing does the following;

When a "1" or an "on" or "plotted" pixel meets a "0" or an "off" or a "non-plotted" pixel then a pixel is plotted.
However, when a "1" or an "on" or a "plotted" pixel meets another, then the original is set to "0" or turned "off" or "unplotted".

Printing Characters

SCOPE has two ways in which characters may be displayed on the screen, they are called “put” and “xput”.
From BASIC, when it is required to display a character upon the screen, the usual format is to “PRINT” the character, SCOPE’s equivalent to this is “put”. The format for “put” is as follows;

put;colour,line,column,test

Notice that “colour”, “line” and “column” may all be either a number or a var, and that NO COLON is required at the end of the line.
This absence of an end-of-line colon is only apparent with the two words, “put” and “xput”, and allows for a colon to be displayed on the screen if required. At first glance, the word “put” appears quite daunting, so here is it’s BASIC equivalent;

“PRINT;INK ?;PAPER ?;AT LINE,COLUMN;“TEXT””

Notice how SCOPE’s “put” includes both the paper and ink colours in a single value, this value is taken from the table of colours given in the section on “colour and graphics”
Thus, as an example, let us assume that you wish to display the word “SCOPE” at line 7 and column 3 of the screen in black ink, on a green paper back ground. To do this you would use the folowing example;

```
10 REM org;40000,50000:
20 REM put;32,7,3,SCOPE
30 REM exit;
```

Try changing the colour value in line 20, to ensure that you are familiar with the Sinclair Spectrum’s attributes system.

Now for a revolutionary new idea; “xput”.
In the previous section, on colour and graphics you noticed how any pixel on the screen may be illuminated using “plot”, and that there are 256 pixels across the screen, and 176 down.
But, you will notice that normally you may only display a character at one of 32 positions across the screen, by 22 down.
“xput” however, gives you the capacity to display a character commencing at any of the 256 horizontal pixels and 168 of the vertical pixels.
The character thus displayed is said to “commence” from the top left corner of the character.
In addition, “xput” also has the capacity to display characters in standard, double or half width!
The format for the word “xput” is as follows;

xput;line,column,colour,mode,text

Notice that the colour, line and column operands may all be either numbers or vars, and that the mode number is derived from this list;

0 Standard width
1 Double width
2 Half width

"xput" has some peculiar properties, which I hope to be able to illustrate with the demonstration programs to be found later, however, for the time being, try this:

```
10 REM org;40000,50000:
20 REM xput;175,1,56,1,SCOPE
30 REM put;56,1,1,SCOPE
40 REM exit;
```

You should be able to see quite clearly the individually assignable pixel offset feature of "xput" from this example.
It should be noted, that "xput" will also display UDGs as well as standard characters, and will work with redesigned character sets, but NOT with the low resolution graphic characters available from the keyboard.

As a further example try this;

```
10 REM org;40000,50000:
20 REM var;x,2:
30 REM var;y,10:
40 REM xput;x,y,56,0,SCOPE
50 REM exit;
```

It doesn't work does it! Instead, an "INTEGER OUT OF RANGE" error report is given by the Sinclair Spectrum.
But, if we change line 20 to read;

```
20 REM var;x,20:
```

And now try it, it does work alright.
Try this for some fun;

```
10 REM org;40000,50000:
20 REM xput;88,247,56,0,SCOPE
30 REM exit;
```

Notice how the screen "wraps around", so that the COPE is displayed to the left of the S at the far right of the screen.

Structure

As we have already discussed, a program written using SCOPE is formulated into two separate areas, the “program” and “routines”. These two areas may be distinguished between by the following generalisation; “A ‘routine’ is a collection of instructions designed to perform a particular task. The ‘program’ is a series of control statements which communicate with, and respond to the actions of ‘routines’.”

I will stress at this point, that this fundamental idea, which is the very foundation of SCOPE programming is very different to BASIC, and as such often difficult for BASIC programmers to grasp and come to terms with.
Machine code programmers, and programmers familiar with Forth, should not experience any difficulty with this structuring idea, but as I expect the vast majority of you are BASIC programmers, I will attempt to explain this structuring idea as simply as I can.
To illustrate this idea of structure I will use an imaginative company situation.
Let us imagine then a simple company which produces a product, it doesn’t matter what product, it can be what ever you wish!!

The company has the following departments;

1] Design

2] Manufacture

3] Marketing

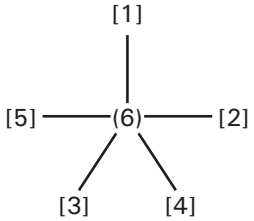
4] Accounts

5] Market Research

6] Management

As I said, a very simple situation

We can illustrate this company in a diagramatic form, thus;



The [ ] units indicate the various departments as per the number within the square brackets, and the central (6) indicates the managerial section of the company.

The lines indicate paths of communication, please remember that as this is a simple example, individual departments cannot communicate with each other.

Notice then how the running of the company is controlled by the managerial department, which requests action from the various departments, and processes the results of that action.
Thus;
For example let us presume that the managerial department decide that they require a new product.
First they call market research on the phone, to discover what sort of product will sell best.
Then they call design to whip up a prototype.
Next, accounts are called, to decide whether or not the company can afford the product.
Having decided that they can afford the product, a call goes out to the manufacturing department to start production, and a call goes through to marketing to start selling the new product.

So the structure of the company is such that a central control system (management) can call various sub-systems (accounts etc) so as to create a final result.

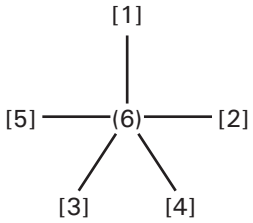
Thus with a SCOPE program a central control system (program) calls various sub-systems (routines) to produce a final result.

Thus a standard arcade style game is composed of a number of routines, and a central control program which calls the various routines, and interprets any effects and as a result calls other routines.
It soon becomes apparent that whilst this style of programming is fairly simple to use and easy to follow, it is very long winded, but it is fast.
It is best to become familiar with "structured" programming by writing short and simple programs before you even contemplate a large, complex and involved program such as an arcade game.

As an example then, we will create the basis for a very simple drawing program. The idea is that lines may be drawn in any of four standard compass directions by pressing the appropriate cursor keys. Similar to the "Etcher-Sketch" type of idea.

It is apparent then that we require four routines to draw lines, each routine will plot one pixel in the appropriate direction.
We also require a routine which will detect a key press.
And finally a central control program which will link together the routines.

Here then is the diagramatic view of the program's departments;



The departments are as follows;

[1] Detect key press

[2] Draw Up

[3] Draw Right

[4] Draw Down

[5] Draw Left

[6] Control

Simply, the control program calls “Detect key press”, and checks which key number is returned, calling departments [2] to [5] as appropriate.
Or if it is found that no key is being pressed, then the control program will loop around continuing to call the “Detect key press” routine until a valid key press is detected.

From this simple example, you should be able to work out your own structures for whatever purpose you require.
For fear of appearing a terrible bore, may I remind you please of that diabolical system “the flowchart”, which despite being tedious and boring is very often essential to good programming.

If more would-be-programmers used flowcharts and worked out their ideas on paper before even touching the keyboard, there would be more good programmers, and a lot less would-be-programmers.
It is all very well establishing a program composed of a central control system and numerous routines, but it is essential to have some means of naming them, and passing program control to a specified routine or what ever.
In BASIC, when it is required to pass program control to a routine or section of program, we use “GOTO” or “GOSUB” followed by the relevant line number. However, SCOPE does not use line numbers, and instead areas of program are named or more appropriately “labelled”.

The SCOPE word to “label” an area of program is “label” and follows the following format;

label;name:

The “name” may be any letter of the alphabet, in either upper or lower case and if required may be prefixed by a # symbol.
Thus, 104 labels may be used, and these are;

a to z
A to Z
#a to #z
#A to #Z

Thus a label instruction may look like this;

50 REM label;A:

or

50 REM label;#S:

Since the labelled name contains only one alphabetical character, it is not very descriptive of the function performed by the routine or section of program which it means.
For this reason SCOPE has a word called “note” which operates in the same way to a BASIC “REM” statement, that is the program ignores it and continues onto the next line in the program in sequential order.
The format for “note” is as follows;

note;text:

Thus, we might designate a routine or area of program thus;

50 REM label;S:
60 REM note;Increase and display score:

Now that we can name and reference routines and areas of program, it is neccessary to inform the SCOPE language system which parts of the program are routines and which are program.
Normally any instructions will be translated into machine code and rewritten to reside in the program area designated by the “org” statement which commenced the program, but there is an instruction which informs SCOPE to write a section of program as a routine into the routine area.
This word is “routine” and follows the following format;

routine;

Upon encountering the instruction “routine”, SCOPE writes all further instructions into the routine area designated by the earlier “org” statement. It only finishes writing the routine when it encounters an “end” instruction which follows the following format;

end;

The instruction “end” causes SCOPE to write a “return” instruction into the routine area, and to revert back to write further instructions to the program area.

