Get with the PROGRAM

FUZE BASIC Programmer's Reference Guide

Written by Jon Silvera
& Colin Bodley

FUZE BASIC

Programmer's Reference Guide

For new commands, features and projects visit;

www.fuze.co.uk

With many thanks and a great deal of respect to Gordon Henderson for his work on RTB where FUZE BASIC originated.

FUZE BASIC is developed by FUZE Technologies Ltd in the UK. The team consists of;

Jon Silvera - Project manager
Luke Mulcahy - Lead programmer
David Silvera - FUZE Product manager
Colin Bodley - Documentation
Additional information – www.fuze.co.uk

Manual Version: 1.4 Date: 25/11/2015 FUZE BASIC Version: 3.3

Introduction

FUZE Keyboard

The keyboard is a standard layout with that same editing and function keys as a standard PC

The following chapter is an introduction to programming in FUZE BASIC on a FUZE Powered by Raspberry Pi®. I have included background sections on how computers work to help with your understanding but if you want to just skip to the programming you can ignore the sections with a box around them.

Computer Programs

Computer programs are sets of instructions that tell a computer to carry out a particular task.

The programs are executed (or run) by the computer's Central Processing Unit (CPU) .

Computers run programs in a language called machine code.

This consists of a series of numbers which represent instructions that the machine understands and can carry out.

In fact, even worse than that, the numbers are represented in the machine as a series of 1s and 0s in something called binary. So to us a machine code program could look something like this: 00101010101010111110101010101010100101...

Programming Languages

Machine code is obviously difficult for us humans to read and understand. This is why we use programming languages to tell a computer what to do. Instructions in the programming language are converted into the 1s and 0s that the computer actually understands.

There are many different programming languages in existence. Over the years I have written programs in several: FORTRAN, PASCAL, C, C#, PHP and SQL, but the first language I learned to program in was a language called BASIC.

BASIC

The BASIC programming language has been around a couple of years longer than me (and I was born when England won the World Cup!). It stands for **B**eginner's **A**ll- purpose **S**ymbolic **I**nstruction **C**ode. It was designed to be easy to use for people who had not used computers before.

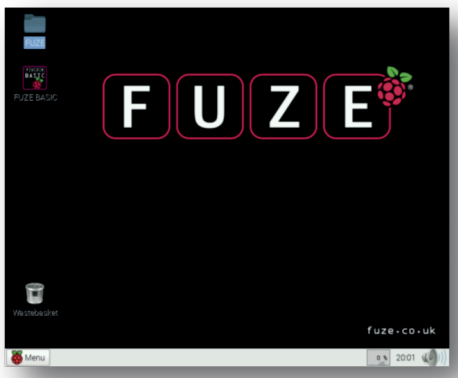
Here at FUZE Technologies Ltd. we believe it is still the best introduction to computer programming.

BASIC is (usually) known as an interpreted language. This means that each instruction is taken in turn and converted to the binary code before being executed. Other languages such as C are called compiled languages because the whole program is converted in one go before being run. Compiled languages are generally faster than interpreted languages because the instructions only have to be converted once. The advantage of an interpreted language is that you can see results more quickly as you are writing the program.

FUZE BASIC

FUZE BASIC is a modern evolution of the many variations of the classic BASIC programming language.

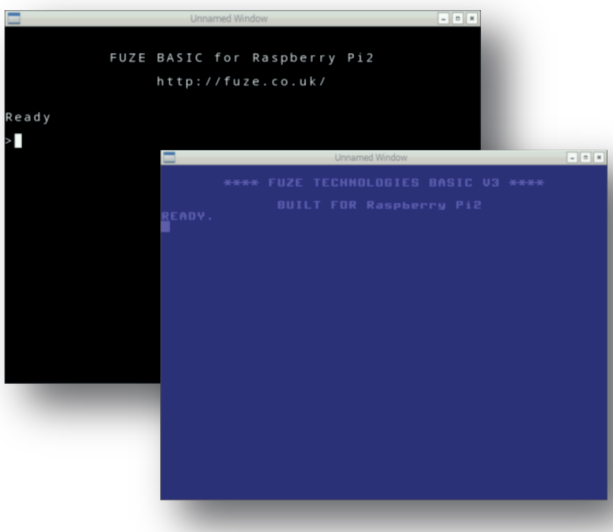
Follow the instructions in your Getting Started guide to set up and start your FUZE computer. Once the computer has finished 'booting' you should see the FUZE desktop:



Use your mouse to point to the FUZE BASIC icon in the top left of the screen which looks like this:

Using the left mouse button click twice quickly (this is called double clicking) on the FUZE BASIC icon in the top left to start FUZE BASIC.

You should now see a FUZE BASIC window:



FUZE BASIC can be configured with a number of different environment settings. When you see the “Ready” prompt, you are ready to start programming!

Before you start, try holding down Shift and tapping the Tab key. You can flick through a few retro styled interfaces.

Immediate Mode

You will notice underneath the banner on the left hand side the following:

This is the input cursor and it is telling you that it is ready for you to enter a command. The command will not be carried out until the **Enter** key is pressed. On the FUZE this key is towards the right hand side, to the left of the Page Down key and looks like this:

This is called **Direct or Immediate Mode** because the command will be carried out immediately. Try it now by typing something and pressing the **Enter** key. Unless you were extremely lucky (or you have done this before!) the chances are that you received an error message something like the on the next page:

```
Ready
>Hello World
*** Syntax error trying to parse: ... World
>
```

The syntax of the language describes the commands that can be executed, so a syntax error is saying that it did not recognise what you typed as being a valid command. Try it again but this time type the following and press the **Enter** key:

```
>PRINT "Hello World"
```

Strings of characters are marked using the speech mark character " which is on the same key as 2 on the keyboard and can be typed by holding down the **Shift** key and pressing the 2 key. There are two **Shift** keys on the keyboard, one on either side, that look like this:

This time you should see *Hello World* echoed back to you. This is because the **PRINT** command is part of FUZE BASIC and what it does is to print the text between the speech marks to the screen.

You have now executed your first FUZE BASIC command but you are yet to write your first program.

Computer Memory/Storage

Like a lot of other computers the FUZE has two different types of storage for programs (instructions) and data (information). The first is called volatile storage and lasts only as long as the computer is switched on. The second is called non-volatile and "remembers" information even when the device is disconnected from the power. Volatile storage tends to be faster but more expensive and lower capacity than non-volatile although the difference is reducing all the time.

In the case of the FUZE the volatile storage is on a microchip on the circuit board of the Raspberry Pi inside the FUZE case. The non-volatile storage is on the SD card inserted in the back of the case. Programs are stored on the non-volatile memory and then loaded into the volatile memory to be executed.

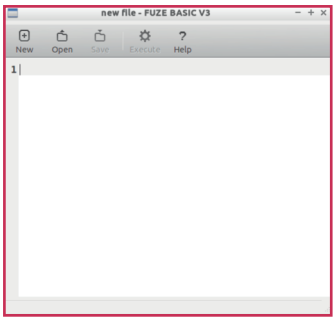
The Editor

In this mode a sequence of FUZE BASIC commands are stored in the computer's memory which can then be executed again and again.

To start the program editor type **EDIT** (and press enter) in immediate mode or press **F2.** Use **F2** to switch between the editor and immediate mode.

See image overleaf;

On the first line in the editor type the command:




```
PRINT "HELLO"
```

You will notice that when you press **Enter** the command is not executed but that the cursor moves down to the line below. Now add the **END** command to the second line (without a line number). Your program should now look like this:

```
PRINT "HELLO"
END
```

RUN

You can execute this program using **RUN**. In the editor press **F3** or click the **EXECUTE** icon. The first time, and every first time with a new program, the editor will ask you to save your program. Save the file in your preferred location and filename then when back to the editor press **F3** again to **RUN** it.

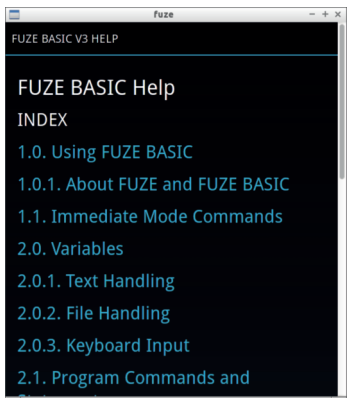
As the first line of the program is executed and carries out the **PRINT** command. This prints ***Hello World*** to the screen. The program runs until a **STOP** or **END** command is reached. If there is no **STOP** or **END** command found you will receive an error like the one below.

Congratulations! You have just written your first (working) FUZE BASIC program. Admittedly it doesn't do very much but this is traditionally the first program that everyone writes when they are learning a new programming language. Now change your program to look like the following;

```
LOOP
PRINT "HELLO"
REPEAT
```

RUN your program from within the editor by pressing the **F3** key or clicking the **EXECUTE** icon**.** The Function keys are along the top of the keyboard and numbered from **F1** to **F12**.

Your program should still be running. Press the **ESC** key to stop it and **F2** to return to the editor.



The Function keys usually have different functions depending on the program running but normally **F1** is used to bring up help information for the current program. Try it now and you will get help pages for the editor:

To exit the Help window, click the close window Icon (X).

Your program is currently only residing in the computer's memory. If the power were to go off all of your hard work would be lost forever. The editor will prompt you to save when you first RUN a program. Thereafter it will save it automatically just before it executes it. This is great as even if you crash a program it will have save the very last change you entered.

When you enter a name for your program. You should choose one that will remind you what it does. In this case you could call it **Hello World**. When you press Enter the program is saved to a file on the card and the program is run, printing **Hello World** to the screen again and then ending.

At this point you can press **F2** to return to the editor of **Esc** to exit the editor and return to immediate mode. Press **F2** to return to the editor. Now if you press **F3** to run your program again you will not be prompted to save the program as it has already been saved. Instead the program will be run again.

Let's look at a few more programming concepts. First of all though return to immediate mode (**F2**) so you have the ready prompt:

Variables

Variables are used in computer programs to store and retrieve pieces of information in the computer's memory. A variable is a friendly name given to a memory location. You could think of it like a box with a name on it. You can put something in the box and find it again later amongst lots of other boxes because it is labelled. You can look at what is in the box or change its contents. Variables are so called because their values can vary as a program is run.

You assign a value to a variable using the **LET** command. In immediate mode, enter the following command to set the value of a variable called X to 10:

Now to see the value of the variable X you can use the **PRINT** command as follows:

You can change the value in a variable by assigning it again to a different value. Try it now using the **LET** command again but setting a different value and using the **PRINT** command to see the value.

In fact you can get and set the value of a variable in the same command.

Suppose you want to increase the value in the variable X by 1. You first need to get the current value, then add 1 to it and then set it again to the new value as shown in the example below.

The plus operator (+) is typed by holding the shift key and pressing the = key at the same time. You may notice that I left out the **LET** command on the first line. This is because the **LET** command is used so much that it is implied when a variable starts a command and can be safely left out.

Variables should be named after the things that they contain so that you can remember what they are for. The variable called X does not really tell you very much about what it contains. For example if you have a variable that you want to store the current temperature in you would probably call it *Temperature* or even *CurrentTemperature*.

Although this requires more typing you will be glad you did it when you come back to your program and try to remember how it works!

You can set a string variable in the same way that you do a numeric one using the **LET** command (or leaving it out!). As before the character string is started and ended with the speech mark character (") which is **Shift** and 2 at the same time.

The contents of string variables can be changed in a similar way to numeric ones. For instance you can add them together:

In this case the plus operator (+) joins the two strings together to make a new longer string. You will notice that there is no space between *Hello* and *Colin*. To do this you would need to add a space either on the end of *Hello* or at the beginning of *Colin*.

Booleans

Some programming languages have a special type of variable called a Boolean. This is a variable that can only have one of two possible values: true and false. These are used by programs to make decisions about which pieces of code to execute. Other languages like FUZE BASIC just use a numeric value to do this job.

A numeric value of 0 is considered to be false and any value other than 0 is true, so for example 10 would be treated as true. There are also two built-in variables called TRUE and FALSE which represent these values.

Conditionals

Conditional statements allow the program to perform different actions depending on the value of a particular variable.

The simplest conditional statement is **IF**...**THEN** which executes a command only if a particular condition is met.

The command has the following syntax:

```
IF Condition THEN
  { Commands }
ENDIF
```

A Condition is an expression that evaluates to either true or false, for instance the expression **1 + 1 = 2** evaluates to true (check on the Internet if you don't believe me).

