SPRITES COMPILER

INSTRUCTION MANUAL

GLENCO SOFTWARE

SPRITES COMPILER PROGRAM

WHAT IS A COMPILER

All computers work in Machine Code. They do not understand High-Level languages such as Basic. When a computer is instructed to RUN a program that is written in Basic, it has to convert every command within your Basic program into machine code so that it can then process your program.

Inside your computer there is a program to convert a Basic command into machine code. This program is called an Interpreter. An interpreter will only work on one instruction at a time. Once the command has been converted into machine code and processed, the machine code for that command is discarded. If the computer has to process that command again, it again has to convert it into machine code.

As an example, consider a complex instruction manual written in a foreign language. You need an interpreter to convert from the language in which the book is written, into your own language. As the interpreter reads you the instructions, you carry out the tasks, one instruction at a time. If you need to repeat an instruction, the interpreter will need to tell you the instruction again.

It would be much simpler if the instruction manual was written in the language you already understood. This would significantly speed up the particular job you were being instructed to do . This would mean the whole of the manual had been compiled into your own language even before you started to read.

This is how a computer compiler works !

ALL of your Basic program is converted into machine code before being executed by the computer, this cuts out the 'middle-man', the interpreter, and so vastly increases the speed at which your programs execute. Your program will then run up to 16 times faster than an equivalent interpreted Basic program.

The Sprites Alive Compiler is a full compiler, as opposed to a semi-compiler, ALL of the Sprites Alive Basic instructions are converted into machine code.

The Compiled machine code or 'object code' will run anything up to 1600% faster than Basic, 16 times faster than Basic.

The Basic has also been extended by 70 new graphics and sound commands, these extra commands are described in detail in the Sprites Alive instruction manual.

THE SPRITES ALIVE COMPILER

The language used within the compiler is a subset of the Basic supplied with your computer. Some of the commands are identical to Basic commands, some of the commands are similar but may operate in a slightly different manner, and some of the commands are totally new.

All maths routines use integer numbers, no decimal points are allowed.

Line numbers are ignored, all GOTO and GOSUB commands use labels as reference points.

No string variables are allowed.

OVERALL VIEW OF THE SPRITES ALIVE COMPILER

There are 5 steps that need to be taken in order to convert your Basic program into an independent machine code program.

1) Write your Basic program using either a word processor or by entering the program as you would enter a standard Basic program.

2) Save the program to disc as an ASCII file.

3) Save any graphic or node files, your program uses,to disc.

4) Run the Compiler program.

5) Your source program will be loaded in from disc, compiled and then saved back to the disc, ready for you to RUN.

The above process may seem a bit involved, however the results are worth waiting for.

WRITING A PROGRAM

The way the commands are used within the Basic version of Sprites Alive and the compiler version differ slightly.

1) The compiler version of the commands do not require the '¦' symbol before every Sprites command.

2) A comma should NOT be placed directly after the command and before the first variable, you should instead, use a space.

3) The '@' symbol should never be used. Where this symbol occurs within your Basic program, simply remove it.

4) The '%' symbol should never be used. Where this symbol occurs within your Basic program, simply remove it.

5) Due to the fact that all sprite commands now do not require the '¦' symbol before them causes some of the command names to clash. ie Basic's MOVE and SPRITE's MOVE. Details are given on page 21 of the manual on how to overcome this problem.

There are two methods you may use to write a program that may then be compiled. The program may be written as you would write a normal Basic program, or you may use a word processor such as Arnors Protext.

USING BASIC

Enter a program as you would write a normal Basic program. However you should only use the commands as specified in this, and the Sprites Alive manual.eg

```
10 .START:LOOP A,1,10:PRINT A:ENDLOOP
```

The line number is entered first followed by the instructions for that line. Don't worry if you do not understand the instructions, you will soon learn.

Line numbers are not recognised by the compiler. The line numbers are only needed to inform the computer where to place that particular line within the program.

If you type the program into the computer using lower case letters, you may find on listing the program that some of the commands have been converted to UPPER case letters. The upper case letters are the commands that the Basic interpreter recognises. This will not affect the compilation process.

If you find that this effect makes your listing look untidy there is a way to get around the problem. You can insert the character ' after the line number and before the command. This will not effect the compilation process.

Once you have entered your program, you will now need to save it. This is achieved by using a slight variation of the Basic save command.

```
SAVE "DEMO1.SRC",A
```

This is an example of how to save your program. There are two things to note about the command.

Firstly is the file extension .SRC, the compiler program will only accept a program if it has the .SRC extension. This denotes your program to be the SOURCE program. If you fail to use the .SRC filename extension the compiler will not compile your program.

The compiler can only compile an ASCII file. The ,A command extension tells the computer to save your program as an ASCII file. If you instruct the compiler to compile a program that is not saved in the ASCII format an error message will be displayed.

SPRITES COMPILER PROGRAM

When you do compile your program and an error is found, the compiler will report the line in which the error occurred. This may not be the line number you have used within your program. The compiler line numbering system starts at 1 and increases by 1 for every new line it compiles. To simplify matters please see the following example.