Thus, a routine follows this sort of format;

100 REM routine;
110 REM label;A:
120 REM.....
.....
190 REM end;

With lines 120 to 180 holding the various instructions which compose the routine.

Remember that a routine, is similar to BASIC’s sub-routines which are called using “GOSUB” and returned from using “RETURN”.
To call a SCOPE routine you use the instruction “call” which calls the specified routine, and returns when it encounters the “return” instruction written by SCOPE into the rountine area.
The format for call is;

call;label:

Where “label” refers to the name of the required label.

If, however, you require to pass program control to a section of program or to an area within a routine, then the instruction “jump” is used. “jump” is SCOPE’s equivalent of the BASIC “GOTO” instruction, and follows the following format;

jump;label:

It is similar to “call” but is used in a direct “GOTO” type situation. As an example consider, but do not run the following endless loop;

```
10 REM org;40000,50000:
20 REM label;S:
30 REM note;START:
40 REM jump;S:
50 REM exit;
```

This is directly equivalent to the following BASIC program;

```
20 REM
30 REM
40 GOTO 20
```

A very boring, but illustrative continuous loop.
Now, as an example of why you must not jump to a routine, try the following;

```
10 REM org;40000,50000:
20 REM label;S:
30 REM jump;#S:
40 REM jump;S:
50 REM routine;
60 REM label;#S:
70 REM end;
80 REM exit;
```

On the face of it this is another continuous loop, as all we do is goto a routine, and after returning from the routine we goto it again and so on. However, notice that line 30 says jump and not call, thus when the program is run, control will pass to the routine, and upon encountering the return instruction at the end of the routine will return to BASIC.
Why does it? I hear you ask!
The reason for this rather peculiar series of events is as follows;

When a program makes a “call” or “GOSUB” to a routine, the address from which the call was made is stored. This is so that upon returning from the routine, the computer picks up the address it has stored, and returns to it. Though in reality this address is the address of the next instruction to be carried out, which would have been executed, had not a call to a routine been made.
These addresses, as there may be more than one of them, if a call is made from within a routine, are stacked up on top of each other.
Thus, in our example above;

First a call to "USR 40000" is made, this stores the address of BASIC, so our stack of addresses looks like this;

[BASIC]

Then, a jump is made to the routine called #S, since this is a jump, no return address is stored and so our stack still looks like this;

[BASIC]

At the end of the routine, a return instruction is found, and in response the computer takes the top address from the stack, BASIC and returns to that address.
To demonstrate this principle further, consider the following short program;

```
 10 REM org;40000,50000:
 20 REM call;A:
 30 REM note;Location 1:
 40 REM routine;
 50 REM label;A:
 60 REM put;56,10,4,SCOPE
 70 REM call;B:
 80 REM note;location 2:
 90 REM end;
100 REM routine;
110 REM label;B:
120 REM put;57,10,10,COMPUTER
130 REM call;C:
140 REM note;location 3:
150 REM end;
160 REM routine;
170 REM label;C:
180 REM put;58,10,19,LANGUAGE
190 REM end;
200 REM exit;
```

Having compiled this, by "PRINT USR 60500" the stack will, as far as we are concerned look like this;

Precisely, empty!

If we now run the program by way of "RAND USR 40000" the return address for BASIC will be stacked; thus our stack of return addresses will look like this;

[BASIC]

The program then "calls" label A, but before actually passing control to the routine which is called "A", the return address "Location 1" is stacked. So the stack at this point becomes;

[Location 1]
[BASIC]

The routine which is called "A" now prints the word "SCOPE" in black ink, before calling label B. So again, a return address is stacked, in this

case “Location 2”, so the stack of return addresses now looks like this;

[Location 2]

[Location 1]

[BASIC]

The routine which is called “B” now prints the word “COMPUTER” in blue ink beside the existing word “SCOPE”, before making a call to label C, which causes the return address, “Location 3” to be stacked onto the stack of return addresses.
The stack now looks like this;

[Location 3]
[Location 2]
[Location 1]
[BASIC]

The routine which is called “C” now prints the word “LANGUAGE” in red ink beside the two existing words.
The program now encounters a return instruction, written into the routine area by the presence of the word “end”
The computer now takes the address from the top of the stack of return addresses and returns to it;

So from the stack off comes “Location 3”;

[Location 3]
[Location 2]
[Location 1]
[BASIC]

Upon returning to “Location 3”, the program encounters another return instruction put there by the “end” instruction, and so off comes the next address from the top of the stack, and a return is made to it;

[Location 2]
[Location 1]
[BASIC]

Once again, at “Location 2” another return instruction is encountered, and so off comes the top address from the stack;

[Location 1]
[BASIC]

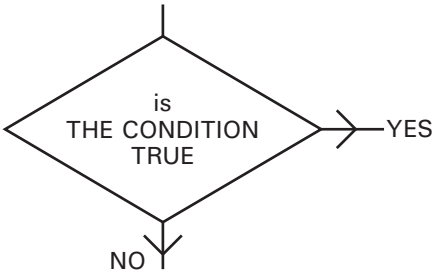
Returning to “Location 1” the program encounters another return instruction, only this one has been written by the “exit” command, and so the top address from the return stack is taken off, and a return made to it;

[BASIC]

Thus leaving the stack as we found it, empty, and our program executed.

Condition Testing

Without a doubt, the most essential aspect of a computer is it's capacity to quickly make decisions.
In BASIC this form of decision making takes the format of "IF . . . THEN" and "IF . . . THEN . . . ELSE".
That is to say, if a statement is true then branch to another part of the program, or if a statement is true then branch to one part of a program, but if the statement is not true then branch to another section.
This idea is best illustrated with a flowchart;



Notice how when a program encounters a decision, like the one above, if the condition is found to be true, then the program branches along the "YES" or "TRUE" path. If however the condition is true, then program execution continues from the next instruction in sequence by travelling along the "NO" or

"FALSE" path.

In BASIC a typical decision may be for example;

"IF X = 10 THEN GOTO 500"

SCOPE however, does not use "IF . . . THEN" condition testing, but instead uses an instruction called "test", which takes the following format;

test;operator,var under test,comparative,branch address:

Although this looks complex, it is quite easy to grasp.
The "operator" is a number which defies the form of test to be performed, and the "var under test" is the "var" which is being tested for a condition against the "comparative" which is either a number or a "var", the branch address is the name of a label to which the program will branch if the condition is found to be true.

The "operator" may test for one of four conditions;

1] Equality =

2] Inequality < >

3] Less than <

4]Greater than or equal to > =

In addition, the "operator" also specifies whether the branch if the condition is true will be either a "jump" or a "call".

There are therefore eight possible operators to choose from, and these are:

Operator	*Condition*
194	JUMP if the comparative tested is not equal to the var tested
196	CALL if the comparative tested is not equal to the var tested
202	JUMP if the comparative tested is equal to the var tested
204	CALL if the comparative tested is equal to the var tested
210	JUMP if the comparative tested is greater than or equal to the var tested
212	CALL if the comparative tested is greater than or equal to the var tested
218	JUMP if the comparative tested is less than the var tested
220	CALL if the comparative tested is less than the var tested

Thus; if we call the var "x" and the comparative "y" the following is true;

Operator	*Condition*
194	JUMP if x<>y
196	CALL if x<>y
202	JUMP if x=y
203	CALL if x=y
210	JUMP if x=<y
212	CALL if x=<y
218	JUMP if x>y
220	CALL if x>y

So let us take an example;

We wish to test the var "a" and if it is equal to 255 then we wish to jump to a label called "B";

The "test" we could use would look like this;

```
50 REM test;202,a,255,B:
                ↑  ↑  ↑  ↑
                o  v  c  l
```

Notice that below this example line I have indicated the "operative" with an "o", the "var under test" with a "v" the "comparative" with a "c" and the "branch address" with a "l".

As this is a complex instruction, here is an example of a short little program which uses "test" in a practical manner;

```
10 REM org;40000,50000:
20 REM var;x,0:
30 REM label;A:
40 REM put;56,10,x,★
50 REM inc;x,1:
60 REM test;194,x,32,A:
70 REM exit;
```

Compile this, with "PRINT USR 65000", and run it by entering "RAND USR 40000".
Notice how very rapidly a line of ★ appears across the centre of the screen. But stops at the far left side, without giving an "INTEGER OUT OF RANGE" error report.
As I said, the program is very simple;
Line 10 sets up the parameters for the program to be rewritten to by SCOPE.
Line 20 initialises a "var" to a value of 0.
Line 30 is a reference point, and indicates the start of a loop.
Line 40 displays a ★ at the co-ordinates 10,x.
Line 50 increases the value of the var x by 1, so as to effectively move the pront position one place to the right.
Line 60 checks the value of x, so as to ensure that it has not become the value 32 which is the first "off screen" value to the right.
If this test passes, that is if x<>32 then the program jumps back to label "A".
Line 70 returns control back to BASIC.