On the other hand the expression **6 * 9 = 42** evaluates to false because the value on the left of the equals sign (=) does not equate to the value on the right i.e. it is not the same.

Note: Usually in programming languages the asterisk (*) symbol is used to denote multiplication. The more normal x (times) used in mathematics would cause confusion with a variable called x.

The **ENDIF** command marks the end of the set of commands that will be executed if the condition is true.

Enter **Edit** mode and type the following program:

```
X=10
IF X=10 THEN
  PRINT "X IS 10"
ENDIF
END
```

If you then run this program (by pressing **F3** – remember that you will have to give it a name first so it can be saved on to the SD card) you will get the following result:

However if you change the first line to X=11 say and run the program again then nothing will be printed.

Suppose that we want a different action to be carried out if the condition is false. This is when you would use the **IF**...**THEN**...**ELSE** conditional construct which has the following syntax:

```
IF Condition THEN
  { Commands }
ELSE
  { Commands }
ENDIF
```

In this case if the condition is true the first set of commands is executed and if it is false the second set. So we could change our program to look like this:

```
X=11
IF X=10 THEN
  PRINT "X IS 10"
ELSE
  PRINT "X IS NOT 10"
ENDIF
END
```

This time when you run the program you will get this result:

What if you wanted to test a variable for a number of different values and execute different code in each case? In some versions of BASIC you have to use something called an IF THEN ELSE ladder as shown in this example:

```
X=2
IF X=1 THEN
  PRINT "X IS 1"
ELSE
  IF X=2 THEN
    PRINT "X IS 2"
  ELSE
    IF X=3 THEN
      PRINT "X IS 3"
    ELSE
      PRINT "X IS Something Else"
    ENDIF
  ENDIF
ENDIF
END
```

Each **IF** statement is nested within the **ELSE** part of the one above it. I think that you will agree that this is not easy to understand. Without the indentation it would be even worse and with it you can soon end up a long way over to the right of the screen as you add more tests. Anyway, luckily for you FUZE BASIC supports another conditional statement called a **SWITCH** statement. Using a **SWITCH** statement the above code can be rewritten as follows:

```
X=2
SWITCH (X)
  CASE 1
    PRINT "X IS 1"
  ENDCASE
  CASE 2
    PRINT "X IS 2"
  ENDCASE
  CASE 3
    PRINT "X IS 3"
  ENDCASE
  DEFAULT
    PRINT "X IS Something Else"
  ENDCASE
ENDSWITCH
END
```

I hope that you agree that this is easier to follow. If the value of X matches the value following a **CASE** statement then the code will be executed up to the next **ENDCASE** statement. If none of the values match and there is a **DEFAULT** statement then this code is executed instead.

Input/Output

Input /Output (or I/O) is the communication between the program and a human being or another device. You have already used the simplest output command **PRINT** to output text to the screen (and therefore to the user).

The simplest Input command is in fact **INPUT** which takes input from the keyboard (and therefore from the user) and puts it into a variable. Enter the following program using the editor (remember to type **NEW** to clear the current program from memory):

```
INPUT A
PRINT A
END
```

When you run this program (**F3**) you will be presented with an enigmatic question mark (?). This is prompting you to enter something using your keyboard. What you type will then be put into the variable called **A** when the **Enter** key is pressed. Because **A** is a numeric variable (it doesn't end with a $ sign) the value should be numeric otherwise a value of 0 will be assigned.

Try it now; type a number and press the **Enter** key. The number you type will be echoed back to the screen by the following **PRINT** command. If you enter something which is not a number 0 will be printed.

The question mark doesn't really tell us what the program is asking us to enter. We can add our own prompt by placing it between the **INPUT** command and the variable as in this example:

```
INPUT "What is your name? ", Name$
PRINT "Hello " + Name$
END
```

The text after the **INPUT** command is first printed to the screen. In this case the program is asking you to enter your name. Because the variable is a string one, this time when you press the **Enter** key everything that you type will be put into the string variable called **Name$**. Then whatever you type will be printed back to the screen with the word **Hello** in front of it e.g.

Now we can use conditional statements and input to make our programs do different things depending on what the user enters on the keyboard. Consider the following example:

```
INPUT "Enter Your Name: ", Name$
IF Name$ = "Dave" Then
  PRINT "I am sorry " + Name$
  PRINT "I am afraid I can't do that"
ELSE
  PRINT "No problem " + Name$
ENDIF
END
```

If you run this and enter *Dave* you will get this:

Anything else and you will get something like:

Note: Because the equals operator (=) is case sensitive if you were to type *dave* without a Capital letter you would also get the **ELSE** part of the conditional statement. Only typing Dave with a capital letter will produce the first result.

Loops

Loops enable sections of code to be executed more than once. This can be a specified number of times or until a condition is met. In fact as shown earlier in the manual a loop can be made to execute forever (or at least until the **Esc** key is pressed or the power is switched off!).

There are a number of different ways of doing loops in FUZE BASIC. The best one to use will vary according to what you are trying to do. The simplest loop you have already seen which is the one that never ends (infinite):

```
Seconds=0
CYCLE
  Seconds = Seconds + 1
  PRINT Seconds; " Seconds"
  WAIT(1)
REPEAT
```

In this example the variable called *Seconds* will initially be set to 0. The **CYCLE** command indicates the start of a loop. Then the value of *Seconds* will be increased by 1. Next the value will be printed to the screen with the word *Seconds* in front of it. On the next line **WAIT**(1) is calling a built in procedure which tells the program to do nothing for the specified amount of time (in this case 1 second). Finally the **REPEAT** command tells the program to jump back to the beginning of the loop (**CYCLE**). Note that no **END** command is needed as it will never be reached. This program when **RUN** will count the number of seconds since the program was started and print it to the screen. It will do this until the loop is interrupted by pressing the **Esc** key.

In practice it is not a good idea to use infinite loops because the program will never finish unless the user interrupts it and that is not very graceful. This brings us to the next type of loop which is conditional.

Suppose we want the program to do nothing until the user presses the space bar, maybe to give them time to read what is on the screen (such as the program's instructions). We can use a **WHILE** loop as shown below:

This needs some explaining. In this case **WHILE** the condition is true the loop will be executed. The condition is **INKEY** <> 32. **INKEY** is another built in variable like **TRUE** and **FALSE.** Whereas their values never change, **INKEY** will always contain the value of the last key pressed. I will cover this in more detail later but for now take it that the value that the space bar returns is 32. The <> operator means *not equal* and is the opposite of the equals operator (=) i.e. it will be true if the values either side are not the same. So as long as the key pressed is not the space bar the loop will be repeated.

Note: Unlike the **INPUT** command **INKEY** does not stop the program from running until the **Enter** key is pressed but instead carries on.

You will notice that there is no actual code inside of the loop (between the **CYCLE** and **REPEAT**) so this loop will execute very quickly (it has nothing to do). The loop will actually execute many times every second. How can we find out how many? We can use a variable to count them. Enter the program below:

```
PRINT "Press the space bar to continue"
Count=0
WHILE INKEY <> 32 CYCLE
  Count = Count + 1
REPEAT
PRINT Count
END
```

Run this program and after a few seconds press the space bar. Yes the loop really has been round that number of times! Modern computers are very fast compared to the ones I first used.

Suppose that we want to execute a loop a specific number of times. We could use a **WHILE** loop with a variable as shown below:

```
Count=0
WHILE Count < 10 CYCLE
  Count = Count + 1
  PRINT Count
REPEAT
END
```

This time I have used the < operator to test whether the value of the variable *Count* is less than 10. As long as it is, the loop will execute and increase the value of *Count* by 1. So this program will count from 1 to 10.

However in FUZE BASIC there is another way to do the same thing called a **FOR** loop as shown below:

```
FOR Count = 1 TO 10 CYCLE
  PRINT Count
REPEAT
END
```

The **FOR** loop sets the value of the variable following it to the initial value specified. It then executes the commands within the loop and repeats increasing the value by 1 each time around until the value after the **TO** command is reached.

Both ways are equally valid but the second is easier to understand and uses two lines less code to do the same thing. In programming there are often many different ways to achieve the same outcome and there isn't always a right way and a wrong way to do something.

There is another part to the FOR loop that lets you change the amount that the variable is increased by each time the loop is repeated. This is done using the **STEP** command as follows:

```
FOR Count = 1 TO 100 STEP 10 CYCLE
  PRINT Count;" ";
REPEAT
END
```

This will produce the following result:

Note: The semi-colon (;) symbol on the end of the **PRINT** command stops the printing of a new line. This means that everything will be printed across on the same line and not down the screen in a column.

You will notice that although the last value in the **FOR** loop was 100 this was never printed. When the previous value was increased by 10 it went over this value and the loop ended.

The next loop we will look at is a different version of the **WHILE** loop that we saw earlier. Because the **WHILE** loop tests the condition at the top of the loop it is possible that the loop will never be executed. Suppose that we want to make sure that the loop will be executed at least once. To do this we can use a **REPEAT UNTIL** loop as shown below:

```
CYCLE
  { Commands }
REPEAT UNTIL { Condition }
```

This will execute the *Commands* once before testing to see whether the *Condition* is true. If it is then the loop will end and the statement following will be executed. If it is false then the loop will start again from the **CYCLE** command.

Procedures and Functions

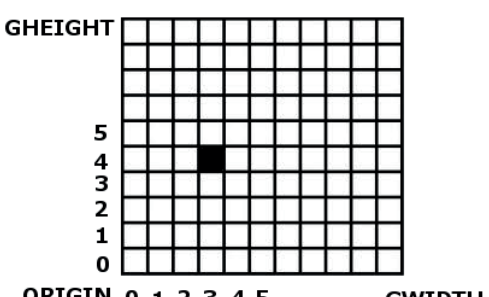
Procedures and functions are built in pieces of code that are not part of the syntax of the programming language but have been written to make your life easier. There is no point in you continually having to write your own code to draw a circle in every program where you need to. Instead you can use the built in procedure (but you don't have to if you don't want to!)

The only real difference between a procedure and a function is that functions give you back a value and procedures don't (they just carry out an action).

Procedures and functions may or may not have values passed to them (called arguments) which change the way they behave. FUZE BASIC has a large number of built in procedures and functions to carry out a number of different tasks such as graphics and mathematics. It is even possible to write your own which we will cover in a later section.

Graphics

Graphics are about drawing things on the computer's screen. You can think of the screen as a grid, like a sheet of graph paper, where each square is an individual picture element called a *pixel*. Each *pixel* has a position on the screen and a colour. The position is given as two numbers which show how far across the screen and how far up the screen from the bottom left hand corner it is as shown:



In the example shown the black pixel is at position 3 across and 4 up or (3, 4) counting from zero.

The position across or horizontal position is usually called the **x** coordinate and the position up or vertical position is called the **y** coordinate. So any pixel's position can be given as (**x**, **y**).

The bottom left corner or position (0,0) is also known as the **ORIGIN**. The number of pixels across and down the screen will vary depending on the graphics mode and on how FUZE BASIC has been set up. You can find out how many pixels there are using two built in variables called **GWIDTH** and **GHEIGHT**. **GWIDTH** gives the number of pixels across and **GHEIGHT** the number of pixels up so that the top right pixel on the screen is at position: (**GWIDTH**, **GHEIGHT**).

You can set the screen or window resolution to best suit your application. However it is also sensible to adapt your program to run in the resolution the device is set to. To find out the number of pixels you have in the current FUZE BASIC setup you could use the following program:

```
PRINT "GWIDTH=";GWIDTH
PRINT "GHEIGHT=";GHEIGHT
PRINT "PIXELS=";GWIDTH*GHEIGHT
END
```

Which when run will give you something like this:

So in this case there are over ¾ of a million pixels! You can set the colour of each individual pixel using the **PLOT** command. First you need to select a colour for your pixel using the **COLOUR** command.