The compiler is trying to compile the following program.

```
10 .START:LOOP A,1,10
15 B=A*5:C=B/2:PRINT C
20 .START:ENDLOOP
```

There is an error in line 20 of the program, 'LABEL ALREADY DEFINED'. The compiler would display the error as being in line 3, the third line it tried to compile. This may cause confusion when you are trying to track down the erroneous line. To solve this problem we suggest that just before saving your program you issue the command.

```
RENUM 1,1,1
```

This command will alter all the line numbers within your program to match the line number the compiler will display. If you wish to insert any new lines into your program you should then use the command.

```
RENUM
```

This will allow you to insert new lines, without overwriting any existing program lines. Once you have inserted the new lines you may then use the first RENUM command prior to saving your program.

USING A WORD PROCESSOR

Your word processor will need to be able to save files in ASCII format, see your word processor instruction manual on how to do this.

You should not include any control code sequences within your program.

You have a free hand on how you set the layout of your program. You may include as many spaces as you like between commands. You may include blank lines between different sections of program.

You do not need to start a line with a line number, the line count on the word processor will be sufficient.

When you save your program please use the file extension '.SRC'. This informs the compiler that your program is written in Source code.

COMPILING YOUR PROGRAM

Having saved your source file to disc, you may now compile it. Reset the computer and load the compiler program into memory. After a slight delay the following screen will be displayed.

```
SOURCE FILE NAME :

DRAWING FILE NAME :

NODES FILE NAME :

OBJECT FILE NAME :

LINE NUMBERS (Y/N) :
```

This screen is requesting the names of the various files which will be merged together in order to form a complete working program.

If you make a mistake when typing in any of the above information, you may use the DELETE key to backspace over the mistake. If you have made a mistake and you have pressed the RETURN key, pressing the ESC key will clear the screen and allow you to enter all of the information again.

When you enter the filename as requested, you should not include the file extension, this will be added automatically.

Source files have the extension .SRC, Drawing files have the extension .DRW, and Nodes files have the extension .NDE.

The object file name will be the name that the finished program will be called.

If you select 'Y' to the LINE NUMBER (Y/N) option, your final compiled program will contain machine code to detect whenever a new line is being processed. If an error occurs during running your compiled program, the line number the error occurred in will be reported.

During program development you should answer 'Y' to this question, this will allow you to track down errors much more quickly. The disadvantage to answering 'Y' to the option is that your compiled program will be longer and will execute slightly slower. Once all errors have been sorted out then answer 'N' to this question.

If you are using a two drive system you may specify which drive to use by making the first letter of the filename the drive letter A or B followed by a colon, followed by the filename.

ie B:WIPEOUT

Once you have answered the line number question, you will be asked to

INSERT THE DISC CONTAINING SOURCE CODE

After inserting the correct disc you should press any key. The compiler takes two passes to fully compile your program. The line number of the line currently being compiled is displayed.

If any errors have occurred within your source program, the compiler will stop and display an error message. The error messages are self explanatory. Please see page 24 of this manual for a further explanation.

There are two types of errors that may occur during the process of compilation. The error types are 'Fatal Error' and 'Compiler Error'.

FATAL ERROR

A fatal error is caused by something not allowing the compiler to continue, ie OUT OF MEMORY. The compilation process will stop.You will need to modify your source code and rerun the compiler program again.

COMPILER ERROR

A compiler error is caused by something that will not allow the compiled program to work correctly once it is fully compiled, ie INCORRECT NUMBER OF VARIABLES. Once an error has been reported the user has the choice to continue to compile the program or to stop the compilation process. If an error has occurred during pass 1, and you have selected the compilation process to continue, the compiler will stop at the end of pass 1 and allow you to make the relevant changes to your program. Please note that some errors may lead to other errors.

Example.

```
10 .START:LOOP A,1,1000,
20 PRINT A
30 ENDLOOP A
```

The above should print out all the numbers between 1 and 1000, however there is an error. On compiling the above program, the compiler will stop at line 10, reporting a SYNTAX ERROR (comma after the 1000). You then select the compiler to continue, and it will stop and report an error in line 30 (MISMATCHED LOOP - ENDLOOP). The LOOP statement in line 10 has been rejected because of a syntax error, therefore the LOOP has not been remembered by the compiler. When the compiler tries to match the ENDLOOP command with a LOOP command, it fails. An error has led to another error.

If no errors have been reported by the end of pass 2, the next stage in the compilation process takes place.

Information about the, now, compiled program is displayed.

```
VARIABLE START  : 02816
PROGRAM START   : 03130
PROGRAM EXECUTE : 05333
PROGRAM FINISH  : 09177
DRAWING DATA    : 16753

VARIABLE LENGTH : 00314
TEXT LENGTH     : 00028
DATA LENGTH     : 02175
PROGRAM LENGTH  : 06047
```

This is an example of how a typical information screen may look. There is no great need for you to remember this information, all of the relevant information will be stored in the LOADER program. The DRAWING DATA section will be blank at present.