Another way to do this program would be to add a line 25 which reads;

25 REM var;y,32:

And to change line 60 to read;

60 REM test;194,x,y,A:

This would then perform in exactly the same way as before, the only difference being that the var under test, "x", is compared with the value of another var, "y" rather than a definite numeric value.

You will notice that the word "test" caters solely for "var"s, for testing "bvar"s a special word exists; "lim", which takes the format;

lim;bvar,comparative,branch address:

The "comparative" here is a number only, and a jump will be made to the label indicated as the "branch address".

Thus should you require "lim" to call a routine, the following system may be used:

50 REM lim:A,60000,C:
60 REM ...
80 REM label;C:
90 REM call:R:

After the "call;R:" you would probably require to jump back to a label preceeding the "lim" instruction.
Notice that "lim" operates a "jump if equal" system, thus in the above example, a jump would be made to C if bvar A=60000.

Essential Extras

It is often essential to be able to clear the screen of all characters, plots, lines and so on.
From BASIC, the command “CLS” is used to achieve this, but this comand always clears the whole screen. Sometimes it is required to only clear a part of the screen, and this facility is provided by the SCOPE word “wipe” which takes the format;

wipe;a number or var:

The value after the word refers to how many lines of the screen are cleared, and ranges from 1 to 24, counting from the bottom of the screen.
Notice that a value of 24 will clear the entire screen, and that a number larger than 24 may cause a system crash.

Often during a program it is found that at a suitable point a delay is required, which from BASIC may be achieved by a “PAUSE”.
SCOPE’s equivalent is “halt” which has the format;

halt;a number or a var:

Halt stops the program for 1/50th of a second per the value of the number or the var following the word.
Thus, a value of 255 gives a delay of 5 seconds, and a value of 50 gives a delay of 1 second.

A very useful aspect of BASIC is it’s capacity to change or read the value within a memory address using “POKE” and “PEEK” respectively.
SCOPE has adopted these two functions in the form of “mem” which has the format; mem;a bvar, a number or a var:

Mem “POKE”s the address indicated by the value held in the bvar, with the value of either the number or var.

The BASIC command “PEEK” is replaced by “view” which follows the format;
view;a bvar, a var:

View assigns the var specified with the value held in the memory address indicated by the value of the bvar.

Reading the Keyboard

Without a doubt all games require some form of player-computer interaction, which may be achieved with either "INPUT" or "INKEY$" from BASIC.
SCOPE does not have any form of "INPUT" command, but it does have an "INKEY$" type of function called "get" which follows this format;

get;a var:

This word scans the key board, and places the "code" of any key press into the specified "var" or a value of 0 if one key is not being pressed. The following is a simple key scan routine;

```
1000 REM routine;
1010 REM label;G:
1020 REM get;k:
1030 REM test;202,k,0,G:
1040 REM end;
```

It is worthwhile noting that the Sinclair Spectrum's system variables "REPDEL" and "REPPER" have an effect upon the speed at which "get" operates.
Thus it will be found that normally "get" is fairly sluggish to use, this can be changed by the following;

```
10 REM org;40000,50000:
20 REM bvar;A,23561:
30 REM mem;A,1:
40 REM bvar;A,23562:
50 REM mem;A,1:
```

rest of your program

```
9000 REM bvar;A,23561:
9010 REM mem;A,35:
9020 REM bvar;A,23562:
9030 REM mem;A,5:
9040 REM exit;
```

It is important to switch back the old values into these system variables before returning to BASIC, as other wise!
Well, you can try it for your self and find out!

Scrolling

SCOPE has two commands with regard to scrolling the screen display. The first, “scr” scrolls a specified number of lines up the screen and takes the format;

scr;a number or a var:

Where the number or var refers to the number of lines to be scrolled, counting from the bottom of the screen.
The range is from 3 to 24, where a value of 24 scrolls the entire screen up one line.
As with “wipe” you should take care not to attempt to scroll more than 24 lines or less than 3, or else the system may crash.

SCOPE also has the capacity to scroll the entire screen in any of four standard compass directions but one pixel at a time, with the word “fscr” which is an abbreviation for “fine scroll” and takes the format;

fscr;a number:

Where the number indicates the direction for scrolling.
These numbers are as follows;

5 = scroll left
6 = scroll down
7 = scroll up
8 = scroll right

As you will notice they are relevant to the numbers printed on the cursor control keys, and this is so that you may remember which number scrolls in which direction.
It should be pointed out that “fscr” only scrolls the display file, and not the attribute file, and thus leaves “colours” behind, an effect which can be amusing, annoying or useful.

Character collision

Often, as with the previous simple game, it is neccessary to detect a collision between two characters.
This may be achieved by use of SCOPE's "attr" word which follows the following format;

attr;a var,line,column:

Both line and column may be either a number or a var.
This word assigns to the specified var the attribute value of the screen position indicated by the line and column co-ordinates.
Thus, the following example will set the "var" "a" to a value of 57.

```
10 REM org;40000,50000:
20 REM put;57,5,10,SCOPE
30 REM attr;a,5,10:
40 REM exit;
```

However, as it stands this is not a very useful feature, that is until you understand fully how to use it.
In the previous game, the attribute at the player's piece position was checked for a value corresponding to the other objects which the player was trying to avoid, and by checking this value, the program was able to detect whether or not the player had collided with an obstacle.
Notice that the value returned in the "var" by "attr corresponds to the colour combination at the specified co-ordinates, and this colour combination in turn corresponds to the table given in the section dealing with colour and graphics.

Sound

We have deliberately used the word “sound” as distinct from the word “music” because it is used to produce a sound as distinct from a musical note, (what do you expect from a piezoelectric transducer?).
Because of its speed SCOPE can handle repetitive sound statements and produce results where the pitch of the note alters at such a rate as to produce a glissando sound.

The command “sound” follows the following format;

sound;pitch,delay:

Where both pitch and delay may be either a number or a “var”.
The following table gives an indication to the various values which may be used to produce sounds of various tones.;

Pitch	*Octave*	*Max Delay*
255	C-2	10
127	C-1	20
63	MIDDLE C	40
31	C+1	80
15	C+2	160
7	C+3	255

Notice that the “max delay” number is the longest recommended delay to be used with a corresponding pitch number, longer delays may cause the sound to continue for up to fifteen minutes!
Notice also that as the pitch number increases in value, so the tone of the note decreases.

Sprite Graphics

Another new addition to the Sinclair Spectrum's range of graphical capabilities provided by SCOPE are 8 user-definable sprites, abbreviated to 'uds".
Each of these sprites is composed of a 16 by 16 grid of pixels, similar to a "UDG" but four times larger.
The sprites may all or each be displayed and moved around the screen at will, and have integral collision detection.
This collision detection which is detected by "view"ing memory address 39272 reveals whether by returning the value 255 sprite/s have "collided" with other objects or sprites on the screen.
Notice that a collision takes place when two "on" or linked in pixels meet. The word for displaying a sprite is "uds" which takes the format;

uds;sprite number,colour,line,column:

The sprite number may range from 0 to 7 inclusive, and the colour is the standard combination colour as taken from the table in the section on "colour and graphics".
The line and column co-ordinates refer to the top left corner of the sprites grid, and range from 0 – 175 for the line and 0 – 255 for the column number.
Notice that a sprite may be made to glide gracefully off one side or the top or bottom of the screen as desired.
The sprites are displayed in an "xor"ed fashion or "overed" if you prefer, and so to wipe a sprite off the screen you *must* re-display it.

Included on your cassette tape is a program to design and generate your sprites, which is easy to use and self-explanatory.

Scope Mathematics

At first appearances, SCOPE's mathematical capacity is very low, being, composed entirely of the two basic operations addition and subtraction.
However, these immediately give rise to the capacity to multiply and divide, if you can add, then you can multiply and if you can subtract and add then you can divide.
It should be remembered though that SCOPE works solely with integers, and most conveniently with integers in the range 0 to 255 inclusive, a range which does not provide for advanced mathematics, but is sufficient for many applications which may be required when writing a game.
First of all then a multiplication routine;

```
1000 REM bvar;R,0:
1010 REM var;x,?:
1020 REM var;y,?:
1030 REM label;M:
1040 REM add;R,x:
1050 REM dec;y,1:
1060 REM test;194,y,0,M:
```

In the above routine, the vars "x" and "y" hold the two values to be multiplied, and the section from label "M" onwards multiply them together and store the result in the bvar "R".
Thus the routine is based on the equation.