FUZE BASIC has 30 predefined colours as listed below:

0 Black	10 Pink	20 Teal
1 DarkGrey	11 LightPink	21 Cyan
2 Grey	12 DarkGreen	22 Aqua
3 Silver	13 Green	23 LightBlue
4 LightGrey	14 BrightGreen	24 Brown
5 White	15 Olive	25 LightBrown
6 Maroon	16 Lime	26 Orange
7 Red	17 LightGreen	27 Gold
8 Purple	18 Navy	28 Yellow
9 Raspberry	19 Blue	29 LightYellow

To set a colour you can either use the number or the predefined constant so **COLOUR=RED** and **COLOUR=7** are the same thing. Once you have set the colour, all of the following graphic commands will use this colour until you change it again. The syntax of the plot command is:

PLOT(x,y). So to set the pixel in the middle of the screen you would use:

PLOT(GWIDTH/2,GHEIGHT/2)

Note: the / symbol is used to denote division, so / 2 means divide by 2. The more usual mathematical symbol is ÷ which is not on your keyboard!

Now select the colour red using **COLOUR=RED**. Finally plot a pixel using the command above. You should just about be able to see a red pixel in the middle of the screen. You may have to look hard to see it!

It will obviously take a long time to draw anything on the screen one pixel at a time even in the low resolution graphics mode. Luckily there are lots of built in graphics procedures for drawing different shapes. The simplest of these, **LINE** draws a line in the current **COLOUR** between two points on the screen:

```
LINE(xpos1, ypos1, xpos2, ypos2)
```

This will draw a straight line between point (xpos1, ypos1) and point (xpos2, ypos2). For example to draw a yellow line from the bottom left of the screen to the top right you would use the following program:

```
CLS
COLOUR=YELLOW
LINE(0,0,GWIDTH,GHEIGHT)
END
```

The CLS command simply clears the screen display. Say we want to draw a box on the screen. As usual there are several ways that we could do this. We could use the **LINE** procedure that we have just met:

```
CLS
COLOUR=LIME
LINE(100,100,200,100)
LINE(200,100,200,200)
LINE(200,200,100,200)
LINE(100,200,100,100)
END
```

You will notice that the endpoint of each line is the start point of the next one. We can use the **LINETO** procedure to simplify this as follows:

```
CLS
COLOUR=LIME
LINE(100,100,200,100)
LINETO(200,200)
LINETO(100,200)
LINETO(100,100)
END
```

LINETO always uses the previous point plotted as the start point of the next line. In fact we don't even need to do this because there is built in procedure for drawing rectangles called **RECT** which has the following syntax:

```
RECT(xpos,ypos,width,height,fill)
```

In this case the point (*xpos*, *ypos*) is the bottom left hand corner of the box and the *width* and *height* are the number of pixels to draw across and down from there. So our program is now simplified again to:

```
CLS
COLOUR=LIME
RECT(100,100,100,100,FALSE)
END
```

This will draw exactly the same box as before. What do you think will happen if you change the value of *fill* from **FALSE** to **TRUE**? Try it and see. Did you guess correctly?

There are a couple of other shapes that you can draw easily. Firstly a circle using the **CIRCLE** procedure:

```
CIRCLE(centrex,centrey,radius,fill)
```

Here the point (*centrex*, *centrey*) is the middle of the circle and the *radius* is the distance of each point in the circle from the middle (in pixels). Again, if *fill* is TRUE then the shape will be filled in, otherwise it will just be an outline.

Finally **TRIANGLE** does exactly what you would expect:

```
TRIANGLE(x1, y1, x2, y2, x3, y3, fill)
```

In this case the 3 points (*x1*, *y1*), (*x2*, *y2*) and (*x3*, *y3*) are the points of a triangle and fill is whether to fill it or not.

At this point there is one other graphics command that you need to know about. This is the **UPDATE** command and it causes the screen to be refreshed (or redrawn). As we saw earlier there are a lot of pixels on the screen and if we updated them after every graphics command it would make the program very slow and flickery. So instead the graphics commands are carried out behind the scenes in memory and then drawn on the screen when an **UPDATE** command is used.

Up until this point all of the examples we have done have been programs that ended (with an **END** command) and this is causing an automatic **UPDATE**. Consider a non-terminating program like this one :

```
CLS
COLOUR=AQUA
CYCLE
  CIRCLE(GWIDTH/2, GHEIGHT/2, 100, TRUE)
REPEAT
```

When we **RUN** this the screen remains blank until we press the Esc key and the program is interrupted. If we now change the program to this:

```
CLS
COLOUR=AQUA
CYCLE
  CIRCLE(GWIDTH/2, GHEIGHT/2, 100, TRUE)
  UPDATE
REPEAT
```

The circle appears immediately. In fact we can call as many graphic procedures as we like and they will only appear when an **UPDATE** command is called.

Note: In FUZE BASIC the PRINT command (which prints text to the screen has been made by default to do an **UPDATE** as well when a new line is printed).

Comments

Comments are bits of text that you add to your program to make it easier for someone else to understand what it is doing and how it works. They are also for you when you come back to a program that you wrote earlier to remind you of the same!

In FUZE BASIC comments are added using the **REM** (remark) command at the beginning of a line. You can then type whatever you like as it will be ignored by the interpreter.

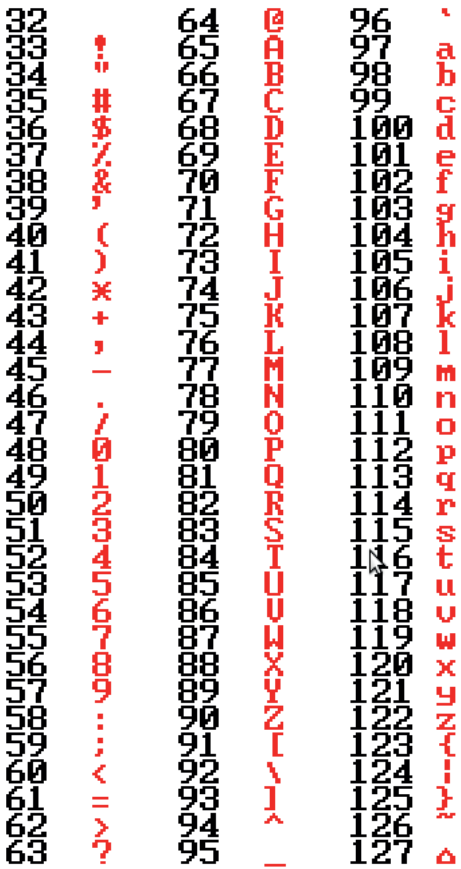
```
REM This is a comment
```

You don't need to comment on every line, but each routine or block of code should have one. You can also use // to add comments. This can be added at the end of a command:

```
PRINT "Hello Word" // This is a comment too
```

As mentioned before everything is stored in the computer as numbers. How do we convert from these numbers to letters, numbers and punctuation characters on the output device? The most commonly used system is one called ASCII which defines how each number maps to a specific character. This is a standard and as long as you know that a system uses it you will be able to understand something encoded in it. For example in ASCII the number 65 is always treated as the capital letter A. The standard ASCII character set is shown overleaf:

ASCII



Text

OK we have covered drawing stuff on the screen, now in this section I will talk about processing text. Text is just the letters, numbers and punctuation symbols that you enter with your keyboard and read off your screen. Computers are quite good at manipulating it.

You have already seen the two simple commands INPUT to read from the keyboard and PRINT to output to the screen. FUZE BASIC has a number of built in functions for manipulating strings of characters (called string functions). The first one we will look at is the LEN function. This tells you how many characters (letters, numbers and punctuation – including spaces) there are in a particular string. So for instance:

```
PRINT LEN("Hello World!")
```

This will print 12 (10 letters, 1 punctuation and 1 space). You may ask why would we want to know that? We can obviously count them ourselves easily enough. Suppose however that the string was something that had been entered by the user using the INPUT command and placed in a variable:

```
INPUT "What is your name? ",name$
```

We do not know how many characters the user will type and how long the string in the variable name$ will be. Again you may be asking why we would need to know. Suppose we want to print the person's name centred on the screen. To do this we will first need to know how many characters will fit across the screen. This will depend on how FUZE BASIC has been set up but you can find it out using the TWIDTH built-in variable. So to centre the text you could subtract the length of the string from this number, divide it by 2 and then print this number of spaces before it as follows:

```
INPUT "What is your name? ",name$
PadLength=(TWIDTH – LEN(name$))/2
FOR P=1 TO PadLength CYCLE
  PRINT " ";
REPEAT
PRINT name$
END
```

As usual there is a simpler (and better) way of doing this same thing:

```
INPUT "What is your name? ",name$
PadLength=(TWIDTH – LEN(name$))/2
PRINT SPACE$(PadLength) + name$
```

The **SPACE**$ function returns a string of the specified number of space characters.

There are 3 string functions provided that let you chop up a string into pieces. These are **LEFT$, RIGHT$** and **MID$**. The first of these **LEFT$** returns the specified number of characters from the left hand side of the string. If the number is greater than the length of the string then it returns the whole string. Here is an example and the result of running it:

```
string$="0123456789"
FOR C=1 TO LEN(string$) CYCLE
  PRINT LEFT$(string$, C)
REPEAT
END
```

As you might expect **RIGHT$** returns the specified number of rightmost characters:

```
string$="0123456789"
FOR C=1 TO LEN(string$) CYCLE
  PRINT RIGHT$(string$, C)
REPEAT
END
```

The final one **MID$** has an extra argument which specifies the start position in the string (starting from zero):

```
result$ = MID$(string$, start, count)
```

So this can be used to extract a string from the middle of the string as shown below:

```
string$="0123456789"
FOR C=1 TO LEN(string$) CYCLE
  PRINT MID$(string$, C-1, 1)
REPEAT
END
```

Note: We have subtracted one from **C** here to make sure that it starts from position 0 which is the start of the string. If the start position is past the end of the string then you will get an error.
Now you may be wondering what the point of all this is so I will give you a 'Real World' example. Suppose you want to ask someone their name and get them to type it into the computer but that you want to make sure that it is formatted correctly. The user could type all in lower case letters or in CAPITALS. This does not look very good; you would normally print a name with an initial Capital letter and then lower case. We could write a program to make sure that the name was in this format. Here is an example of (one) way that you could do it:

```
INPUT "What is your name? ",name$
newname$=""
FOR C=1 TO LEN(name$) CYCLE
  char$=MID$(name$, C - 1, 1)
  IF C = 1 THEN
    IF char$ >= "a" AND char$ <= "z" THEN
      char$ = CHR$(ASC(char$) - 32)
    ENDIF
  ELSE
    IF char$ >= "A" AND char$ <= "Z" THEN
      char$ = CHR$(ASC(char$) + 32)
    ENDIF
  ENDIF
  newname$=newname$+char$
REPEAT
PRINT "Hello " + newname$
END
```

Now this looks complicated but it really isn't, I will walk you through it. The first line prompts the user to enter their name and puts it into the string variable *name$*. The next creates a new empty string variable in which to store our new formatted name.

We then start a loop from 1 to the length of the name that has been entered. The first thing we do in the loop is use the **MID$** function to get the next character from the string and place it into a variable called *char$.*

We then test to see if the loop variable C is 1, that is we have the first character of the name. If it is then we want to make sure that it is a capital letter. Notice that you can compare letters as if they were numbers so for example the letter **a** is less than the letter **z**.

Note: The <= operator means less than or equal to and the >= means greater than or equal to.

If the first character is a lower case letter (between **a** and **z**) then we need to convert it to a capital one. As has been mentioned before the letters are really stored as numbers, in this case in **ASCII** format. Because of the way that the characters are laid out in **ASCII** you can easily convert from lower case to upper case by subtracting 32 from the codes.

First you have to find the **ASCII** code for your character using the **ASC** function. Then you can subtract 32 from this to find the code for the equivalent capital letter. Finally you need to convert it back from a code to a character again using the CHR$ function. You could do this on separate lines of code but it is also possible to do it all in one go as shown:

```
char$ = CHR$(ASC(char$) - 32)
```

The **ELSE** part of the **IF** statement will be executed when C is greater than 1 which will be every other time around the loop. This time we test for the letters being capital ones and convert them to lower case.

Finally we add our converted (or not) character onto our new string variable. Once every character has been checked and converted if necessary we can print out our new formatted name string.

As usual there are other ways this could be done and the program does not check that the user entered just letters. What if they had typed numbers or punctuation or even their full name with surname? It is difficult to allow for every possible thing that a user could enter.

Bits and Bytes

As I mentioned before information is stored in a computer as a series of 1s and 0s. Each one of these 1s or 0s is known as a **bit** and is represented in the computer by something being either **on** (1) or **off** (0).