The information above informs the user of how the memory is allocated within your program.

VARIABLE START is the location in memory where your variables defined within your Basic program are stored. The area immediately below the variable memory is used by the disc drive as a buffer area.

PROGRAM START is the starting position of your program, this may not be the program execute address. The area immediately above the program start, is the text area. All the text used within your Basic program is stored from PROGRAM START onwards. The area directly after the text storage area is the data area, any DATA you defined within your Basic program is stored directly after the text.

PROGRAM EXECUTE is the address from which your compiled program runs.

PROGRAM FINISH is the address of the end of your compiled program. The area between PROGRAM EXECUTE and PROGRAM FINISH is pure machine code.

DRAWING DATA is the address of the start location of your Drawings, Nodes and Sprites machine code. This variable will be blank.

VARIABLE LENGTH is the amount of memory used by variables within your program, each variable takes two bytes of memory.

TEXT LENGTH is the amount of memory the text uses within your Basic program.

DATA LENGTH is the amount of memory any data defined within your Basic program uses.

PROGRAM LENGTH is the total length of your program.

SPRITES COMPILER PROGRAM

After displaying the program information, the computer will prompt:

INSERT DISC TO SAVE THE COMPILED CODE

Insert your disc and press any key. If you typed in a filename for a drawing file, you will now be asked:

INSERT DISC CONTAINING DRAWING DATA

If you typed in a filename for a node file, you will now be prompted:

INSERT DISC CONTAINING NODE DATA

Insert your disc and press a key. The information for the DRAWING DATA will now be displayed.

IMPORTANT NOTE : When the DRAWING DATA number is displayed, you must ensure that the DRAWING DATA figure is higher than the PROGRAM FINISH variable. If the DRAWING DATA figure is NOT higher the compiled program will NOT work.

You may use the area of memory between PROGRAM FINISH and DRAWING DATA for your own purposes. The compiler uses the area between 768 and 2815 as a disc buffer. Please do not use this area unless you require a disc buffer within your programs.

Finally you may be again asked:

INSERT DISC TO SAVE THE COMPILED CODE

Once you have done this you should press any key.

Your new compiled program will now be saved to disc. To get it to execute, RESET the computer, insert the disc on which you saved the compiled program and type RUN " followed by the filename you specified when asked to enter the OBJECT filename.

The compiler creates three files during the compilation process.

"username. " - Basic loader program
"username.sa1" - Your compiled program
"username.sa2" - Sprites Alive, Drawings and Nodes

VARIABLES

You may use up to 286 variables within your programs, these consist of all the letters A - Z, and all the letters A - Z followed by a single number 0 - 9.

Example. All of these variables are different.

A , A0 , A1 , A2 , A3 , A4 , A5 , A8 , A9 , B , B0 , Z9

ARRAYS

You may use up to 26 different SINGLE dimension arrays. Array subscripts may be between 0 and 255.

To define an array, simply use the standard Basic DIM command. The array identifier can be any letter within the alphabet (A - Z).

```
DIM A(100),B(240),C(10)
```

You may access arrays, by using variables or, directly using numbers.

```
A(78)=R : A(U1)=234 : B(20)=3425
```

Arrays can only be used for storage purposes. You cannot use an array within any mathematical calculations or commands.

To make use of arrays, you will need to store the value from within the array in a standard variable.

Example: Set array A to contain the numbers 100 to 1, then repeat the loop and print the numbers.

```
LOOP I,1,100
B=101-I:A(I)=B
ENDLOOP
LOOP I,1,100
A=A(I):PRINT A
ENDLOOP
```

In the first stage, we have used variable B to store the mathematical answer before storing the result within the array. In the second stage we have retrieved the value from the array and stored it within variable A, before it has been printed.

Arrays can only be used as shown below:

```
A(I)=2143
A(T6)=P
P=A(I)
P=B(J1)
```

There is an exception to this rule. If you know the position within the array you require ie A(54), P(67) then you may use this type of expression, as you would use any standard variable.

Example of using Arrays:

NOT ALLOWED	ALLOWED
A(I)=45*P	T=45*P:A(I)=T
P=A(Y1)+1	P=A(Y1):P=P+1
P(I)=568/P	S=568/P:P(I)=S
F(J)=F(I)*Y(I)	A=F(I):B=Y(I):C=A*B:F(J)=C

VARIABLE ASSIGNMENT

Variables are assigned in the same way as you would assign variables within Basic. You can, if you wish, use the LET command, although this is not necessary.

```
LET a=1 : b=200 : c5=230 : z9 = -2431
```

If you use the LET command, you must include a space after the command and before the first variable.

EXPRESSIONS

You may use the following mathematical expressions:

```
A=-B
A=B+2
A=B-2
A=B*2
A=B/2
A=B OR 2
A=B AND 2
A=B XOR 2
```

Each single expression is classed as one instruction. If you require to do a number of mathematical tasks to calculate a result, you must use a number of commands.

To express A=4*(R+B1)+5

Use R1=B1+R : R2=R1*4 : A=R2+5

The spaces are used for clarity only.