$R = x \star y$

The use of a bvar for the result rather than a var, allows the multiplication routine to handle integers up to and including 65535. Thus this routine cannot overflow, as the largest values which may be held in the vars is 255 each, and $255 \star 255 = 65025$, and $65025 < 65535$

It is also possible to multiply the two vars together, and store the result in the first var, in the form of the equasion;

$x = x \star y$

This however suffers from the problem of a possible overflow, remember that vars operate in modulo 256.
But there is that routine in any case, as it does have a use;

```
1000 REM var;c,y:
1010 REM label;M:
1020 REM inc;x,x:
1030 REM dec;c,1:
1040 REM test;194,c,1,M:
```

Notice how by transfering the value of var "y" into var "c" the value of var "y" is preserved.

Division is a shade more difficult, but here again is the routine;

```
1000 REM bvar;R,0:
1010 REM label;D:
1020 REM dec;x,y:
1030 REM test;218,x,200,#D:
1040 REM add;R,1:
1050 REM jump;D:
1060 REM label;#D:
```

The result is here again stored in the bvar "R", but this could easily be a var, just by changing lines 1000 and 1040 to read;

```
1000 REM var;R,0:
1040 REM inc;R,1:
```

Line 1030 makes a check to see whether the var "x" has gone below 0 in value, remember that in modulo arithmetic numbers "wrap around". The comparative value of 200 may be changed as required, but remember that the test is for x > 200.
This routine is based upon the equation;

$R = x/y$

But x is destroyed, it could be preserved by first switching its value into an ununsed var, such as "c" in one of the above examples.
Special thanks are due to Sheila Newman for reminding me of the method of doing "long division" as they call it in middle schools, yet another seemingly useless aspect of mathematics I dismissed at the time.

So far I have stated that SCOPE has the capacity only to handle positive numbers, that is from 0 to 255 or 0 to 65535.
It does however also have the capacity to operate in a numeric representation system known as "2's compliment".
By working in 2's complement, the range of numbers which may be handled stretches between −128 and +127 or −32768 to +32767.
However, to work with 2's complement requires the programmer to formulate his or her own system, as SCOPE does not provide the function automatically.
Whether or not you will ever have any use for negative numbers within a SCOPE Program is perhaps debatable, but it is useful to cover as many possibilities as possible.
Here then is a short system for operating in modulo 256 2's complement, that is with a number range between −128 and +127.

```
1000 REM bvar;R,A:
1010 REM inc;x,9:
1020 REM test;212,x,128,M:
1030 REM add;R,x:
1040 REM num;56,R,0,1:
1050 REM routine:
1060 REM label;M:
1070 REM put;56,0,0,-
```

```
1080 REM var;m,0:
1090 REM dec;m,x:
1100 REM end;
```

This routine adds together two vars (x and y) in a 2's complement fashion, detecting for a negative result (line 1020) and displaying the result of the calculation at the top left of the screen.
Before we can actually use this routine however, we need to know how a negative number is represented in 2's complement.
The answer is fairly straight forward.
Let us take for example the number −5.
Since our actual range of numbers is from 0 to 255, we obviously cannot represent −5 as "−5".
Instead we convert the number "−5" into 2's complement by adding "−5" to 256, that is;

256 + −5 becomes 256 − 5
256 − 5 = 251

So "−5" is represented in 2's complement as "251".
We can prove this to be correct.
We know the −5 + 5 = 0.
So, it follows that if 251 is the same as −5, then 251 + 5 will also equal 0. To test this we must use a modulo 256 arithmetic system, because otherwise we will not obtain a correct result.
So, to test whether 251 + 5 does or does not equal 0 run the following;

```
10 REM org;40000,50000:
20 REM chg;56:
30 REM bvar;R,0:
40 REM var;x,251:
50 REM var;y,5:
60 REM inc;x,y:
70 REM add;R,x:
80 REM num;56,R,10,10:
90 REM exit;
```

When you run this you will see the number 0 appear in the middle of the screen, so 251 + 5 does equal 0, and therefore 251 is the same as −5.

We are now presented with yet another problem, how do we add our minus number to 256, when the largest number that we can store is 2552.
If you remember, we were calling the number system "modulo 256", and not "modulo 255", the reason being that there are a total of 256 numbers in the system, that is 0 to 255 inclusive.
Now, if we add (in standard base 10) 1 to 255 we obtain the result 256, agreed? good!
So, if in modulo 256 we add 1 to 255 we also get the result 256. Though it appears as though we have the result 0, which we do in fact also have!
This is obviously going to take a bit of explaining, and to do this I must digress slightly.

Consider if you will please the following situation.
There are 24 hours in a day/night cycle.
So, we may calculate the time in a 24 hour clock system which runs from time 0 hour, which is midnight to time 23 hours which is 11 o'clock at night.
This time system thus runs in modulo 24.
So, if we imagine the passage of time to go rather quickly, then 1 hour on from time 23 hours is time 0 hour.
So, if I was to get into bed at 0 hour on Monday morning, and stay in bed for the next 23 hours I will have been in bed for 23 hours.
If I then stayed in bed for another hour the time would be 0 hour, but I will have stayed in bed for; 1 cycle of 24 hours and 0 extra hours.
Can you see then that in modulo 256, 2551 + 1 is equal to 1 "256" and no extra units?
Thus, in modulo 256, the number 0 can represent both 0 and 256.
So, when we wish to add a minus number to 256, we add it to 0 instead, because;

1] We cannot hold the number "256„ as 256
2] The number 0 may represend "256"

So, the number "–5" may be calculated into 2's compliment by;

"0 + –5 = 251"

Or as a SCOPE program we might write this as;

```
1000 REM var;Z,0:
1010 REM dec;Z,N:
```

Where "Z" stands for and holds the value 0, and "N" stands for the holds the value of our number.
Why have I used "dec" when I have been talking about adding?
If you remember your school mathematics, a "+" when added to a "–" is the same as subtracting a "+" from a "+". Thus;

10 + –5 is the same as 10 – 5

and;

–5 + 10 is the same as 10 – 5

Let's do an actual test example using the routine I showed you at the start.

```
   10 REM org;40000,50000:
   15 REM chg;56:
   20 REM var;x,10:
   30 REM var;z,01:
   40 REM var;y,5:
   50 REM dec;z,y:
   60 REM var;y,z:
   70 REM note; y is now in 2's complement as "-5":
1000 REM bvar;R,0:
1010 REM inc;x,y:
1020 REM note;x is now equal to x+-5 or x-5:
1030 REM test;220,x,128,M:
1040 REM add;R,x:
1050 REM note;transfer result into a bvar so that it can be displayed:
1060 REM num;56,R,0,1:
1070 REM routine:
1080 REM label;M:
1090 REM put;56,0,0,-
1100 REM var;m,0:
1110 REM dec;m,x:
1120 REM var;x,m:
1130 REM end;
1140 REM exit;
```

This example will display the number 5 at the top of the screen, because 10+-5=5!
You might like to try substituting other values for x and y, bearing in mind however that in the above example, what ever value is put into y is made negative, and as such must be in the range 0 to 128 inclusive before negation takes place, and the value in x must not exceed 127, as we are working in 2's complement.