We count in a system called decimal which uses base 10. This is mostly because with have ten digits (fingers and thumbs) on our hands. Computers count in a system called binary which uses base 2. We count from 1 to 9 and then start a 10s column. In binary you count from 0 to 1 and then start a 2s column.

Each column is then 2 times the one before (just as in decimal it is 10 times).

8 bits of computer storage are known as a **byte**. This can be used to store a decimal number in the range 0 to 255. Here is an example:

128	**64**	**32**	**16**	**8**	**4**	**2**	**1**
1	0	1	1	0	0	0	1

To convert 10110001 in binary to decimal you do (128 x 1) + (64 x 0) + (32 x 1) + (16 x 1) + (8 x 0) + (4 x 0) + (2 x 0) + (1 x 1). Anything times 0 is 0 and anything times 1 is itself so this simplifies to 128 + 32 + 16 + 1 = 177.

My first computer had 32KB (Kilobytes) which is about 32,000 Bytes of memory. Your FUZE has 1GB (Gigabyte) which is over 1,000,000,000 bytes. The SD card sticking out of the back has at least 8 GB (Gigabytes) of storage which is over 8,000,000,000 bytes. You can now buy hard disk drives with a capacity of 8TB (Terabytes) which is an enormous 8,000,000,000,000 Bytes of data!

There's an old computer programming joke: There are 10 sorts of people in the world; those who understand binary and those who don't!

Arrays

The next key concept that I want to introduce is called an array or indexed variable. An array is really just a way of grouping together related variables in a convenient way for processing.

This is best illustrated by an example. Say that you wanted to store a list of 5 numbers given by the user so that you can then perform calculations on them such as adding them together. You could do something like this:

```
Input "Enter Number 1: ", Num1
Input "Enter Number 2: ", Num2
Input "Enter Number 3: ", Num3
Input "Enter Number 4: ", Num4
Input "Enter Number 5: ", Num5
Sum = Num1 + Num2 + Num3 + Num4 + Num5
PRINT "Sum = "; Sum
END
```

This will work perfectly well but what if we want to add 20 numbers? or 30? or indeed an unknown number. This is where an array can come in useful. An array is a variable that can store multiple values which can be accessed using a number called an *index*. The indexed array variables are known as array elements.

To create an array we have to specify the maximum number of elements that we want to store. This can be tricky because we want to leave enough space for our use but have to bear in mind that computer memory (a finite resource) is being allocated whether we use it or not, so try to make arrays as small as possible. To create an array you use the *Dim* command (dimension) as follows:

```
DIM VariableName(Size)
```

Where *VariableName* is the name of the array and *Size* is the maximum size. To access the elements of the array you then use:

```
VariableName(Index)
```

where *Index* is a number from 0 to *Size*. If you try to access an array element greater than *Size* you will get an *Out of bounds* error.

So we can now rewrite our summing program using an array as follows:

```
DIM Num(5)
FOR Index= 1 to 5 CYCLE
  INPUT "Enter Number: ", Number
  Num(Index) = Number
REPEAT
Sum=0
FOR Index= 1 to 5 CYCLE
  Sum = Sum + Num(Index)
REPEAT
PRINT "Sum = "; Sum
END
```

You will notice that this program is actually slightly longer than the original, but if we wanted to add 10 numbers rather than 5 in the first case we would almost double the length of the program whereas in the second you would just change the 5s into 10s.

As well as arrays of numbers you can also have arrays of strings. These work in a very similar way to number arrays but the values stored are character strings. To dimension a string array you just add the $ sign to the end of the name, same as with a string variable:

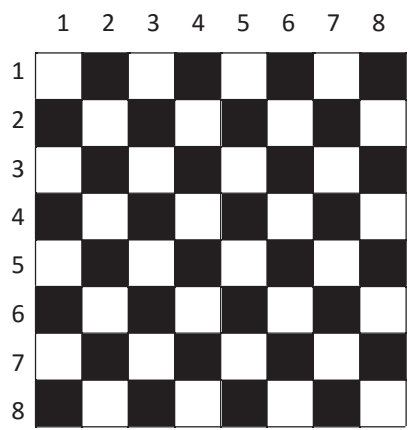
```
DIM DayOfWeek$(7)
```

You can then access the array elements in the same way as in a numeric array:

```
DayOfWeek$(1) = "Monday"
DayOfWeek$(2) = "Tuesday"

PRINT "The first day of the week is ";
PRINT DayOfWeek$(1)
```

Arrays can also have more than one dimension. Suppose that we wanted to write a program to play chess. We could represent the board using a 2 dimensional array. A chess board has 8 rows and 8 columns as shown below:



To create an array to represent this we would use the following:

```
DIM ChessBoard(8,8)
```

We can then access individual board positions by cross referencing the row number and the column number as shown below:

	1	2	3	4	5	6	7	8
1	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)
2	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	(2,8)
3	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)	(3,8)
4	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)	(4,8)
5	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)	(5,8)
6	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)	(6,8)
7	(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)	(7,8)
8	(8,1)	(8,2)	(8,3)	(8,4)	(8,5)	(8,6)	(8,7)	(8,8)

For example the position highlighted would be accessed using *ChessBoard(4,5)*

Say that we wanted to store the colour of each position in the array we could use the following:

For example the position highlighted would be accessed using *ChessBoard(4,5)*

Say that we wanted to store the colour of each position in the array we could use the following:

```
DIM ChessBoard(8,8)
Count = 0
FOR Row= 1 TO 8 CYCLE
  FOR Col = 1 To 8 CYCLE
    Count = Count + 1
     IF  Count MOD 2 = 1 THEN
       ChessBoard(Row, Col) = 7
     ELSE
       ChessBoard(Row, Col) = 0
     ENDIF
  REPEAT
REPEAT
END
```

The *Row* loop executes 8 times. Within it the *Col* loop executes for each row. This means that the *Count* variable is incremented 64 times going from1 to 64. The MOD (modulus) operator gives the remainder when the first number is divided by the second (if there is no remainder it returns 0). So anything MOD 2 will return either 1, if it is an odd number or 0 if it is even. The net result is that the value will toggle between 1 and 0 (or True and False) as *Count* increases i.e.

```
1 Mod 2 = 1 (True)
2 Mod 2 = 0 (False)
3 Mod 2 = 1 (True)
4 Mod 2 = 0 (False)
5 Mod 2 = 1 (True)
And so on.
```

In fact arrays can have any number of dimensions limited only by the amount of available memory.

There is another type of array supported by FUZE BASIC which is called an associative array. This allows you to associate a key with an array index. So for example youcould create an array to hold information about a person and then use the key to store different bits of information could create an array to hold information about a person and then use the key to store different bits of information about them:

```
DIM Person$(10)
Person$("firstname") = "Ford"
Person$("surname") = "Prefect"
PRINT "Full name: ";
PRINT Person$("firstname");" ";
PRINT Person$("surname");
END
```

Data, Read and Restore

These three commands provide an easy way to initialise (set initial values) variables with a set of values. This is probably best shown with an example. Suppose that you have an array to store the names of the days of the week. You could set it up like this:

```
Dim DayOfWeek$(7)
DayOfWeek$(1) = "Monday"
DayOfWeek$(2) = "Tuesday"
DayOfWeek$(3) = "Wednesday"
DayOfWeek$(4) = "Thursday"
DayOfWeek$(5) = "Friday"
DayOfWeek$(6) = "Saturday"
DayOfWeek$(7) = "Sunday"
```

This will work fine but as the list of values gets longer it will start to make the program get very long. A neater way to do this is to use the **DATA** command to store the values and then **READ** to set them as follows:

```
DIM DayOfWeek$(7)
FOR DayNo = 1 TO 7 CYCLE
  READ DayOfWeek$(DayNo)
REPEAT
END
DATA "Monday", "Tuesday", "Wednesday"
DATA "Thursday", "Friday", "Saturday"
DATA "Sunday"
```

The **DATA** statements can actually occur anywhere in the program but are usually placed at the end. The **READ** command will start reading values from the first **DATA** statement it finds in the program and continue from there.

The **RESTORE** command will set it back to the first one again if you needed to reread it. This could be useful if the values have been changed by the program but you want to reset them back to their initial values.

Files

Another way to store data is as collections of bytes on a storage device (such as an SD card or hard drive) called files. Files fall broadly into two types: text, consisting of **ASCII** characters and readable by a human and binary, consisting of 1s and 0s and readable by machines only. Binary data is used to store things like images, video or music.

FUZE BASIC has support for creating files. You have already been using text files as they are used to store the FUZE BASIC programs themselves. To demonstrate this let's create a program to read itself!

In the editor enter the following program:

```
handle=OPEN("ReadSelf.fuze")
UNTIL EOF(handle) CYCLE
  INPUT# handle, record$
  PRINT record$
REPEAT
CLOSE (handle)
END
```

When you run this (**F3**) and are prompted to enter a file name, enter **ReadSelf** (make sure that the case of the letters matches exactly). You should see the program printed to the screen. So how does this work?

The first line opens a file in the current location called *ReadSelf.fuze* and returns a number into the variable called *handle*. This number, called a file handle, is linked to this file and we can access the file by passing the number to file handling routines. If the file does not exist then a new file will be created.

Note: File names usually have two parts. The first part is its name and the second part after the dot is called the file extension. This is used to tell what type of file it is, for example a music file might have the file extension **.mp3** and a text file **.txt**. FUZE BASIC programs have the file extension **.fuze**

The second line starts a loop which will loop as long as **EOF**(*handle*) is **FALSE**. The **EOF** function returns **TRUE** when the end of the file has been reached and **FALSE** otherwise so initially this should be **FALSE** as we are at the start of the file.

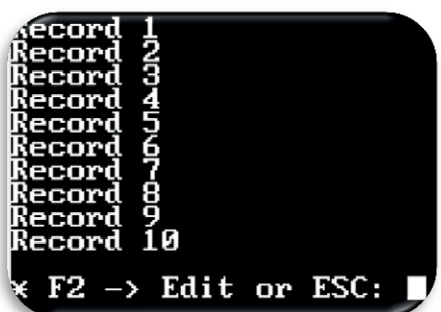
The next line uses a special version of the **INPUT** command **INPUT#** which instead of reading from the keyboard reads from the file linked to the file *handle* that follows it and puts the result in the variable specified (in this case the string variable *record*$). The INPUT# command will read up until the end of a line is reached.

Finally the contents of the *record*$ variable is printed to the screen and the program loops around. When the end of the file is reached **EOF**(*handle*) returns **FALSE** and the loop terminates. Finally the **CLOSE** statement is used to close the file and release the file handle (which can then be used again for a different file if required).

So that is how you read a file, how do we go about writing to one? Just as you use **INPUT#** instead of **INPUT** you can use **PRINT#** instead of **PRINT** to print to a file.

```
handle=OPEN("TestFile.txt")
FOR RecNo=1 to 10 CYCLE
  PRINT# handle,"Record "; RecNo
REPEAT
CLOSE(handle)
handle=OPEN("TestFile.txt")
UNTIL EOF(handle) CYCLE
  INPUT# handle, record$
  PRINT record$
REPEAT
CLOSE (handle)
END
```

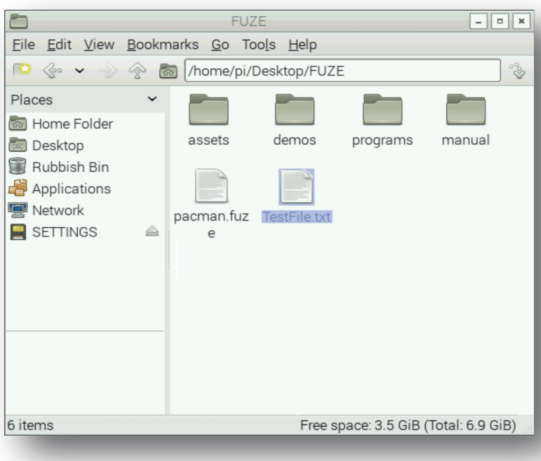
Run this program and you will get the following:



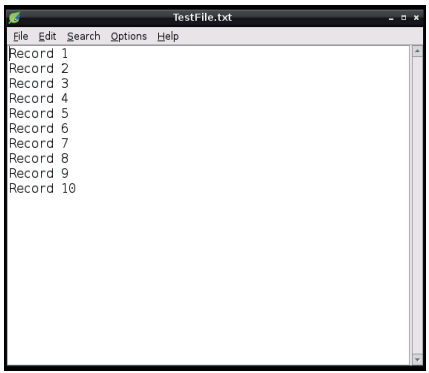
This program first opens a file called *TestFile.txt* and creates it if it doesn't exist. It then loops with the variable *RecNo* going from 1 to 10.