CLEARING VARIABLES

To set all variables back to zero, use the command CLEAR. This will reset all variables, including any loops you may have defined. Please use this command with care. The variables are cleared automatically when you RUN your program. You do not need to initialise any variables with the statement a=0:b=0 etc.

CONSTANTS

The Compiler allows you to use any integer number within the range -32768 to 32767.

There is one exception to this rule. If you are using commands that require you to use numbers in the range 0 - 65535, (POKE,PEEK,CALL) you can include the ! symbol before the number. You may not use this symbol in any mathematical commands.

SCREEN USAGE

The way the compiler prints text to the screen is different to the way it is printed using standard Basic. We have incorporated Turbo-Text machine code routines that will allow screen printing to be up to 4 times faster than normal.

The Sprites program will not operate correctly if the screen is scrolled, we have therefore incorporated within the Turbo-Text code a routine that will stop the screen scrolling. If you print text to the screen that would normally make the screen scroll, the screen will now clear and the text will continue to be printed from the top of the screen again.

Even taking into account the time it takes for the screen to clear, printing speed is greatly improved.

All control characters are printed. You will not be able to alter any of the screen attributes by printing the control characters.

All text handling commands use only one window, the complete screen. You may not define a window within the screen and you cannot use print streams.

When using different size screens using the SCREEN command (see Sprites Alive Manual) it was necessary to experiment with the LOCATE command in order to print a message or number in a certain section of the screen. You do not need to do this any more, the Turbo Text routines take into account the shape of the screen when you want something printed.

You may output a single character to the screen using the CHAR command.

You may output numbers to the screen using the PRINT command.

You may output text to the screen using the TEXT command.

As mentioned above, you may not use control codes within any text output to the screen. The exception to this rule is the carriage return and line feed control codes. You may send control codes 10 and 13 to set the cursor to a new line. You will need to use the CHAR command to do this.

There are no commands incorporated within the compiler to allow you to alter the paper number. The paper has to be set to zero in order to allow the sprites to work correctly. If you wish to change the colour of the paper you should use the INK 0,n command.

When using a different sized screen, using the SCREEN command, the computers own graphics commands will not work correctly.

SCREEN AND TEXT COMMANDS

BORDER i Set border colour.

CHAR a Output a single character (a) to screen in the range 0 to 127. Character 10 causes a line feed. Character 13 causes a carriage return.

CLS Clear the screen.

COLOUR Set inks to previously defined inks.

INK i,c Set ink (i) to colour (c).

INKBLACK Turn off all pens.

LOCATE x,y Position the text cursor at location x,y.

MODE m Set the screen to mode (m).

PEN p Select a pen (p) for displaying text.

PRINT n Print a number (n) onto the screen at the current text location. Do not use this for printing text. You may optionally include a semi-colon after the variable to stop the carriage return and line feed characters being sent.

TEXT "abcd" Print the text enclosed within the quotes to the screen. No string variables are allowed. You may optionally include a semi-colon after the last quote to stop the carriage return and line feed characters being sent.

The PRINT and TEXT commands cannot be mixed within a single statement. You need to separate all commands by using either a colon or by using a separate line.

ie

```
TEXT "YOU HAVE SCORED ";A              IS NOT ALLOWED.
TEXT "YOU HAVE SCORED ";:PRINT A       IS ALLOWED.
```

GRAPHICS COMMANDS

The graphics commands supported by the Sprites Alive compiler are identical to the computers own graphics commands.

DRAW x,y Draw a line onto the screen from the current graphics location to position x,y.

DRAWR x,y Draw a line relative to current graphics position.

FRAME Wait for frame flyback.

GMODE g Set the graphics mask mode. Each mode effects how any graphics, using the computers own graphics system, appear onto the screen.

g = 0 : Normal Write mode.
g = 1 : XOR Write mode.
g = 2 : AND Write mode.
g = 3 : OR Write mode.

GPEN p Set the current graphics pen.

MOVE x,y Move current graphic cursor position to position x,y.

MOVER x,y Move current graphic cursor position relative to coordinates x,y.

PLOT x,y Plot a point at position x,y.

PLOTR x,y Plot a point relative to current graphics position.

SPEEK x,y,a Find the pixel colour from the screen location x,y and store value in (a).

SPOKE x,y,a Plot a point on the screen at location (x,y). Much faster than PLOT. Does not move graphics cursor.

SYMBOL s,a,b,c,d,e,f,g,h

Define a new character (s). s may be between 0 and 127.

SOUND COMMANDS

The sound commands that are available through Basic are allowed within the compiler. The commands are:

SOUND a,b,c,d,e,f,g

This command is identical to the standard Basic sound command with one exception, all the parameters must be supplied. If you do not require a certain function within the sound command you should use the value 0.

a - Channel and rendezvous requirements.
b - Amplitude envelope to use.
c - Tone envelope to use.
d - Tone period.
e - Noise period.
f - Initial amplitude.
g - Duration or envelope repeat count.