Input

"Input", is a very useful program which allows for a number within the range 0 to 255 to be "input" by the user.
As it stands, the program will only allow for a small number to be entered, but the range may be easily increased to 999 from the present 255 by changing the lines which read "var" r to read "bvar R" and other associated changes, such as instead of "inc;r" to "add;R".

```
 10 REM org;40000,50000:      organise memory layout
 20 REM wipe;24:              CLS
 30 REM var;a,0:              initialise
 40 REM var;b,0:              variables
 50 REM var;c,0:              to
 60 REM var;r,0:              0
 70 REM bvar;A,0:
 80 REM bvar;B,0:
 90 REM bvar;C,0:
 95 REM put;184,21,0,L Print a flashing cursor
100 REM label;a:              1st input stage
110 REM get;a:                check for not in the range
120 REM test;210,a,47,a:          0 to 9
130 REM test;218,a,58,a:
140 REM dec;a,48:             obtain CHR$
150 REM add;A,a:              Print
160 REM num;56,A,21,1:            obtained number
170 REM label;b:              next input stage
180 REM get;b:
190 REM test;202,b,13,f:
200 REM test;210,b,47,b:
210 REM test;218,b,58,b:
220 REM dec;b,48:
230 REM add;B,b:
240 REM num;56,B,21,2:
250 REM label;c:              last input stage
260 REM get;c:
270 REM test;202,c,13,f:
280 REM test;210,c,47,c:
290 REM test;218,c,58,c:
300 REM dec;c,48:
310 REM add;C,c:
320 REM num;56,C,21,3:
330 REM label;f:              interpret "got" key strokes into
340 REM test;202,b,13,u:          a number
350 REM test;202,c,13,t:
360 REM var;r,0:              initialise "result" variable to 0
370 REM label;h:
380 REM inc;r,100:
390 REM dec;a,1:
```

```
400 REM test;194,a,0,h:
410 REM label;i:
420 REM inc;r,10:
430 REM dec;b,1:
440 REM test; 194,b,0,i:
450 REM inc;r,c:
460 REM jump;e:
470 REM label;t:
480 REM var;r,0:
490 REM label;j:
500 REM inc;r,10:
510 REM dec;a,1:
520 REM test;194,a,0,j:
530 REM inc;r,b:
540 REM jump;e:
550 REM label;u:
560 REM var;r,a:
570 REM label;e:          "end" stage
580 REM bvar;R,0:         copy result into a bvar ready
590 REM add;R,r:          for
600 REM num;56,R,0,0:     printing
610 REM REM exit;         exit to BASIC
```

This program may be divided into sub-sections and these sub-sections may be named after the label which signifies their start within the program, thus we have;

1] A,B and C which read the key board, and continue to do so until it is found that the user is pressing a key in the range "0" to "9" or "ENTER".
These sections display the CHR$ of the key being pressed, if it is a "numeric" key.

2] E tests the subsequently obtained number for one of three conditions;
a) Smaller than 10, b) greater than 10, but smaller than 100 and c) greater than 100.
Depending on which condition the number is, the program jumps to the appropriate section, to convert the three variables "a", "b" and "c" into one value to be held in variable "r".
The sections jumped to are;
If the number is less than 10 the program jumps to U.
If the number is within the range 10 to 99 the program jumps to T.
Otherwise the program continues as it was going.

3] E copies the number into a bvar so that it may be displayed, and display the number at the top left of the screen.

The capacity to be able to "input" a number is very useful, but our program above would perhaps be better if we could enter a number larger than 255, though this would mean storing the obtained number in a bvar, and bvars are not as friendly as vars.

To change the program to accept numbers in the increased range of 0 to 999, the following lines need to be changed;

```
360 REM bvar;R,0:
380 REM add;R,100:
420 REM add;R,10:
450 REM add;R,c:
480 REM bvar;R,0:
500 REM add;R,10:
530 REM add;R,b:
560 REM bvar;R,0:
565 REM add;R,a:
```

The following lines need to be deleted;

```
580
590
```

Notice the addition of line number 565.

Guess the Number

"Guess the Number" is a simple guessing game based around the "input" program we have just discussed.
The object of the game is to deduce in as few attempts as possible the random number generated by the computer.
To help you with your guesses, the computer will, after each guess has been entered, tell you whether your guess was correct, lower than its number, or higher than the number it has generated.
Notice that the "input" program has been slightly improved here, with the addition of a short section which, after a third numeral has been typed in waits for the "ENTER" key to be pressed, rather than the original version which, after the typing in of a third numeral would go straight ahead into the deciphering stage.
I will not make any claims about this being a "well constructed" program, as it is in reality fairly badly put together, but the game is fun to play, or at least I think so, and it demonstrates particularly well the use of SCOPE condition testing, and the way in which with a little thought, many of the functions you thought SCOPE couldn't handle, it in fact can be made to handle fairly well.

```
10 REM org;40000,50000:
 15 REM wipe;24:
 20 REM bvar;T,0:
 25 REM call;x:
 30 REM label;y:
 40 REM put;56,21,1:   (6 spaces)
 50 REM put;184,21,0,L
 60 REM var;a,0:
 65 REM var;b,0:
 70 REM var;c,0:
 75 REM var;r,0:
```

```
  80 REM bvar;A,0:
  85 REM bvar;B,0:
  90 REM bvar;C,0:
 100 REM label;a:
 110 REM get;a:
 120 REM test;210,a,47,a:
 130 REM test;218,a,58,a:
 140 REM dec;a,48:
 145 REM add;A,a:
 150 REM num;56,A,21,1:
 160 REM label;b:
 170 REM get;b:
 175 REM test;202,b,13,f:
 180 REM test;210,b,47,b:
 190 REM test;218,b,58,b:
 200 REM dec;b,48:
 205 REM add;B,b:
 210 REM num;56,B,21,2:
 220 REM label;c:
 230 REM get;c:
 235 REM test;202,c,13,f:
 240 REM test;210,c,47,c:
 250 REM test;218,c,58,c:
 260 REM dec;c,48:
 265 REM add;C,c:
 270 REM num;56,C,21,3:
 280 REM label;g:
 285 REM var;k,0:
 290 REM get;k:
 295 REM test;194,k,13,g:
 300 REM label;f:
 310 REM test;202,b,13,u:
 320 REM test;202,c,13,t:
 330 REM var;r,0:
 340 REM label;h:
 350 REM inc;r,100:
 360 REM dec;a,1:
 370 REM test;194,a,0,h:
 420 REM label;i:
 430 REM inc;r,10:
 440 REM dec;b,1:
 450 REM test;194,b,0,i:
 460 REM inc;r,c:
 470 REM jump;e:
 500 REM label;t:
 510 REM var;r,0:
 520 REM label;j:
 530 REM inc;r,10:
 540 REM dec;a,1:
```

```
 550 REM test;194,a,0,j:
 560 REM inc;r,b:
 570 REM jump;e:
 600 REM label;u:
 610 REM var;r,a:
 700 REM label;e:
 710 REM bvar;R,0:
 720 REM add;R,r:
 725 REM put;56,7,1, (3 spaces)
 730 REM num;56,R,7,1:
 800 REM test;202,r,x,#c:
 810 REM test;210,r,x,i:
 820 REM test;218,r,x,#h:
 830 REM label;#c:
 840 REM put;57,7,7,correct!
 850 REM num;57,T,7,20:
 860 REM jump;z:
 870 REM label;i:
 880 REM add;T,1:
 890 REM put;56,7,7,too low!   (2 spaces)
 900 REM num;56,T,7,20:
 910 REM jump;y:
 920 REM label;#h:
 930 REM att;T1:
 940 REM put;56,7,7,too high!
 950 REM num;56,T,7,20:
 960 REM REM jump;g:
 970 REM routine;
 980 REM label;x:
 990 REM put;56,0,8,GUESS THE NUMBER
1000 REM put;60,5,0,GUESS STATUS ATTEMPTS
1010 REM rnd;x,255,0:
1020 REM end;
1030 REM label;z:
1040 REM exit;
```

De-bugging

This section has been written for compassionate reasons for any user who has ever experienced problems in getting a program to work properly.
How many times have you sat back having spent several weeks, days, hours okay minutes typing in a long and involved program only to run it when . . . After the initial exclamations of despair, the most common phrase when a program goes wrong seems to be;

"Now why did it do that?"

Should your program fail to compile properly or when run it doesn't do what you hoped I would try following these steps;

1] Keep calm!

2] Stop!

3] Think!

Okay, I'm a hypocrite, when my programs go wrong I certainly do not keep calm, but sooner or later we all calm down, and then it's time to stop, and think about what happened, and why it could possibly have occured.
Some of the most common faults encountered with SCOPE programs are;

1] Syntax errors
2] "NEW"ing crashes
3] Endless loops

The easiest way to check on possible causes of an error, or the breed of the bus if you would prefer, is by consulting a table, so here's one;

continued *continued*

Error encountered	Possible causes
Error encountered	Possible causes
Draw fails to operate	Missing +/– prefix.
Endless loop	You may have forgotten to include an "exit" command at the end of your program. Check that any loops you have which test use a test against a counter, the counter may never actually hold the value being tested for to end the loop.
General "crash"	Attempting to wipe or scr more than 24 lines. Check mem commands.
General problems	Check that you have not referred to either a label or a var by a capital letter when it should be a small letter and vice versa.

INTEGER OUT OF RANGE error	Xput vertical co-ordinate smaller than 8 plot vertical co-ordinate larger than 175. Attempting to "put" either too far to the right, or too low. Check that you haven't put the colour operand in the wrong place in either a num or a put statement.
"NEW"ing crash	Jumping or calling a non-existent label. If this occurs during compilation, check that all bvars are capital letters.
Peculiar colour to lower two lines of the display	Using a number not within the range 0 to 7 with a bdr statement, this word will change the attributes of the lower two lines of the screen where you cannot normally print, as well as changing the border colour.
Returns to BASIC for no reason	Jumping to a routine, check your test numbers. The label used in conjunction with a lim statement may be a routine, remember, lim always jumps top the label specified upon satisfaction to the condition.
Syntax error report	Missing colon from the end of the line. Insufficient operands, (especially xput). Missing comma separating operands. Missing semi-colon after word.
Very long "Beep"	Incorrect duration value, check all sound statements, especially those using vars.
Will not return to BASIC	You could be calling a routine, and then jumping out of it back to the program.