The **PRINT#** command then prints *Record* and the value of *RecNo* to the file linked to *handle*. The file is then closed and reopened again. This will position the file handle back to the start again. The file is then read back and printed to the screen as before. On the left hand side of the FUZE Desktop you will see an Icon that looks like this:

If you double click on this icon it will open the File Manager window for the folder where FUZE BASIC stores its files by default:



In this folder you should see the file you have just created. If you double click on this file you will see its contents in the editor:



There is another way to position the file back to the start and that is using the **SEEK** function which has the following syntax:

```
SEEK(handle, offset)
```

The offset argument supplied is the position in the file to jump to, 0 being the beginning of the file. So instead of closing and reopening the file we could have simply written:

```
SEEK(handle, 0)
```

We can use the **SEEK** function to access the file at any point rather than having to read it sequentially from the start. This is known as random file access. To do this successfully we need to know how long the records are and they need to be a fixed size.

```
handle=OPEN("TestFile.txt")
recSize = 20
FOR recNo=0 to 10 CYCLE
  record$ = "Record " + STR$(recNo)
  pad = recSize - LEN(record$)
  PRINT# handle,record$;SPACE$(pad)
REPEAT
SEEK (handle,(recSize+1)*7)
INPUT# handle,record$
PRINT record$
CLOSE(handle)
END
```

In this example the maximum record size is set to be 20 characters. Each record is then padded with spaces to be exactly this length using the **SPACE**$ function. Then you can seek to any record using:

```
SEEK(handle,(recSize+1)*recNo)
```

where recNo is the record number. Notice that you have to add 1 to the record size. This is to skip over the new line character (ASCII character 10) at the end of each record.

There are a couple of other file handling functions that I haven't covered yet. The **REWIND**(*handle*) function sets the file back to the beginning. This is equivalent to **SEEK**(*handle*, 0).

The fast forward function **FFWD**(*handle*) sets the file pointer to the end of the file. This allows you to append records to the end of the file.

Sprites

Now for some fun! Sprites are small computer graphics that can be animated and moved around on the screen. They form the basis of a lot of computer games. To create sprites you really need to use a sprite editor or graphic editing program. However you can use FUZE BASIC'S ability to be able to save what's on the screen to a file to create your own using the graphic routines you saw earlier.

The SAVEREGION function takes a snapshot of the specified area of the screen and saves it to a file:

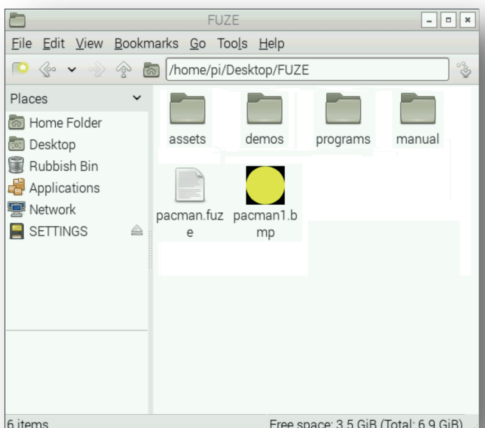
SAVEREGION(*file$,xpos,ypos,width,height*)

The region is given by the rectangle with bottom left at point (*xpos,ypos*) with *width* and *height pixels,* and is saved to a file named *file$* in a bitmap (.bmp) format.

Let's try this now. We will draw a yellow circle on the screen and save it to a file:

```
CLS
COLOUR=YELLOW
CIRCLE(100,100,50,TRUE)
SAVEREGION("pacman1.bmp",50,50,101,101)
END
```

Now if you open the FUZE folder on the Desktop you will see this:



Our yellow circle has been saved to a file called *pacman1.bmp*. Now let's create another image with a triangular piece missing. We can do this by drawing a black triangle on our circle and saving it again:

```
CLS
COLOUR=YELLOW
CIRCLE(100,100,50,TRUE)
COLOUR=Black
TRIANGLE(100,100,150,125,150,75,TRUE)
SAVEREGION("pacman2.bmp",50,50,101,101)
END
```

If you run this there should now be a new file in the folder that looks like this:

Finally we want to create another one with a bigger chunk missing, so we just make the triangle bigger and draw it again:

```
CLS
COLOUR=YELLOW
CIRCLE(100,100,50,TRUE)
COLOUR=Black
TRIANGLE(100,100,150,150,150,50,TRUE)
SAVEREGION("pacman3.bmp",50,50,101,101)
END
```

Now that we have created our bitmap images we can make a sprite using the **NEWSPRITE** function. This takes an argument to say how many images it will contain, in this case 3.

We can then use **LOADSPRITE** to load the images from a file:

```
CLS
pacman=NEWSPRITE(3)
LOADSPRITE("pacman1.bmp",pacman,0)
LOADSPRITE("pacman2.bmp",pacman,1)
LOADSPRITE("pacman3.bmp",pacman,2)
```

LOADSPRITE has 3 arguments. The first is the name of the file to load, the second is the handle of the sprite which was returned by **NEWSPRITE** and the last one is the image index (which goes from 0 to the number of images minus one).

To draw our sprite on the screen we use the **PLOTSPRITE** function which has the following syntax:

```
PLOTSPRITE(handle,xpos,ypos,index)
```

The *handle* argument is returned from the **NEWSPRITE** function and index is as above. The point (*xpos*, *ypos*) specifies the bottom left hand corner of the bounding rectangle of the sprite. When you call **PLOTSPRITE** the sprite at the current position is cleared and it is redrawn at the new one. So now we can make our sprite move across the screen using the following program:

```
CLS
pacman=NEWSPRITE(3)
LOADSPRITE("pacman1.bmp",pacman,0)
LOADSPRITE("pacman2.bmp",pacman,1)
LOADSPRITE("pacman3.bmp",pacman,2)
FOR X=1 TO GWIDTH STEP 25 CYCLE
  FOR S=0 TO 2 CYCLE
    PLOTSPRITE(pacman,X,GHEIGHT/2,S)
    UPDATE
    WAIT(.05)
  REPEAT
REPEAT
HIDESPRITE(pacman)
END
```

You will notice that there are two loops, one inside the other. The first one moves the sprite across the screen. The second cycles though the different versions of the sprite. Finally when it has reached the other side of the screen the **HIDESPRITE** function removes it from the screen.

When writing games you quite often need to know when one sprite collides with another one on the screen. First let's create another sprite using the following program:

```
CLS
COLOUR=RED
CIRCLE(100,100,50,TRUE)
RECT(50,50,101,50,TRUE)
COLOUR=WHITE
ELLIPSE(72,120,13,17,TRUE)
ELLIPSE(112,120,13,17,TRUE)
COLOUR=BLUE
CIRCLE(68,115,7,TRUE)
CIRCLE(108,115,7,TRUE)
COLOUR=BLACK
TRIANGLE(55,49,65,65,80,50,TRUE)
TRIANGLE(85,49,95,65,110,50,TRUE)
TRIANGLE(115,49,125,65,140,50,TRUE)
SaveRegion("ghost1.bmp",50,50,101,101)
END
```

Now we can use the SPRITECOLLIDE function to work out when our two sprites have hit each other as shown below and opposite:

```
CLS
pacman=NEWSPRITE(3)
LOADSPRITE("pacman1.bmp",pacman,0)
LOADSPRITE("pacman2.bmp",pacman,1)
LOADSPRITE("pacman3.bmp",pacman,2)
ghost=NEWSPRITE(1)
LOADSPRITE("ghost1.bmp",ghost,0)
midY=GHEIGHT/2
```

(Continued)

```
FOR X=0 TO GWIDTH STEP 10 CYCLE
  PLOTSPRITE(pacman,X,midY,X MOD 3)
  PLOTSPRITE(ghost,GWIDTH-X-100,midY,0)
  IF SPRITECOLLIDE(pacman) <> -1 THEN
    BREAK
  ENDIF
  UPDATE
  WAIT(0.1)
REPEAT
END
```

When you call SPRITECOLLIDE it will return -1 if the given sprite is not in contact with any other and the handle of the colliding sprite if it is.

User Defined Procedures & Functions

As mentioned before it is possible for you to define your own procedures and functions. This allows you to reuse code and makes your program much more readable. It will also make your program easier to maintain and modify.

As usual I will try to illustrate this using an example. Say you want to print out some messages and you want them to be centred on the screen. I showed you how this could be done before:

```
Message$="This text will be in"
PadLength=(TWIDTH - LEN(message$))/2
PRINT SPACE$(PadLength) + message$
Message$="the centre of each"
PadLength=(TWIDTH - LEN(message$))/2
PRINT SPACE$(PadLength) + message$
Message$="line on the"
PadLength=(TWIDTH - LEN(message$))/2
PRINT SPACE$(PadLength) + message$
Message$="screen"
PadLength=(TWIDTH - LEN(message$))/2
PRINT SPACE$(PadLength) + message$
END
```

You will notice that we have repeated some code here. Line 2 is the same as line 5, 8 and 11. Line 3 is the same as line 6, 9 and 12. This is a good sign that we should probably use a procedure.

```
PROC CentreText("This text will be in")
PROC CentreText("the centre of each")
PROC CentreText("line on the")
PROC CentreText("screen")
END
REM Centre text on the line
DEF PROC CentreText(message$)
  PadLength=(TWIDTH - LEN(message$))/2
  PRINT SPACE$(PadLength) + message$
ENDPROC
```

I hope you will agree that this is much neater and easier to read and understand. The procedure is defined using the **DEF PROC** (define procedure) command and this is placed after the **END** of the program as it will only be used when it is called using the **PROC** command.

In this example the procedure has one variable (called a parameter) which is the text string to be printed but it could have zero or many (in which case they would be separated by commas). When the procedure is called the argument, in this case the text to be printed is passed into this parameter.

When the **ENDPROC** command is executed the program returns to the point where it was called and resumes on the following line.

What if you decide that you want the text to be in upper case (capital letters)? For this we could write a function to convert any string to upper case.

Note: The only difference between a procedure and a function is that a function returns a value.

```
DEF FN ToUpper(string$)
  result$=""
  FOR C=1 TO LEN(string$) CYCLE
    char$=MID$( string$, C - 1, 1)
    IF char$ >= "a" AND char$ <= "z" THEN
      char$ = CHR$(ASC(char$) - 32)
    ENDIF
    result$= result$+char$
  REPEAT
=result$
```

This uses the code that we wrote earlier but puts it into a user defined function using the **DEF FN** command. You would call this function using the **FN** command:

```
message$=FN ToUpper("String to convert")
```

The last line *=result$* returns the value in this variable and places it in the variable on the calling line.

You can think of writing procedures and functions as writing your own extensions to the language. You could even write your own library of standard routines to be included into your programs.
There is one other thing that I need to mention about procedures. If you set the value of a variable in a procedure and you use the same variable name in the main body of the program then the value will be changed there as well.

This is because variables are global by default. However it is possible to declare a variable as being local to a procedure to avoid any nasty side effects. Consider the following example:

```
X=10
PRINT "1 Global X="; X
PROC Test()
PRINT "2 Global X="; X
END
DEF PROC Test()
  PRINT "3 Global X="; X
  X=15
  LOCAL X
  X=5
  PRINT "4 Local X="; X
ENDPROC
```

This will produce the following result when run:

So let me just run through this. First you define a (global) variable X and give it a value of 10. Then you call a procedure called Test. This first prints out the value of the (global) variable X. The value is then changed to 15. It then declares a new local variable X and gives it the value 5. As long as you are in the procedure this is the one that will be used.

When the procedure ends and the program goes back to the calling line the local variable X is destroyed. However the global variable X still exists and has the last value that it was set to (15).

In general it is good practice to declare variables local to a procedure or function and to pass any values required as parameters rather than using global variables. This prevents unforeseen consequences!

Numerical Functions

Computers are good for doing mathematical calculations and can perform many thousands per second. I have already covered some arithmetic operators such as add (+), multiply (*), subtract (-) and divide(/) but there are also a number of built in functions in FUZE BASIC to do more complicated mathematical calculations for things like trigonometry (the mathematics of triangles).

I am not going to cover all of these in detail as you can find them in the language reference section of this manual but I will just go over an example.