The above is the Basic sound command. There is another sound command used within Sprites Alive Basic. The Sprites Alive sound command is used to link sounds to sprites. Both versions of the command may be used within the compiler. Please see page 38 of the Sprites Alive instruction manual for details of the sprites sound command.

ENV n,a1,b1,c1,a2,b2,c2,a3,b3,c3

This is identical to the Basic version of the ENV command. All three sections of the envelope must be defined, if you do not require three sections, please use the value 0.

n - Envelope number.
a - Step count.
b - Step size.
c - Pause time.

ENT n,a1,b1,c1,a2,b2,c2,a3,b3,c3

This is identical to the Basic version of the ENT command. All three sections of the envelope must be defined, if you do not require three sections, please use the value 0.

n - Envelope number.
a - Step count.
b - Step size.
c - Pause time.

JUMPING COMMANDS

LABELS

Source programs do not use line numbers as references throughout your program, instead it uses labels. A label consists of a full stop followed by up to 16 characters.

```
.PROGSTART          .GAME.END

.TRY-LEFT           .SHOOT.IT

.OPEN-DOOR          .HI-SCORE-ROUTINE
```

Labels allow source code to look very tidy and should make the program easier to write and debug.

A label can use any alphabetical character, up to 16 characters in length. The case (upper/lower) is not taken into account when the labels are stored.

```
goto start
GOTO START
```

These two instruction would both go to the same point within the program.

Labels must be treated as if they were any other command. ie they must be followed by a colon or they must be on a line by themselves.

GOTO

The GOTO command is identical to Basic's version of the command except that instead of jumping to a line number, we are jumping to a label. Please note that you should not use the '.' before the label name. This is only used when we are defining labels and not when we want to jump to them.

GOSUB - ENDSUB

This again is identical to the Basic's version of the command except that we are now jumping to labels instead of line numbers. Please note that the command to return from the subroutine is ENDSUB and not RETURN as with Basic.

Please ensure that you have an ENDSUB command at the end of your subroutine. You may nest up to 16 subroutines.

OTHER COMMANDS

CALL 230

This command can be used to call one of your own machine code subroutines. Your subroutines may exist in free memory as explained on page 22.

This command may need to call a routine that is located higher than 32767 in memory. Under normal circumstances trying to use numbers greater than 32767 would cause an overflow error. To get around this problem you should include an ! symbol immediately before the number:

```
call !50000
```

USING FOR - NEXT LOOPS

There is a command within the compiler that simulates the Basic FOR NEXT loop, it is simply called the LOOP command.

LOOP A,s,f,st

The A is the control variable as in FOR A=. The s is the start value, as in the value before the 'TO'. The f is the finish value, as in the value after the 'TO'. The st is the optional step value, if this is not included the default value will be 1

ENDLOOP A

This is the replacement for the Basic NEXT command. The control variable after the ENDLOOP statement is optional.

You may nest up to 8 loops.

CONDITION TESTING : IF

There are a number of commands built in to the compiler to test variables. You do this using the IF command.

We have extended the Basic IF,THEN,ELSE structure to be much more flexible. In Basic, all the statements on the line following the IF are executed until either an ELSE command or the end of the line is reached. Any statements on the next line of Basic program will be executed whether or not the IF condition is met.

The Sprites Alive IF statement is much more flexible than that. If a condition is met all the commands following the IF command are executed, even though they may be on separate lines, until an ENDIF command is executed. If a condition was not true, all the statements following the IF command are skipped until the ENDIF statement is reached.

The tests you may use in an IF statement are shown below.

```
IF A=B:
IF A>B:
IF A<B:
IF A<>B:
```

EXAMPLE:

```
.START
        INPUT A
        IF A<100:TEXT "VALUE LESS THAN 100":ENDIF
        IF A>100:TEXT "VALUE GREATER THAN 100":ENDIF
        IF A=100
        TEXT "VALUE EQUALS 100"
        1ENDIF
        IF A<>100:TEXT "VALUE DOES NOT EQUAL 100"
        ENDIF
        IF A<>0:GOTO START:ENDIF
```

This program will accept a number and store it in variable A, it will print the relevant information about the variable A's relation to 100. Providing the value of A is not 0 the program will repeat the process.

You may not nest any IF statements, doing so will cause an error message during compilation.

ENDIF A

This command is used to mark the end of an IF statement. Its use is described above.

PROGRAM INTERACTION

There are various commands that allow the user to enter data into the program.

INPUT A

This command is identical to the Basic's INPUT command. It will accept a numerical entry typed into the keyboard and store it in the specified variable. You may enter any number in the range -32768 to 32767.

INKEY 1,A

This command allows the user to check if a certain key has been pressed. This command will not stop and wait for you to press a key, if no key is available, program execution will continue. The first variable after the command is the required key number you wish to test, the second variable will contain a value depending on if the specific key tested for was pressed: 1 - key pressed, 0 - key not pressed.

Please note the key number required is not the Ascii value of the character. Key numbers can be found within your Computer manual.

WAITKEY A

This command will pause the computer until a key is pressed. The Ascii value of the key pressed will be stored in the variable after the command.