Admittedly this table of conditions and possible causes is fairly brief, but, it is really up to you, the programmer, in the end to discover for yourself what will work, and what doesn't.

This is perhaps equally good advice, computer programming is not a battle of your intelligence against the intelligence of a machine, it is a battle between your logical reasoning and your illogical reasoning. Remember, a computer is a pathetic collection of plastic and metal, carefully and skillfully welded, screwed, and glued together, it has no reasoning powers, no malice against you, or collection of tiny creatures which dwell within its mass of circuitry for the sole purpose of ruining your latest masterpiece.
When a program you have written goes wrong, do not blame the computer, it is your reasoning which is at fault.

Logical reasoning and computer programming are two skills which everyone has the capacity to develop, and they cannot really be taught, only learnt by practice, experience and most of all experimentation.
Don't be afraid of your computer, don't be afraid of programs crashing or going wrong, you can always pull the plug out, and no lasting damage will have been done. But at least you may have learnt something.
If you suffer a crash, ask your self "Why?", and don't stop asking until you have found out.
If you get really stuck, ring me on the programmer's hot line, both it and me are there for YOUR benefit, we at ISP marketing Ltd want you to become a competent SCOPE programmer, so don't be afraid or embarrased to ask, you might be able to give me some good tips in return.
So beat the bugs, and write yourself some really good games software which you'll not only be proud to own, but you'll enjoy playing as well.

Dictionary

Throughout this dictionary, the following abbreviations are used;

b refers to a bvar
c refers to a colour
1 refers to a label
n refers to a number
nv refers to a number or a var
v refers to a var

Word	*Format*	*Result*
add	b,nv	Increases the value of the named bvar by the value of the number or var.
attr	v,nv1,nv2	Sets the value of the named var to the attribute value of the character at co-ordinates nv1,nv2.
bdr	nv	Sets the border to colour nv.
bvar	b,n	Assigns a value n to the named bvar.
call	1	Branches program execution to the routine starting at label 1. Notice, ensure that the label called is within a "routine".
chg	nv	Changes the permanent attributes of the screen (PAPER, INK, BRIGHT, FLASH) to the state corresponding to an attribute value nv. Also causes a CLS to occur.
dec	v,nv	Decreases the value of the specified var by the value nv.
draw	c,nv1,nv2	Draws a line in colour c, nv1 pixels up or down depending on the presence of either a "+" or "–", and nv2 pixels right or left, again depending upon the presence of a "+" or "–" prefix.
end	none	Terminates a routine.
exit	none	Terminates compilation, and writes a return to BASIC instruction at the end of the SCOPE program area.
fscr	n	Causes a single pixel scroll in the direction n. Error is n is not in the range 5(=n=(8.
get	v	Causes a key scan to take place and the result to be placed in var v.

globe	nv1,nv2, nv3	Draws a circle with its centre at pixel co-ordinates nv1,nv2 and of a radius nv3.
halt	nv	Stops program execution for nv 1/50ths of a second.
inc	v,nv	Increases the value of the specified var by the value nv.
jump	1	Branches program execution to program sub section 1. Notice, 1 must not be a "routine".
label	1	Establishes a reference point.
lim	b,n,1	Causes program execution to branch by way of a "jump" instruction to label 1 upon satisfaction of the condition b = n.
mem	b,nv	Pokes the memory address held by the bvar with the values nv.
minus	b,nv	Decreases the specified bvar by the value nv. Notice, nv will be considered as being represented in modulo 256.
note	text	None.
nmum	c,b,n1,n2	Displays the value of the specified bvar in colour c at co-ordinates n1,n2.
org	n1,n2	Forces memory organisation for the compiled program, with the program area starting at n1, and the routine area starting at n2.
over	n	Sets or unsets the "over" flag, if n = 1 then over flag is set, if n = 0 then over flag is unset.
plot	c,nv1,nv2	Inks in the pixel nv1,nv2 in attribute c.
put	c,nv1,nv2, text	Displays the character or characters "text" in attribute c at co-ordinates n1,n2.
rnd	v,n1,n2	Assigns a random value to the specified var. the value is within the range; (0 to n1) + n2.
routine	none	Causes subsequent commands to be compiled as a routine into the routine area until an "end" statement is encountered.
scr	nv	Scrolls nv – 3 lines of the screen display up one character line, thus nv = 24 scrolls the entire screen up.
sound	nv1,nv2	Sounds the "beeper" at a pitch nv1 for duration nv2. Notice, nv2 is relative to nv1.
test	n,v,nv,1	Conditionally tests v against nv, and if the condition is satisfied, the program branches to label 1.

uds	n,c,nv1,nv2	Displays sprite number n, in attribute c at pixel co-ordinates nv1,nv2. Notice, nv1,nv2 refers to the top left corner of the sprite's possible grid.
var	v,nv	Assigns a value nv to the specified var.
view	b,v	Assigns a value peek(b) to the var.
wipe	nv	Clears nv number of lines of the screen where nv = 24 clears the entire screen.
xput	nv1,nv2,c,n, text	Displays character or characters "text" in attribute value c at pixel co-ordinates nv1,nv2 and in horizontal width n. Notice, nv1,nv2 refers to the top left corner of the first character's possible grid. Notice, when n = 0, horizontal width is normal, n = 1 gives double width and n = 2 gives half width.

Scope to Basic Equivalents

For many programmers new to SCOPE, a SCORE to BASIC equivalent of a statement or routine is a very handy item. By observing the BASIC line, which they understand very easily, and comparing it with its SCOPE equivalent, they are quick to learn SCOPE, and find programming with SCOPE a good deal easier.
it is for this reason that this section has been written, giving fairly typical BASIC program lines and commands, and beside them the SCOPE line or lines to perform the same task, but of course very much faster.

BASIC	*SCOPE*
BORDER 5	bdr;5:
CIRCLE 128,88,50	globe;56,128,88,50:
CLS	wipe;24:
DRAW 128,0	draw;56,+128,0:
FOR N=0 TO 10	var;n,0: label;?:
NEXT N	inc;n,1 test;194,n,11,?:
FOR N=0 TO STEP 2	var;n,0:
NEXT N	label;?: inc;n,2: test;194,n,12,?:
FOR N=10 TO 0 STEP –1	var;n,10: label;?:
NEXT N	dec;n,1: test;194,n,255,?:

FOR N=0 TO 21 FOR N=0 TO 31 PRINT AT N,M;"." NEXT M NEXT N	var;n,0: label;R: var;m,0: label;B: Put;56,n,m,. inc;m,1: test;194,m,32,B: inc;n,1: test;194,n,22,A:
GOSUB line	call;?:
GOTO line	jump;?:
IF ATTR(X,Y)=56 THEN LET E=0	attr;a,x,y: test;194,a,56,A: var;e,0: label;A:
IF INKEY$=”” THEN GOTO line (loop)	label;A: get;k: test;202,k,0,A:
IF X=5 THEN GOTO line	test;202,x,5,?:
IF X=5 AND Y=6 THEN GOTO line	test;194,x,5,A: test;194,y,6,A: jump;?: label;A:
IF A=B THEN LET C=C★3	test;204,a,b,A: . . . routine; label;A: var;d,c: inc;c,d: inc;c,d: end;
IF X+2)10 THEN GOTO line	var;z,x: inc;z,2: test;218,z,10,?:
LET N=15	var;n,15:
LET X=X+5	inc;x,5:
LET X=X+Y	inc;x,y:
LET X=Y	var;x,g:
LET X=Y–5	var;x,g: dec;x,5:
LET X=X+(INKEY$=”8” AND X(31)– (INKEY$=”5” AND X)0)	get;k: test;204,k,56,R: test;204,k,53,L: . . . routine;

	label;R: test;218,x,31,#R: inc;x,1: label;#R: end; routine; label;L: test;210,x,0,#L: dec;x,1: label;#L: end;
LET R=INT(RND*6)+0	rnd;r,5,0:
LET P=PEEK 50000	bvar;x,50000: view;x,p:
LET I=CODE INKEY$	get;i:
OVER 1	over;1:
PAUSE 50	halt;50:
POKE 50000,123	bvar;x,50000: mem;x,123:
POKE 128,88	plot;56,128,88:
PRINT AT 0,10;"SCORE"	put;56,0,10,SCORE
PRINT AT 0,16;S	num;56,S,0,16:
PRINT PAPER 7;INK 2;BRIGHT 1;AT 10,7;"HELLO!"	put 87,10,7,HELLO!
RANDOMIZE 5	bvar;R,23671: mem;R,5:
REM text	note;text:

From this table of equivalents you should be able to translate your BASIC into SCOPE quite easily.
However, there are a few points to notice with respect to SCOPE does not have equivalents to the following BASIC commands;

1] ABS
2] ACS
3] AND
4] ASN
5] ATN
6] BIN
7] CAT
8] CHR$
9] CLEAR
10] CLOSE#
11] CODE (except for INKEY$ equivalent)
12] CONTINUE
13] COPY

14] COS
15] DATA
16] DEF FN
17] DIM
18] EXP
19] FN
20] FORMAT
21] IN
22] INPUT
23] INT (all SCOPE numbers are integers)
24] INVERSE (but see colour & graphics section)
25] LEN
26] LIST
27] LLIST
28] LN
29] LOAD
30] LPRINT
31] MERGE
32] MOVE
33] NEW (but resets may be arranged!)
34] OPEN#
35] OR
36] OUT
37] POINT
38] RANDOMIZE (but see above)
39] READ
40] RESTORE
41] RUN (use jump to jump to the start of your program)
42] SAVE
43] SCREEN$ (use attr instead)
44] SGN (but see mathematics section)
45] SIN
46] SQR
47] STR$
48] TAB
49] TAN
50] VAL
51] VAL$
52] VERIFY

To summarise, SCOPE does not handle;

Strings
Saving and loading
Floating point arithmetic

Though many functions may be imitated by the use of careful “mem”s or “POKE”s.

Appendix 1

USEFUL SPECTRUM SYSTEM VARIABLES

Address	*Name*	*How to use*
23561	REPDEL	May be used to vary the delay between a key being held down and that key auto-repeating.
23562	REPPER	Set to a value of 1 for extra fast key auto-repeating.
23624	BORDCR	Use as an alternative to bdr.
23658	FLAGS2	Set to a value of 8 to force "CAPS LOCK".
23670/1	SEED	Set to a value as an alternative to RANDOMIZE 1, set 23670 to 1, and 2367 to 0.
23672/3/4	FRAMES	By "viewing" 23672 and "meming" this value into 23670, RANDOMIZE is effected, that is a random RANDOMIZE. May also be used for timing longer delays.
23677/8	COORDS	23677 holds x co-ordinate from which next "draw" statement will commence, and 23678 holds the y co-ordinate. You may set these to values instead of using "plot" to define the starting point for a "draw".
23681		A conveniently spare byte, handy for storing a number in.
23728/9		As above but two consecutive bytes.
23730/1	RAMTOP	This may be set to 38000 by "mem"ing 23730 with 112, and 23731 with 148 so as to protect a compiled program against "NEW" and "LOAD".

Music Maker!

The following program is a very untuneful demonstration of the word "sound" and general programming methods, I apologise now for the attrocious sounds which are generated by the program, but perhaps you could change the sound statements to bring the various pitches more into tune?

```
 40 REM over;0:
 50 REM call;1:
 60 REM over;1:
 70 REM var;x,3:
 80 REM label;A:
 90 REM call;B:
100 REM test;204,k,99,C
110 REM test;204,k,100,D
120 REM test;204,k,101,E
130 REM test;204,k,102,F
140 REM test;204,k,103,G
150 REM test;204,k,97,H
160 REM test;204,k,98,1
170 REM test;202,x,32,K
180 REM test;194,k,32,A
190 REM routine;
200 REM label;R:
210 REM get;k:
220 REM end;
230 REM routine;
240 REM label;C:
250 REM sound;63,30:
260 REM put;56,3,x,0
270 REM inc;x,1:
280 REM end;
290 REM routine;
300 REM label;D:
310 REM sound;58,38:
320 REM put;56,2,x,o
330 REM inc;x,1:
340 REM end;
350 REM routine;
360 REM label;E:
370 REM sound;52,30:
380 REM put;56,8,x,o:
390 REM inc;x,1:
400 REM end;
410 REM routine;
420 REM label;F:
430 REM sound;48,30:
440 REM put;56,7,x,o
```

```
450 REM inc;x,1:
460 REM end;
470 REM routine;
480 REM label;G:
490 REM sound;43,30:
500 REM put;56,5,x,o
510 REM inc;x,1:
250 REM end;
530 REM routine;
540 REM label;H:
550 REM sound;38,30:
560 REM put;56,4,x,o
570 REM inc;x,1:
580 REM end;
590 REM routine;
600 REM label;I:
610 REM plot;56,0,171:
620 REM draw;56,+255,0:
630 REM plot;56,0,155:
640 REM draw;56,+255,0:
650 REM plot;56,0,123:
660 REM draw;56,+255,0:
670 REM plot;56,0,107:
680 REM draw;56,+255,0:
690 REM plot;56,0,99:
700 REM draw;56,REM 4,-5:
710 REM draw;56,+2,0:
720 REM draw;56,+4,+5:
730 REM draw;56,0,+70:
740 REM draw;56,0,+2:
750 REM draw;56,+5,+3:
760 REM draw;56,+5,-3:
770 REM draw;56,0,-10:
780 REM draw;56,-19,-30:
790 REM draw;56,-1,-8:
800 REM draw;56,+2,-12:
810 REM draw;56,+8,-3:
820 REM draw;56,+11,-3:
830 REM draw;56,+1,+5:
840 REM draw;56,-6,-2:
850 REM draw;56,-1,-3:
860 REM draw;56,+2,-1:
870 REM put;57,14,1,USE KEYS A,B,C,D,E,F,G TO PLAY
880 REM put;57,19,3,PRESS BREAK/SPACE TO STOP
890 REM end;
900 REM exit;
```

Demonstration 2

As a result of SCOPE's speed, many visual and audio effects are possible which are not possible from BASIC.
This next short program or routine demonstrates how very simple, and quickly, SCOPE can be used to produce some quite dazzling effects.

```
 10 REM org;40000,50000:
 20 REM var;c,0:
 30 REM label;A:
 40 REM var;b,0:
 50 REM label;B:
 60 REM bdr;b:
 70 REM sound;c,1:
 80 REM inc;b,1:
 90 REM test;194,b,8,B:
100 REM inc;c,1:
110 REM test;194,c,80,A:
120 REM exit;
```

If you run this program, you will be presented with a rapidly changing multi-coloured border, and your ears will be greeted with a space age sliding sound.
The program is simply two nested loops, one using var "c" as a counter, the other var "b".
You might like to play around with this and use it as a routine within a larger program.
If the sound statement, line 70 is deleted, and line 110 changed to read;

```
110 REM test;194,c,255,A:
```

then the effect, though short lived, will be of a very multi-coloured border.
The width of the border stripes may be altered by introducing delays within the loop "B", and in this way many varied effects may be achieved.

Attacked!

This next program is a game written as a demonstration of how to use "attr" practically, and also to simulate a simple "invader" game.
It is called, "Atacked!", and it is rather simple, fast and fun.
It also provides another example of structuring.

three UDGs were used in the original, and these may be established by the following BASIC program, which after being run should be "NEW"ed.

```
 10 DATA 0,24,60,255,60,0,0
 20 DATA 1,1,1,255,255,255,255,255,128,128,128,255,255,255,255,
    255
 30 FOR N=0. TO 7
 40 POKE USR "A"+N,A
 60 NEXT N
 70 FOR N=0 TO 15
 80 READ A
 90 POKE USR "H"+N,A
100 NEXT N
```