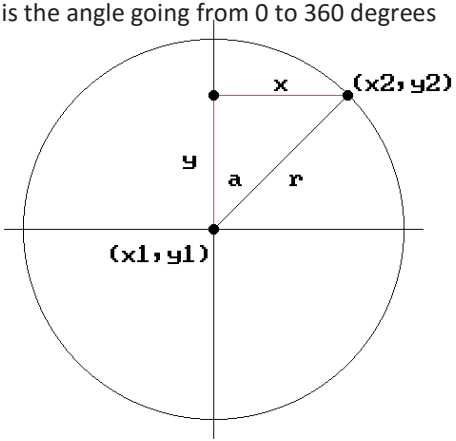
Suppose we want to draw the second hand of a clock and make it move round once per second. We can use the trigonometric functions **SIN** (sine) and **COS** (cosine) to do this. Without going

into a huge amount of detail, for a circle of radius **r** and centre at point (**x1**,**y1**) we can calculate any given point (**x2**,**y2**) using:

x2 = r * SIN(a) + x1 and

y2 = r * COS(a) + y1

where **a** is the angle going from 0 to 360 degrees



By default angles in FUZE BASIC are measured in degrees where there are 360 degrees in a circle. You can change the units to Radians using the **RAD** command (there are 2 * PI Radians in a circle) or to minutes using the **CLOCK** command (there are 60 minutes in a circle). You can change back to degrees using the **DEG** command.

So to draw our second hand we can set the units to CLOCK and then loop from 0 to 60 minutes (this is the same as using degrees and looping from 0 to 360 with a STEP of 5). Our program could then look like this:

```
CLS
CLOCK
FOR Angle=0 TO 60 CYCLE
  PROC SecondHand(Angle-1, BLACK)
REPEAT
FOR Angle=0 TO 60 CYCLE
  PROC SecondHand(Angle, WHITE)
  PROC SecondHand(Angle-1, BLACK)
  UPDATE
  WAIT(1)
REPEAT
END
REM Draw second hand on the screen
DEF PROC SecondHand(seconds, handColour)
  COLOUR=handColour
  Xpos=200*SIN(seconds)+GWIDTH/2
  Ypos=200*COS(seconds)+GHEIGHT/2
  LINE(GWIDTH/2,GHEIGHT/2,Xpos,Ypos)
  IF handColour = BLACK THEN
    COLOUR=WHITE
    CIRCLE(Xpos,Ypos,2,TRUE)
  ENDIF
ENDPROC
```

The first loop draws a white circle at 1 minute intervals. The second draws the second hand and erases the previous second hand. It then waits for 1 second before looping around.

The output from our program is shown below:

Turtle Graphics

FUZE BASIC supports another type of computer graphics called Turtle Graphics. They are called this because they were originally used to control a small robot which had a domed top and looked a bit like a turtle. This was done using a programming language called LOGO which was designed to teach people (and especially children) about programming.

The turtle had a pen and by moving the pen up and down and the robot around on a piece of paper it could draw pictures.

In FUZE BASIC the turtle is 'virtual', that is it exists only inside the computer's memory but it can be made to draw pictures on the screen. It should be possible for you to make a real turtle controlled by your FUZE!

There are two commands that control the virtual pen. The **PENUP** command lifts the pen from the paper and stops the turtle from drawing. As you might expect the **PENDOWN** command does the opposite. When the pen is down every time you move the turtle it will draw in the currently selected **COLOUR**. The pen is initially in the 'up' position.

To move the turtle you can either use an absolute position on the screen using the **MOVETO***(xpos,ypos)* command or you can make it move relative to the current point using the **MOVE***(pixels)* command. The turtle is initially placed in the middle of the screen.

The turtle has a current direction that it is facing. You can rotate the turtle in either direction using the **LEFT**(*angle*) or **RIGHT**(*angle*) commands. The *angle* will be in the current angle units (degrees is the default). You can also find or set the current turtle angle using the **TANGLE** built- in variable.

The turtle is initially facing towards the top of the screen. This example draws a spiral on the screen:

```
CLS
PENDOWN
FOR I=2 TO GWIDTH CYCLE
  MOVE(I)
  RIGHT(30)
REPEAT
UPDATE
END
```

GPIO

GPIO stands for General Purpose Input/Output and allows you to connect your FUZE to other electronic devices. These can either send signals to the FUZE or be controlled by it. The circuit board at the top left of your FUZE is connected to the GPIO of the Raspberry PI inside it.

I won't go into lots of detail here as it will be covered in the project cards, but I will go over the main items you will use.

The PINMODE procedure is used to set a particular GPIO pin to be either an output pin or an input pin. An output pin could be used to light up an LED and an input pin to read whether a button has been pressed. The syntax of PINMODE is as follows:

```
PINMODE(pinno,pinmode)
```

Where pinno is the number of the pin (from 0 to 16) and pinmode is 0 for input and 1 for output. So to set pin 7 to be an output pin you would do:

```
PINMODE(7, 1)
```

Then to set the output to on or off you use the DIGITALWRITE procedure which has the following syntax:

```
DIGITALWRITE(pinno,pinvalue)
```

Where pinno is the number of the pin and *pinvalue* is 0 for off and 1 for on. So to set pin 7 to be on you would use:

```
DIGITALWRITE(7,1)
```

So if there was an LED attached to pin 7 of the GPIO this would switch it on. The following program would cause an LED to flash on and off if it were attached to pin 2:

```
PINMODE(2,1)
CYCLE
  DIGITALWRITE(2,1)
  WAIT(1)
  DIGITALWRITE(2,0)
  WAIT(1)
REPEAT
END
```

If you set a pin to be input you use the **DIGITALREAD** function to read whether it is set to on or off. The following program would loop and do nothing until a button attached to pin 7 is pressed:

```
PINMODE(7,0)
REM Wait for button to be pushed
UNTIL DIGITALREAD(7) CYCLE
REPEAT
PRINT "Button Pushed"
END
```

Robot Arm

FUZE BASIC has built in support for the popular OWI-535 robot arm kit.

The robot arm comes in two forms, one with just a manual remote control and the other with a PC USB interface as well. You will need to have the USB interface in order to be able to control it from the FUZE. This is also available to be bought separately. You will also need a spare USB socket which may require a separate powered USB hub.

Note: If you have an older version of the FUZE boot image you may be missing some of the libraries required to connect to the robot arm and you may get an error. You can download the latest version of the image from the FUZE website.

Connect the robot arm to a spare USB socket and make sure that it is switched on before booting up your FUZE. Then start FUZE BASIC and to test that the communication is working type ARMLIGHT(1) and press the Enter key. This should switch on the LED in the middle of the arm's grippers. Typing ARMLIGHT(0) will switch it off again.

The arm has 5 motors and each of them can be controlled separately to move in a clockwise direction, by passing 1 as the argument, or anti-clockwise by passing -1. The ARMRESET command is used to stop them again.

For Example to move the Elbow motor for 2 seconds in a clockwise direction you would do the following:

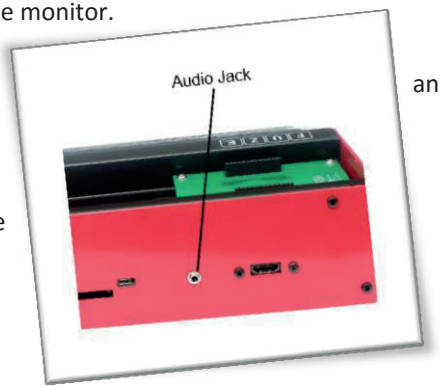
```
ARMELBOW(1)
WAIT(2)
ARMELBOW(0)
```

Music

FUZE BASIC can play various audio formats. You will need to have some additional hardware in order to hear the sound.

If your FUZE is connected to an HDMI monitor then by default the sound is played through the audio output of the monitor. If the monitor does not have built-in speakers you will need to connect some headphones or external powered speakers to the audio output of the monitor.

If you are not using HDMI monitor the sound will be directed to the audio jack on the back of the FUZE unit.



an

Even if you are using an HDMI monitor you can force it to use this audio jack by starting a terminal session (click on the Icon in the bottom left of the desktop and select *Accessories* and then *LXTerminal*) and typing the following command:

```
amixer cset numid=3 1
```

To switch back to automatic simply change the 1 to a 0 in the above command. If your FUZE unit is connected to the internet you can download a sample file by opening a terminal session and entering the following commands:

```
cd /home/pi/Desktop/FUZE
```

```
wget http://www.fuze.co.uk/FUZEBIN/media/takeoff.wav
```

Alternatively you can download the file on an internet connected device and transfer it to your FUZE using a memory stick. Enter the address above into an internet browser, then right-click and select *Save as* from the menu. Copy the file into the folder;

/home/pi/Desktop/FUZE on the FUZE.

To load a music file you use the LOADMUSIC function:

```
handle=LOADMUSIC("takeoff.wav")
```

And to play the music file you use the PLAYMUSIC function as follows:

```
PLAYMUSIC(handle,1)
```

This starts the music playing in the background which means that you can carry on issuing new commands while it plays. For example the PAUSEMUSIC function will pause the currently playing music and RESUMEMUSIC will start it playing again. The STOPMUSIC function will stop the music playing completely. You can also set the playback volume using SETMUSICVOL*(level)* where *level* is a number from 0 to 100%.

The second argument of the PLAYMUSIC functions tells it how many times to repeat the track before stopping. You can have 8 different music tracks loaded at the same time.

Sound Effects

Sound samples are similar to music playback except that you can play up to 4 tracks at the same time. This will let you add sound effects to your games. You can load up to 32 tracks and play any 4 at the same time in a similar manner to the music:

```
handle=LOADSAMPLE("filename.wav")
```

You can then playback a sample using;

```
PLAYSAMPLE(handle, channel, loops)
```

Here, the channel is 0, 1, 2 or 3. This lets you play up to 4 concurrent samples. The *loops* parameter is different to the *repeats* one in the PLAYMUSIC function. Here it means the number of times to loop the sample - zero means no loops which means play it ONCE.

You then have similar functions to the music playback ones to stop, pause or resume the sample as shown below. You can also control the volume of each individual channel using SETCHANVOL.

```
STOPCHAN(handle)
PAUSECHAN(handle)
RESUMECHAN(handle)
SETCHANVOL(channel, volume)
```

So now we can add some sound effects to the pacman animation that we created earlier. You can download two samples from our website by opening a terminal session and entering the following commands:

```
cd /home/pi/Desktop/FUZE
```

Wget http://www.fuze.co.uk/FUZEBIN/media/pacman_chomp.wav
wget http://www.fuze.co.uk/FUZEBIN/media/pacman_death.wav

Now try the following:

```
CLS
pacman=NEWSPRITE(3)
LOADSPRITE("pacman1.bmp",pacman,0)
LOADSPRITE("pacman2.bmp",pacman,1)
LOADSPRITE("pacman3.bmp",pacman,2)
ghost=NEWSPRITE(1)
LOADSPRITE("ghost1.bmp",ghost,0)
chomp=LOADSAMPLE("pacman_chomp.wav")
death=LOADSAMPLE("pacman_death.wav")
PLAYSAMPLE(chomp,0,10)
midY=GHEIGHT/2
FOR X=0 TO GWIDTH STEP 10 CYCLE
  PLOTSPRITE(pacman,X,midY,X MOD 3)
  PLOTSPRITE(ghost,GWIDTH-X-100,midY,0)
  IF SPRITECOLLIDE(pacman) <> -1 THEN
    HIDESPRITE(ghost)
    STOPCHAN(chomp)
    PLAYSAMPLE(death,1,0)
    BREAK
  ENDIF
  UPDATE
  WAIT(0.1)
REPEAT
END
```

You will need the sprite files that you created earlier. I will leave it as an exercise for you to add the pacman death animation sequence.

Synthesized Music

The simplest way to generate sound in FUZE BASIC is using the *tone* command which has the following syntax:

```
TONE(channel, volume, frequency, duration)
```

This plays a simple tone of the given *frequency* (1 to 5000Hz), *volume* (%) and *duration* (0.01 to 20 seconds) on the given *channel* (0 to 3).

You can play multiple tones by playing them one after the other and up to three tones can be played simultaneously on different channels. *Channel* 0 is white noise and the *frequency* has no bearing on the sound produced.