FLUSH

This command will clear the keyboard buffer of any keypresses that have yet to be reported.

RND A

This command will return a random number in the range 0 to 255 in the variable specified after the command. The random number generator used within the compiler produces a very large even distribution of numbers.

OTHER BASIC COMMANDS SUPPORTED

PEEK 1234,A

This command will store in the variable (a), the contents of the memory location specified within the command. If you require to use a number greater than 32767, you should use the ! character immediately before the number.

POKE 1234,a

This command will store in memory location 1234 the contents of variable a. Variable a should contain a number between 0 to 255. If you require to POKE a memory location higher than 32767 you should use the ! symbol immediately before the number.

REM

You may use this command to insert comments into your source code. The comments will be ignored by the compiler.

DATA

This command is used in exactly the same way as the Basic DATA command. You may use it only to store integer numbers, you may not use String variables.

READ a

This command is used to retrieve the information that is stored in your DATA statements. You may only use one variable after the command name.

eg READ a1:READ C:READ D

RESTORE 20

This command is identical to the Basic RESTORE command, with one exception. Instead of restoring the data pointer to a particular line number, you restore the pointer to a particular piece of data.

ie RESTORE 12 - Will restore the data pointer to the 12th piece of data.

RESTORE 1 - Will restore the data pointer to the first piece of data.

NEVER USE RESTORE 0

Every item of data you define within your Basic program takes up two bytes of memory when compiled into machine code. If you are going to use a large amount of data, you will use up a great deal of memory.

The READ - DATA combination is a good way of storing data about sprites for various levels within a game. If you have a large number of levels you may run out of memory.

We therefore have included a command call DATALOW, this command will allow you to use data in the range 0 - 255 instead of -32768 to 32767. The advantage of using this command is that each piece of data used in your Basic program will only take up ONE byte of memory in the compiled program. This will allow you to enter twice as much data as was previously possible.

DATALOW

This command can be used anywhere within your program, even after all the DATA statements. The effect this command has, is to allow your DATA to use only half the memory it would use if this command had been omitted. ALL of your DATA has to be in the range 0 to 255, you should not use negative numbers. No checks are made on your data. If you have entered any data with a number outside of the range 0 - 255, no error messages will be displayed, however the particular piece of data that is out of range will be corrupt.

MISCELLANEOUS COMMANDS

LOAD "filename",2341

This command will load a binary file into memory. You should ensure you will not overwrite any of the compiled program when using this command. If you require to load a file into memory whose location is greater that 32767, you should include the ! symbol immediately before the number.

STOP

This command will stop the compiled program and return control back to Basic. You may use this command anywhere within your compiled program, even in the middle of executing a subroutine.

TIME W

This command will return a value of the time elapsed since the computer was turned on. The value is returned in the variable specified after the command. The number returned is a count in non specific time units, the timer increments every .85 of a second.

SPRITES ALIVE COMMANDS SUPPORTED

All Sprites Alive commands are supported with the exception of the MEMORY, DRAW and NODE commands.

Certain commands have had their names altered due to the fact that they clash with BASIC commands. These commands are:

BEFORE	AFTER
MOVE	SMOVE
MOVEALL	SMOVEALL
MOVEHIT	SMOVEHIT
SWINDOW	WINDOW

Simply add an 'S' to all commands that start with the word MOVE. SWINDOW has now become window.

If you are using Nodes within your program, you may not use the DGET command.

ERROR DETECTION

If an error occurs during the process of executing your compiled program, an error line will be displayed. This will look like this:

ERROR: COM 23 TYPE 12 LINE 15

This message means that an error has occurred in command 23, error type 12 in line 15. If you did not specify line numbers to be included in your compiled code, the line number will, for obvious reasons, not be printed.

The commands and their associated numbers can be found on page 27 of this manual. The error type numbers can be referenced to your Sprites Alive manual. An extension to the error types can be found below

ERROR	DESCRIPTION
100	Overflow.
101	No matching gosub found.
102	Too many gosubs.
103	Number greater than 127.
104	Division by zero.
105	Ran out of data.

GENERAL INFORMATION

MEMORY MAP OF YOUR COMPILED PROGRAM

The compiler program uses memory from location 768 upwards. The Basic loader program that the compiler creates is the largest Basic program that will fit into memory. Do not try to modify the loader program in any way. Altering the Basic loader program may corrupt your compiled program.

Disc Buffer	&300
Variable Start	&B00
Program Start	Variable Start + No of Variables * 2
Text Storage Area	Program Start
Data Storage Area	Text Area + Amount of Text
Program Execute	Data Area + Amount of Data
Program Finish	Program Execute + Length of M/C
Sprites Alive End	&A670
Sprites Alive Start	&525E
*Drawing Info	&525E - Memory for drawings
*Node Info	Drawing Info - Memory for nodes
*DRAWING DATA	Node Info

* Working backwards in memory

The Compiler works its way up in memory, the Drawings and Nodes work their way down in memory. You may use the memory between PROGRAM FINISH and DRAWING DATA for your own purposes.