Type this in, run it and then type NEW, don't worry about the UDGs they're not affected by "NEW".
Now you can take in the SCOPE program, remembering that the statements which apparently display either an "A" or "HI" really display the associated UDGs, and as such they should be replaced with their UDG counter parts.

```
 10 REM org; 40000,50000:
 20 REM label;Z:
 30 REM var;a,1:
 40 REM var;E,0:
 50 REM var;b,15:
 60 REM rnd;d,1,0:
 70 REM bvar;S,0:
 80 REM var;x,20:
 90 REM var;y,15:
100 REM var;T,1:
110 REM bdr,0:
120 REM chg,0:
130 REM put;69,0,11,ATTACKED!
140 REM put;71,5,0,Defend your planet from the approaching
        flying saucers "A"
150 REM put;71,8,0,If 5 land of they hit your base the game
        will end
160 REM put;71,11,0,Move left with key 5 and right with key 8.
        Fire your laser with key 7.
170 REM put;71,16,0,Press BREAK to return to BASIC
```

```
180 REM put;196,21,1,PRESS ENTER TO START THE GAME
190 REM label;m:
200 REM call;J:
210 REM test;202,k,0,m:
220 REM chg,0:
230 REM put;71,x,y,HI
240 REM num;71,S,0,17:
250 REM put;71,0,11,SCORE
260 REM put;69,21,0:
270 REM routine;
280 REM label;A:
290 REM var;T,1:
300 REM test;204,b,31,B:
310 REM test;204,b,0,C:
320 REM test;204,d,1,D:
330 REM test;204,d,0,E:
340 REM call;F:
350 REM end;
360 REM routine;
370 REM label;B:
380 REM var;d,0:
390 REM end;
400 REM routine;
410 REM label;C:
420 REM var;d,1:
430 REM end;
440 REM routine;
450 REM label;D:
460 REM rnd;e,4,0:
470 REM test;194,e,1,b:
480 REM put;0,a,b,          [one space]
490 REM inc;b,1:
500 REM put;4,a,b,A:
510 REM label;b:
520 REM end;
530 REM routine;
540 REM label;E:
550 REM rnd;e,4,0:
560 REM test;194,e,1,c:
570 REM put;0,a,b,          [one space]
580 REM dec;b,1:
590 REM put;4,a,b,A
600 REM label;c:
610 REM end;
620 REM routine;
630 REM label;F:
640 REM rnd;e,5,0:
650 REM test;194,e,1,H:
660 REM test;202,a,20,H:
```

```
 670 REM put;0,a,b,          [one space]
 680 REM inc;a,1:
 690 REM put;4,a,b,A:
 700 REM label;H:
 710 REM end;
 720 REM routine;
 730 REM label;J:
 740 REM var;k,0:
 750 REM get;k:
 760 REM end;
 770 REM routine;
 780 REM label;K:
 790 REM test;204,k,56,L:
 800 REM test;204,k,53,M:
 810 REM test;204,k,55,N:
 820 REM end;
 830 REM routine;
 840 REM label;L:
 850 REM test;202,y,30,0:
 860 REM put;0,x,y,          [two spaces]
 870 REM inc;y,1:
 880 REM put;71,x,y,HI:
 890 REM label;0:
 900 REM end;
 910 REM routine;
 920 REM label;M:
 930 REM test;202,y,0,P:
 940 REM put;0,x,y,          [two spaces]
 950 REM dec;y,1:
 960 REM put;71,x,y,HI:
 970 REM label;P:
 980 REM end;
 990 REM routine;
1000 REM label;N:
1010 REM inc;y,1:
1020 REM var;f,19:
1030 REM label;0:
1040 REM attr;A,f,y:
1050 REM test;202,A,4,R:
1060 REM put;69,f,y,1
1070 REM sound;f,1:
1080 REM put;0,f,y,         [one space]
1090 REM dec;f,1:
1000 REM test;194,f,0,0:
1110 REM jump;S:
1120 REM label;R:
1130 REM put;4,f,y,X
1140 REM halt;5:
1150 REM sound;31,10:
```

```
1160 REM sound 25,5:
1170 REM sound;31,10:
1180 REM put;7,f,y,X
1190 REM halt;5:
1200 REM put;0,f,y,          [one space]
1210 REM add;S,50:
1220 REM num;71,S,0,17:
1230 REM var;T,0:
1240 REM label;S:
1250 REM dec;y,1:
1260 REM end;
1270 REM label;G:
1280 REM call;A:
1290 REM test;202,a,20,I:
1300 REM call;J:
1310 REM test;202,k,32,z:
1320 REM call;K:
1330 REM test;202,T,0,T:
1340 REM jump;0:
1350 REM label;T:
1360 REM var;a,1:
1370 REM jump;G:
1380 REM label;1:
1390 REM attr;A,x,y:
1400 REM test;202,A,4,Y:
1410 REM inc;y,1:
1420 REM attr;R,x,y:
1430 REM test;202,A,4,Y:
1440 REM dec;y,1:
1450 REM put;0,a,b,          [one space]
1460 REM var;a,1:
1470 REM inc;E,1:
1480 REM test;194,E,5,G:
1490 REM label;Y:
1500 REM var;c,0:
1510 REM label;U:
1520 REM var;B,1:
1530 REM label;V:
1540 REM bdr;B:
1550 REM sound;c,1:
1560 REM inc;B,1:
1570 REM test;194,B,8,V:
1580 REM inc;c,1:
1590 REM test;194,c,20,U:
1600 REM bdr;0:
1610 REM chg;0:
1620 REM put;198,10,11,GAME OVER
1630 REM put;240,9,10,     [ten spaces]
1640 REM put;240,11,10,    [ten spaces]
```

```
1650 REM put;240,10,10,    [one space]
1660 REM put;240,10,20,    [one space]
1670 REM var;a,0:
1680 REM label;M:
1690 REM sound;a,1:
1700 REM inc;a,1:
1710 REM test;194,a,150,M:
1720 REM chg;0:
1730 REM put;71,5,9,YOUR SCOPE
1740 REM num;71,S,5,20:
1750 REM put;240,15,3,PRESS ENTER TO PLAY AGAIN!
1760 REM label;a:
1770 REM call;J:
1780 REM test;202,k,13,Z:
1790 REM test;194,k,32,a:
1800 REM label;z:
1810 REM exit;
```

When and if you type in this program, please remember that the instructions contained within square brackets, are for some reference only, and should not be typed in.
Obviously you could change the graphics if you wished to suit your own personal taste.
But I think you will find a great deal of useful information and insight into writing a program with SCOPE from this example.

Using Scope from BASIC and with other Machine Code Routines

SCOPE can be used as a complete language for writing finished games and other programs. There may be occasions however, when it is preferred to call SCOPE routines from within a BASIC program. Also you may wish during the course of a SCOPE program to utilise machine code routines you have written yourself.

TO CALL A SCOPE ROUTINE FROM BASIC

First write and complete your SCOPE routine making a note of where it has been compiled. You can then call the machine code routine that SCOPE has written by means of a USR call.

You may wish to transfer BASIC variables into SCOPE variables. In the appendices you will find the memory addresses of all SCOPE variables and double variables. From BASIC you simply poke your BASIC variable values into the appropriate SCOPE variable addresses, and then call your SCOPE routine.

TO CALL A MACHINE CODE ROUTINE FROM SCOPE

In the appendices there is a list of the memory addresses used for labels. Choose a label that you wish to designate for your machine code routine and poke the address of your machine code routine into the table for that label. The values should be poked low byte/high byte.

Then whenever you CALL that label (NOT JUMP) you will be calling your machine code routine.

APPENDICES

LABEL TABLE	60162 – 60393 In the order:-	A – Z a – z #A – #Z #a – #z
VARIABLE TABLE	60104 – 60161 In the order:-	A – Z a – z
BVAR TABLE	60394 – 60445	A – Z ONLY
N.A.P.	60482 – 60483	
R.A.P.	60484 – 60485	

NOTES

SCOPE USER'S CLUB

Application for Membership

By simply completing this sheet and sending it to us you will be able to benefit from the unique **Scope User's Club.** The aim of the club is to provide a forum for users and also a definitive source of information and assistance for the serious young programmer.

On receipt of your application form we will send you a membership package which will include your personalised membership card upon which will be the programmer's telephone hot-line number.

Additionally, we hope to be producing a regular news-sheet to keep you informed of developments and details of special offers.

Complete this page, tear it out and send it to us to join a really unique club.

APPLICANT NAME _____

ADDRESS _____

_____

_____

PHONE _____ AGE _____

COMPUTERS OWNED _____

WHERE DID YOU FIRST LEARN OF SCOPE? *Please tick as applicable*

☐ FRIEND ☐ RETAILER ☐ PUBLICATION

IF PUBLICATION, PLEASE STATE WHICH ONE _____

WHAT COMPUTER PUBLICATIONS DO YOU READ REGULARLY?

_____

_____

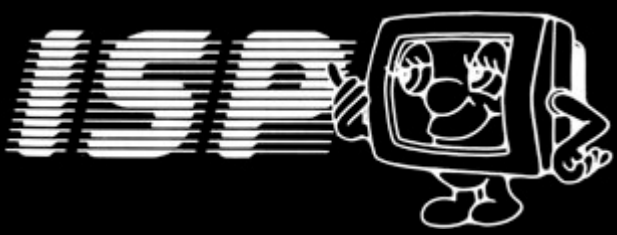
WHERE DID YOU BUY YOUR COPY OF SCOPE? *Please tick as applicable*

☐ MAILORDER ☐ RETAILER

NAME OF RETAILER _____

_____

Please post your completed form to:
ISP Marketing Ltd., 15a Castons Yard, Basingstoke, Hants.


ISP