The demonstration program below reads the notes and durations using the DATA and READ commands to play a simple tune.

```
Channel=1
Volume=70
FOR Note=1 TO 5 CYCLE
  READ Frequency,Duration
  TONE(Channel,Volume,Frequency,Duration)
REPEAT
END
DATA 440,1
DATA 493,1
DATA 392,1
DATA 196,1
DATA 294,2
```

Here are the note frequencies for the Bass and Treble clefs.

Note	Bass	Treble
C	131	262
C#	139	277
D	147	294
D#	156	311
E	165	330
F	175	349
F#	185	370
G	196	392
G#	208	415
A	220	440
A#	233	466
B	247	493

Immediate Mode Commands

There are a number of immediate mode commands that we haven't covered yet.
By default FUZE BASIC will store your programs in a folder on your Desktop called **FUZE**:

In Immediate mode type the command **pwd** (print working directory) and you will be shown the path of the folder that will be used by default to load and save files:

```
Ready
>pwd
CWD: /home/pi/Desktop/FUZE
>
```

You can list the FUZE program files in this directory using the **DIR** command (you can also use the unix **ls** command to do the same thing). Note that this will only show FUZE BASIC program files, not images or other text files that are in the folder.

You can then load a program into memory using the **LOAD** command. To do this type **LOAD** followed by the program name but without the file extension (.fuze). Don't put quotation marks (") around the filename and it is case sensitive so the upper and lower case letters must match, otherwise you will get an error.

You can save changes to your program using the **SAVE** command on its own or indeed save it to a different filename by using **SAVE** *filename* where *filename* is the new name.

If you want to change the default folder where files are loaded and saved you can use the **CD** (change directory) command followed by the path of the folder.

You can list the program currently loaded using the **LIST** command or edit it using the **EDIT** command. If you want to start a new program you use the **NEW** command but as this deletes the program that is currently loaded make sure that you have saved any changes first!

To run the currently loaded program outside of the editor you can use the **RUN** command and finally to exit FUZE BASIC and return to the desktop you use the **EXIT** command.

Immediate mode Commands

CLEAR

Purpose
Clear variable memory.

Syntax
```
CLEAR
```

Description
Clears all variables and deletes all arrays. It also removes any active sprites from the screen. Stopped programs may not be continued after a CLEAR command.

Example
```
REM Immediate Mode
VARIABLE=10
PRINT VARIABLE
CLEAR
REM Unassigned variable error!
PRINT VARIABLE
```

Associated
NEW

CLS

Purpose
Clear the video display screen.

Syntax
```
CLS
```

Description
Clears the video display screen and places the cursor in the top left corner. The background is set to the current background (PAPER) colour.

Example
```
PAPER=WHITE
INK=BLACK
CLS
PRINT "Hello World"
END
```

Associated
CLS2, INK, PAPER, UPDATE

CONT

Purpose
Continue running a program that has been stopped.

Syntax
CONT

Description
Continues program execution after a STOP instruction. Variables are not cleared.

Example
```
REM Immediate mode
CONT
```

Associated
STOP

CONTINUE

Purpose
Continue a loop.

Syntax
CONTINUE

Description
Will cause the loop to re-start at the line containing the CYCLE instruction, continuing a FOR instruction and re-evaluating any WHILE or UNTIL instructions.

Example
```
REM Prints 1 to 10 but skips over 5
FOR I=1 to 10 CYCLE
  IF (I=5) THEN CONTINUE
  PRINT I
REPEAT
END
```

Associated
BREAK, CYCLE, CYCLE REPEAT, FOR REPEAT, REPEAT UNTIL, UNTIL REPEAT, WHILE REPEAT

DIR

Purpose
List FUZE BASIC program files in the current or specified directory.

Syntax
`DIR [directory]`

Description
Lists the program files in your current working directory or the optional specified *directory*. These will have the .fuze file extension.

Example
```
REM Immediate Mode
DIR
```

Associated
LOAD, SAVE

EDIT

Purpose
Edit the program in memory.

Syntax
`EDIT`

Description
Edits the program in memory using a full screen editor.

Example
```
REM Immediate mode
EDIT
```

EXIT

Purpose
Exit FUZE BASIC and return to the environment.

Syntax
`EXIT`

Description
Exit FUZE BASIC and return to the environment you started it in.

Example
```
REM Immediate mode
EXIT
```

LIST

Purpose
List the program stored in memory to the screen.

Syntax
```
LIST
```

Description
This lists the program stored in memory to the screen. You can pause the listing with the space-bar and terminate it with the escape key.

Example
```
REM Immediate Mode
LIST
```

LOAD

Purpose
Load a program into memory.

Syntax
LOAD *filename$*

Description
Loads in a program from the local non-volatile storage. As with SAVE, you need to supply the filename without any quotes. Do not include the .fuze file extension. Note, if your filename has spaces then you must enter it within quoatation marks. LOAD "my game" for example.

Example
```
REM Immediate Mode
LOAD demos/ball
RUN
```

Associated
SAVE

NEW

Purpose
Start a new program

Syntax
NEW

Description
Deletes the program in memory. There is no verification and once it's gone, it's gone. Remember to save first!

Example
```
REM Immediate Mode
NEW
REM List will now be empty
LIST
```

Associated
CLEAR

RUN

Purpose
Runs the program in memory.

Syntax
RUN

Description
Runs the program in memory. Note that using RUN will clear all variables.

Example
```
REM Immediate Mode
RUN
```

SAVE

Purpose
Saves your program to the local non-volatile storage.

Syntax
SAVE *filename$*

Description
Saves your program to the local non-volatile storage. The *filename$* is the name of the file you wish to save and may not contain spaces. If you have already saved a file, then you can subsequently execute SAVE without the filename and it will overwrite the last file saved. (This is reset when you load a new program or use the NEW command)

Example
```
REM Immediate Mode
SAVE testprog
```

Associated
LOAD

VERSION

Purpose
Print the current version of FUZE BASIC.

Syntax
VERSION

Description
Print the current version of FUZE BASIC.

Example
```
REM Immediate Mode
VERSION
```

Functions, Constants & Procedures

ABS

Purpose
Return the absolute value of the argument.

Syntax
positivenumber=ABS(*number*)

Description
Returns the absolute value of the supplied argument *number* i.e. If the argument is negative, make it positive.

Example
```
PRINT FN ElapsedYears(1966,2013)
PRINT FN ElapsedYears(2013,1966)
END
REM Return Number of years elapsed
REM Between two dates
DEF FN ElapsedYears(Year1, Year2)
=ABS(Year1-Year2)
```

Associated
SGN

ACOS

Purpose
Returns the arc cosine of the supplied argument.

Syntax
angle=ACOS(*cosine*)

Description
This is the inverse of the COS function returning the angle for a given cosine.

Example
```
PRINT "Angle with cosine = 0.5: "
DEG
REM 60 Degrees
PRINT "In Degrees:  ";ACOS(0.5)
RAD
REM PI/3 Radians
PRINT "In Radians: "; ACOS(0.5)
CLOCK
PRINT "In Minutes: "; ACOS(0.5)
END
```

Associated
ATAN, COS, SIN, TAN

ADVANCESPRITE

Purpose
Advances a sprite a specified amount

Syntax
ADVANCESPRITE(*sprite, distance*)

Description
Moves a sprite forward by the specified distance. The direction is set by the SETSPRITEANGLE function. This is very useful when used with rotating sprites.

Example
```
pic = NEWSPRITE( 1 )
LOADSPRITE("/usr/share/fuze/logo.bmp", pic,0)
PLOTSPRITE( pic, gWidth / 2, gHeight / 2, 0 )
FOR angle = 0 TO 360 CYCLE
SETSPRITEANGLE( pic, angle )
ADVANCESPRITE( pic, 5 )
UPDATE
REPEAT
END
```

Associated
PLOTSPRITE, SETSPRITEANGLE

ANALOGREAD

Purpose
Returns the value from an analog input.

Syntax
ANALOGREAD(0)

Description
Returns the a value of between 0 and 255 (0V & 3.3V) from the specified analog pin. There are four inputs, 0 to 3.

Example
```
REM Attach an LDR to 3.3V and analog pin 0
PRINT analogRead(0)
REM cover the LDR and run again.
END
```

Associated
ANALOGREAD, ANALOGWRITE

ANALOGWRITE

Purpose
Sends a value to the analog output.

Syntax
`ANALOGWRITE(0,value)`

Description

Sends a voltage between 0V and 3.3V (as a value of 0 to 255) to analog pin 0. There is one analog output.

Example
```
REM Attach an LED to analog OUT pin 0 (the longer leg) and the shorter leg to GND
LOOP
FOR volt= 0 TO 255 CYCLE
analogWrite(0,volt)
REPEAT
REPEAT
END
```

Associated
ANALOGREAD, ANALOGWRITE

ARMBODY

Purpose
Move the robot arm body.

Syntax
`ARMBODY(direction)`

Description
Activates the motor in the base of the robot arm according to the *direction* argument as follows:

1 Move clockwise

0 Stop

-1 Move anti-clockwise.

Example
```
REM Move the body for 2 seconds clockwise
ARMBODY(1)
WAIT(2)
ARMBODY(0)
END
```

Associated
ARMELBOW, ARMGRIPPER, ARMLIGHT, ARMRESET, ARMSHOULDER, ARMWRIST

ARMELBOW

Purpose
Move the robot arm elbow.

Syntax
ARMELBOW(*direction*)

Description
Activates the motor in the elbow of the robot arm according to the *direction* argument as follows:
1 Move clockwise
0 Stop
-1 Move anti-clockwise.

Example
```
REM Move the elbow for 1s anti-clockwise
ARMELBOW(-1)
WAIT(1)
ARMELBOW(0)
END
```

Associated
ARMBODY, ARMGRIPPER, ARMLIGHT, ARMRESET, ARMSHOULDER, ARMWRIST

ARMGRIPPER

Purpose
Open or close the robot arm gripper.

Syntax
ARMGRIPPER(*direction*)

Description
Activates the motor in the gripper of the robot arm according to the *direction* argument as follows:
1 Open gripper
0 Stop
-1 Close gripper

Example
```
REM Open the gripper for 1 seconds
ARMGRIPPER(1)
WAIT(1)
ARMGRIPPER(0)
END
```

Associated
ARMBODY, ARMELBOW, ARMLIGHT, ARMRESET, ARMSHOULDER, ARMWRIST

ARMLIGHT

Purpose
Switch the robot's LED on or off.

Syntax
ARMLIGHT(*switch*)

Description
If the *switch* argument is **1** then the LED is illuminated and if it is **0** it is switched off.

Example
```
REM Flash the LED with 1 second interval
CYCLE
  ARMLIGHT(1)
  WAIT(1)
  ARMLIGHT(0)
  WAIT(1)
REPEAT
```

Associated
ARMBODY, ARMELBOW, ARMGRIPPER, ARMRESET, ARMSHOULDER, ARMWRIST

ARMRESET

Purpose
Reset the robot arm.

Syntax
ARMRESET

Description
Stops any moving motors on the robot arm and switches off the LED.

Example
```
ARMLIGHT(1)
ARMELBOW(1)
WAIT(2)
ARMRESET
END
```

Associated
ARMBODY, ARMELBOW, ARMGRIPPER, ARMLIGHT, ARMSHOULDER, ARMWRIST

ARMSHOULDER

Purpose
Move the robot arm shoulder.

Syntax
ARMSHOULDER(*direction*)

Description
Activates the motor in the shoulder of the robot arm according to the *direction* argument as follows:
1 Move clockwise
0 Stop
-1 Move anti-clockwise.

Example
```
REM Move shoulder for 1s clockwise & back
ARMSHOULDER(1)
WAIT(1)
ARMSHOULDER(-1)
WAIT(1)
ARMSHOULDER(0)
END
```

Associated
ARMBODY, ARMELBOW, ARMGRIPPER, ARMLIGHT, ARMRESET, ARMWRIST

ARMWRIST

Purpose
Move the robot arm wrist.

Syntax
ARMWRIST(*direction*)

Description
Activates the motor in the wrist of the robot arm according to the *direction* argument as follows:
1 Move clockwise
0 Stop
-1 Move anti-clockwise.

Example
```
REM Move wrist for 1s clockwise & back
ARMWRIST(1)
WAIT(1)
ARMWRIST(-1)
WAIT(1)
ARMWRIST(0)
END
```

Associated
ARMBODY, ARMELBOW, ARMGRIPPER, ARMLIGHT, ARMRESET, ARMSHOULDER

ASC

Purpose
Return the ASCII code for a string character.