THE SPEED FACTOR

If your source programs use a large number of Sprites Alive commands with very little Basic involved in linking them together you may find they do not execute much more quickly. This is due to the fact that the Sprites Alive graphics routines are the fastest they could possibly be. It is Basic that slows your programs down, if you do not have many Basic commands you will not notice a great speed increase. If you use a large number of Basic commands, well .....

HOW THE COMPILER PROGRAM WORKS

OR

HOW TO PASS 5 MINUTES TRYING TO UNDERSTAND THIS DESCRIPTION

When you RUN your compiler program a complex process is started. The loader program loads the Compiler into memory at &5800, the banked RAM is turned on and the Sprites Alive run time code is loaded into the very top of the banked RAM.

After setting all of the system variables and flags, you are asked to enter the filenames of your Source, Drawing and Node files.

Pass 1 now begins. Your source program is read into the computer, one character at a time until a complete line has been entered. Your line is scanned to check for labels, instructions, maths, etc. If the line contains a label the label name is stored at the highest area of free memory available (below &5800).

If the line contains instructions, the instruction is decoded and checked to ensure correct syntax, number of variables etc. When an instruction is decoded, a lexical code is stored in memory starting from &900. A lexical code is a single byte of information informing the compiler of the instruction type.

Once the complete line is decoded, the next line is loaded into memory and the process repeats itself.

When the complete program has been converted into lexical code, the checks at the end of pass 1 begin. The checks are to ensure you do not have any undefined labels, mismatched loops, etc.

If your program passes all of the tests, we start on pass 2.

The compiler starts to convert the lexical code into machine code. As the lexical code is converted it is stored in the banked RAM. When all of the lexical code is completely converted into machine code, the lexical code is erased. The machine code is copied back from the banked RAM into normal memory. You are now prompted to insert a disc, and the machine code is saved with the file extension .SA1.

The compiler will then erase the machine code from its memory and copy the Sprites Alive run time module from the top of the banked RAM. You are prompted to insert your disc containing the Drawing and Node files. The information from these files is merged with the run time module and the complete module Drawings, Nodes and run time module is saved to disc with the extension .SA2.

COMPILER ERRORS

During the compilation process a number of errors may be detected within your source program. The error messages that the compiler may display are self explanatory. Below is a brief description of any errors that may occur.

IMPROPER LABEL NAME

The first character of the label must be a letter (A - Z).

LABEL TO LONG

The label name is greater than 16 characters.

SYNTAX ERROR

The syntax of the command is not correct.

TOO MANY LABELS DEFINED

More than 255 labels have been defined. The maximum number of labels is 255.

LABEL MEMORY EXHAUSTED

The labels have taken up to much memory. Try to shorten the label names.

LABEL ALREADY DEFINED

You have tried to define a label which has already been defined.

LINE TO LONG

A single line of source code is greater than 255 characters.

COMMAND TOO LONG

A command is greater than 127 characters. If this occurs within a DATA statement, split the statement into smaller sections.

QUOTES LEFT OPEN

You have not included both quotes within a command.

UNKNOWN COMMAND

The compiler does not recognise the command.

ARRAY ALREADY DEFINED

An array has tried to be defined more than once.

ARRAY NOT DEFINED

An array has attempted to be accessed without first being defined.

ARRAY DIMENSIONS INCORRECT

An array has attempted to be defined with a subscript larger than 255.

NUMBER OVERFLOW

A number has been encountered that is outside the range 32767 to -32768.

ARRAY OVERFLOW

An access to an array is outside the defined subscript maximum.

IF STATEMENT ALREADY OPEN

An IF statement has occurred within your program without the last IF statement being closed with an ENDIF.

NO IF STATEMENT DEFINED

An ENDIF statement has occurred without a matching If statement.

TOO MANY IF'S IN PROGRAM

More than 255 IF statements have occurred within your program.

TEXT COMMAND ERROR

A TEXT statement has been defined without a quote character being included.

LOAD COMMAND ERROR

A LOAD statement has been defined without a quote character being included.

LOOP ALREADY DEFINED

A loop has attempted to be defined whilst a loop with the same control variable is still open.

MAXIMUM NESTED LOOPS 8

An attempt has been made to nest more than 8 LOOP statements.

MISMATCHED LOOP - ENDLOOP

An attempt has been made to close a LOOP which is not the current loop.

NO LOOP MATCHED

An ENDLOOP has occurred without a matching LOOP being open.

INCORRECT NUMBER OF VARIABLES

The number of variables after the command is not correct.

TOTAL NUMBER OF LOOPS EXCEEDED

You have attempted to use more than 255 loops within your program.

IF STATEMENT LEFT OPEN

You have not closed an IF statement within your program.

LOOP STATEMENT LEFT OPEN

You have not closed a LOOP statement within your program.

UNDEFINED LABEL

You have tried to jump to a label that has not been defined.

UNSIGNED NUMBER CANNOT BE NEGATIVE

An attempt has been made to set an unsigned number to a negative value.

UNSIGNED NUMBERS NOT ALLOWED IN MATHS

Unsigned numbers can not be used within mathematical expressions.

EXPRESSION ERROR

An expression has been defined that attempts to evaluate more than 2 variables.

ARRAYS ILLEGAL WITHIN STATEMENTS

Arrays may not be used within expressions.

PASS 1 CODE, RAN OUT OF MEMORY

Your program is too long.

PASS 2 CODE, RAN OUT OF MEMORY

Your program is too long.

DATA STATEMENT NOT NUMERIC

Data statements can only contain numbers.

COMMAND ERROR NUMBERS

ERR	COMMAND	TYPE	PAGE
1	ANIMATE	SPRITES	29
2	ANIMOFF	SPRITES	29
3	ARRAY	BASIC	9
4	BORDER	BASIC	12
5	BULLET	SPRITES	33
6	CALL	BASIC	16
7	CHAR	BASIC	12
8	CLEAR	BASIC	10
9	CLEARREP	SPRITES	24
10	CLS	BASIC	12
11	COLLIDE	SPRITES	22
12	COLLTEST	SPRITES	23
13	COLOUR	SPRITES	12
14	CSPRITE	SPRITES	27
15	DGET	SPRITES	9
16	DIM	BASIC	9
17	DIVISION	BASIC	10
18	DRAW	BASIC	13
19	DRAWR	BASIC	13
20	ENDSUB	BASIC	15
21	ENDIF	BASIC	17
22	ENDLOOP	BASIC	16
23	ENT	BASIC	14
24	ENV	BASIC	14
25	ERASE	SPRITES	15
26	EXPLODE	SPRITES	36
27	FLUSH	BASIC	18
28	FRAME	SPRITES	37
29	GMODE	BASIC	13
30	GOSUB	BASIC	15
31	GOTO	BASIC	15
32	GPEN	BASIC	13
33	HIT	SPRITES	25
34	IF	BASIC	16
35	INK	BASIC	12
36	INKBLACK	SPRITES	12
37	INKEY	BASIC	17
38	INPUT	BASIC	17
39	KEB	SPRITES	17
40	KEBDEF	SPRITES	19
41	KEBSPEED	SPRITES	19
42	LET	BASIC	10
43	LOAD	BASIC	20
44	LOCATE	BASIC	12
45	LOOP	BASIC	16
46	MAZEOFF	SPRITES	43
47	MAZEON	SPRITES	43
48	MERGE	SPRITES	35
49	MISSDELAY	SPRITES	32
50	MISSDIST	SPRITES	32

51	MISSHIT	SPRITES	34
52	MISSILE	SPRITES	31
53	MISSTYPE	SPRITES	30
54	MODE	BASIC	12
55	MOVE	BASIC	13
56	MOVER	BASIC	13
57	MULTIPLY	BASIC	10
58	NEXTREP	SPRITES	24
59	NODEALTER	SPRITES	42
60	NODEATTR	SPRITES	41
61	NODEOFF	SPRITES	42
62	NODEON	SPRITES	42
63	NODESPEED	SPRITES	41
64	NODESPRITE	SPRITES	42
65	PEEK	BASIC	18
66	PEN	BASIC	12
67	PLOT	BASIC	13
68	PLOTR	BASIC	13
69	POKE	BASIC	18
70	PRINT	BASIC	12
71	READKEB	SPRITES	20
72	READSTIX	SPRITES	20
73	REM	BASIC	18
74	REPOFF	SPRITES	23
75	REPON	SPRITES	22
76	REPORT	SPRITES	24
77	RESET	SPRITES	15
78	RND	BASIC	18
79	SATTR	SPRITES	38
80	SCENERY	SPRITES	36
81	SCREEN	SPRITES	9
82	SDIR	SPRITES	17
83	SDRAW	SPRITES	27
84	SEQUENCE	SPRITES	28
85	SGET	SPRITES	14
86	SHOOT	SPRITES	34
87	SMOVE	SPRITES	21
88	SMOVEALL	SPRITES	21
89	SMOVEHIT	SPRITES	25
90	SOUND	SPRITES/BASIC	38/14
91	SPEEK	SPRITES	37
92	SPOKE	SPRITES	37
93	SPUT	SPRITES	15
94	SPUTALL	SPRITES	15
95	STIX	SPRITES	17
96	STIXSPEED	SPRITES	19
97	STOP	BASIC	20
98	STUCK	SPRITES	37
99	SXPOS	SPRITES	27
100	SYMBOL	BASIC	13
101	SYPOS	SPRITES	27
102	TEXT	BASIC	12
103	TIME	BASIC	20
104	WAIT	SPRITES	27

105	WAITKEY	BASIC	18
106	WINDOW	SPRITES	21
107	WP	SPRITES	12
108	XDIR	SPRITES	27
109	XEDGE	SPRITES	16
110	YDIR	SPRITES	27
111	YEDGE	SPRITES	16
112	RESTORE	BASIC	19
113	READ	BASIC	19
114	DATALOW	BASIC	19