Syntax
asciicode=ASC(*string$*)

Description
Returns the ASCII value represented by the first character of *string$* e.g. "A" would return 65. It is the opposite of the CHR$ function.

Example
```
code=ASC("A")
PRINT "ASCII value of letter A is: "; code
PRINT "ASCII char of code 65: "; CHR$(code)
END
```

Associated
CHR$

ASIN

Purpose
Returns the arc sine of the supplied argument.

Syntax
angle = ASIN(*sine*)

Description
This is the inverse of the SIN function returning the angle for a given sine.

Example
```
PRINT "Angle with sine = 0.5: "
DEG
REM 30 Degrees
PRINT "In Degrees: ";ASIN(0.5)
RAD
REM PI/6 Radians
PRINT "In Radians: ";ASIN(0.5)
CLOCK
PRINT "In Minutes: ";ASIN(0.5)
END
```

Associated
ACOS, ATAN, COS, SIN, TAN

ATAN

Purpose
Returns the arc tangent of the supplied argument.

Syntax
angle=ATN(*tangent*)

Description
This is the inverse of the TAN function returning the angle for a given tangent.

Example
```
PRINT "Angle with  tangent equal to 1: "
DEG
REM 45 Degrees
PRINT "In Degrees: ";ATAN(1)
RAD
REM PI/4 Radians
PRINT "In Radians: ";ATAN(1)
CLOCK
PRINT "In Minutes: ";ATAN(1)
END
```

Associated
ACOS, ASIN, COS, SIN, TAN

BREAK

Purpose
Provide an early exit from a loop.

Syntax
BREAK

Description
Terminates a loop before the terminating condition is met.

Example
```
REM Loop until the space bar is pressed
PRINT "Press the space bar to continue"
CYCLE
  IF INKEY=32 THEN BREAK
REPEAT
END
```

Associated
CONTINUE, CYCLE, CYCLE REPEAT, FOR REPEAT, REPEAT UNTIL, UNTIL REPEAT, WHILE REPEAT

CHR$

Purpose
Returns the string character for the specified ASCII code.

Syntax
*character$*=CHR$(*asciicode*)

Description
Returns a one-character string consisting of the character corresponding to the ASCII code indicated by the *asciicode* argument. This is the opposite of the ASC function.

Example
```
PRINT "The following print the letter A"
PRINT CHR$(65)
PRINT CHR$(ASC("A"))
PRINT "The following print the number 3"
PRINT CHR$(51)
PRINT CHR$(ASC("0") + 3)
END
```

Associated
ASC

CIRCLE

Purpose
Draw a circle on the screen.

Syntax
CIRCLE(*centrex*,*centrey*,*radius*,*fill*)

Description
Draws a circle at position (*centrex*,*centrey*) with the specified *radius* in the current foreground COLOUR. The final parameter *fill* is either TRUE or FALSE, and specifies filled (TRUE) or outline (FALSE).

Example
```
CLS
PRINT "Draw a filled yellow circle ";
PRINT "In the centre of the screen"
COLOUR=yellow
CIRCLE(GWIDTH/2,GHEIGHT/2,100,TRUE)
UPDATE
END
```

Associated
ELLIPSE, RECT, TRIANGLE

CLEARKEYBOARD

Purpose
Clear all pending keyboard input.

Syntax
```
CLEARKEYBOARD
```

Description
Clears the keyboard input buffer.

Example
```
PRINT "Press space to continue"
WHILE NOT SCANKEYBOARD(scanSpace) CYCLE
REPEAT
CLEARKEYBOARD
END
```

Associated
SCANKEYBOARD

CLOCK

Purpose
Set angle units to minutes.

Syntax
```
CLOCK
```

Description
Switches the internal angle system to clock mode. There are 60 minutes in a full circle.

Example
```
PRINT "ATAN(1) = "
DEG
PRINT ATAN(1); " Degrees"
CLOCK
PRINT ATAN(1); " Minutes"
RAD
PRINT ATAN(1); " Radians"
END
```

Associated
DEG, RAD

CLONESPRITE

Purpose
Create a new sprite from an existing one.

Syntax
CLONESPRITE (*sprite handle*)

Description
Copies a sprite and its settings from an existing one (*sprite handle*) into a new one.

Example

```
sprite = newSprite (1)
loadSprite (“ballBlue.png”, sprite, 0)
sprite2 = cloneSprite (sprite)
CYCLE
   PlotSprite (sprite, 100, 100, 0)
   PlotSprite (sprite2, 200, 200, 0)
REPEAT
```

Associated
newSprite, loadSprite

CLOSE

Purpose
Close a file after use.

Syntax
CLOSE(*handle*)

Description
The CLOSE instruction closes the file specified by *handle* after use and makes sure all data is securely written to the storage medium.

Example

```
handle = OPEN("testfile.txt")
PRINT# handle, "Colin"
CLOSE(handle)
handle = OPEN("testfile.txt")
INPUT# handle, Name$
CLOSE(handle)
PRINT "Name: " + Name$
END
```

Associated
EOF, FFWD, INPUT#, OPEN, PRINT#, REWIND, SEEK

CLS

Purpose
Clear screen.

Syntax
```
CLS
```

Description
Completely clear the screen of text and graphics. Note, this does not clear sprites from the screen.

Example
```
PRINT “Hello World”
WAIT (1)
CLS
END
```

Associated
CLS, UPDATE

CLS2

Purpose
Clear screen without an update.

Syntax
```
CLS2
```

Description
Ideally suited to games and graphical programming. CLS2 clears the background or buffer screen and does not issue an update command. This means you can wipe the screen buffer, redraw on it and then issue an update. This ensures flicker free updates. It is also much faster than CLS.

Example
```
y = GHEIGHT / 2
radius = GWIDTH / 10
FOR x = 0 TO GWIDTH STEP radius CYCLE
CLS
  COLOUR = PINK
  CIRCLE (x, y, radius, 1)
  UPDATE
REPEAT
FOR x = GWIDTH TO 0 STEP -10 CYCLE
  CLS2
  COLOUR = YELLOW
  CIRCLE (x, y, radius, 1)
  UPDATE
REPEAT
```

Associated
CLS, UPDATE

COLOUR

Purpose
Set/Read the current graphics plot colour.

Syntax
COLOUR=*setcolour*

Description
Set/Read the current graphics plot colour as follows:

0 Black	10 Pink	20 Teal
1 DarkGrey	11 LightPink	21 Cyan
2 Grey	12 DarkGreen	22 Aqua
3 Silver	13 Green	23 LightBlue
4 LightGrey	14 BrightGreen	24 Brown
5 White	15 Olive	25 LightBrown
6 Maroon	16 Lime	26 Orange
7 Red	17 LightGreen	27 Gold
8 Purple	18 Navy	28 Yellow
9 Raspberry	19 Blue	29 LightYellow

Example
```
CLS
FOR c=0 TO 29 CYCLE
  COLOUR=c
  CIRCLE(50,GHEIGHT-c*40-50,20,TRUE)
  COLOUR=WHITE
  CIRCLE(50,GHEIGHT-c*40-50,20,FALSE)
  READ NameOfColour$
  HVTAB(5,c*2+2)
  PRINT NameOfColour$;" ";c
REPEAT
UPDATE
DATA "Black", "DarkGrey", "Grey", "Silver"
DATA "LightGrey", "White", "Maroon", "Red"
DATA "Purple", "Raspberry", "Pink", "LightPink"
DATA "DarkGreen", "Green", "BrightGreen", "Olive"
DATA "Lime", "LightGreen", "Navy", "Blue", "Teal"
DATA "Cyan", "Aqua", "LightBlue", "Brown"
DATA "LightBrown", "Orange", "Gold", "Yellow"
DATA "LightYellow"
END
```

Associated
INK, PAPER, RGBCOLOUR

COPYREGION

Purpose

Copy a region of the screen from one location to another.

Syntax

COPYREGION(*oldx*,*oldy*,*width*,*height*,*newX*,*newY*)

Description

This enables you to duplicate the contents of a section of the screen from one place to another. This could be used for example to create a background for a game by drawing an image and then duplicating it. The region to be copied is specified by the rectangle at coordinates (*oldx*,*oldy*) *width* pixels wide and *height* pixels high. The region is copied to coordinates (*newx*,*newy*)

Example

```
REM Draws a Brick Wall
COLOUR=RED
RECT(0,0,50,50,TRUE)
COLOUR=WHITE
LINE(0,50,50,50)
LINE(0,25,50,25)
LINE(0,25,0,50)
LINE(50,25,50,50)
LINE(25,0,25,25)
LINE(0,0,50,0)
FOR X=0 TO GWIDTH STEP 50 CYCLE
  FOR Y=0 TO GHEIGHT STEP 50 CYCLE
    COPYREGION(0,0,50,50,X,Y)
  REPEAT
REPEAT
UPDATE
END
```

Associated

GRABREGION

COS

Purpose
Returns the cosine of the given angle.

Syntax
cosine=COS(*angle*)

Description
Returns the cosine of the argument *angle*. This is the ratio of the side of a right angled triangle, that is adjacent to the angle, to the hypotenuse (the longest side).

Example
```
CLS
PRINT "Draw ellipse in screen centre"
DEG
FOR Angle=0 TO 360 CYCLE
  Xpos=100*COS(Angle)+GWIDTH / 2
  Ypos=50*SIN(Angle)+GHEIGHT / 2
  PLOT(Xpos, Ypos)
REPEAT
END
```

Associated
ACOS, ASIN, ATAN, SIN, TAN

CYCLE

Purpose
Defines the start of a block of code to be repeated.

Syntax
CYCLE

Description
Marks the start of a block of repeating code (called a loop). The number of times that the loop is executed depends on the command used before CYCLE or at the end of the loop.

Example
```
REM See Associated commands below.
```

Associated
BREAK, CONTINUE, CYCLE REPEAT, FOR REPEAT, REPEAT UNTIL, UNTIL REPEAT, WHILE REPEAT

CYCLE REPEAT

Purpose
Create an infinite loop.

Syntax
```
CYCLE
  {statements}
REPEAT
```

Description
Execute the *statements* again and again forever.
The BREAK command can be used to terminate the loop or control can be explicitly transferred to somewhere outside of the loop by commands like GOTO (not recommended).
Pressing the Esc key will also interrupt the loop (and program).

Example
```
REM Loop until the space bar is pressed
PRINT "Press the space bar to continue"
CYCLE
  IF INKEY = 32 THEN BREAK
REPEAT
END
```

Associated
BREAK, CONTINUE, CYCLE, FOR REPEAT, REPEAT UNTIL, UNTIL REPEAT, WHILE REPEAT

DATA

Purpose
Store constant data.

Syntax
```
DATA constant{,constant}
```

Description
Stores numerical and string constants for later retrieval using the READ command.

Example
```
REM Load the name of the days of the
REM week into a string array
DATA "Monday", "Tuesday", "Wednesday"
DATA "Thursday", "Friday", "Saturday"
DATA "Sunday"
DIM DaysOfWeek$(7)
FOR DayNo=1 TO 7 CYCLE
  READ DaysOfWeek$(DayNo)
REPEAT
FOR DayNo=1 TO 7 CYCLE
  PRINT "Day of the week number ";DayNo;
  PRINT " is ";DaysOfWeek$(DayNo)
REPEAT
END
```

Associated
READ, RESTORE

DATE$

Purpose
Return the current date.

Syntax
*todaysdate$* = DATE$

Description
This returns a string with the current date in the format: YYYY-MM-DD. For example: 2015-03-24.

Example
```
PRINT "Today is ";FN FormatDate(DATE$)
END
DEF FN FormatDate()
  DayNo=VAL(RIGHT$(DATE$, 2))
  MonthNo=VAL(MID$(DATE$, 5, 2))
  Year$=LEFT$(DATE$,4)
  SWITCH (DayNo MOD 10)
    CASE 1
     DaySuffix$ = "st"
    ENDCASE
    CASE 2
     DaySuffix$ = "nd"
    ENDCASE
    CASE 3
     DaySuffix$ = "rd"
    ENDCASE
    DEFAULT
     DaySuffix$ = "th"
    ENDCASE
  ENDSWITCH
  FOR I=1 TO MonthNo CYCLE
    READ MonthName$
  REPEAT
  Result$=STR$(DayNo)+DaySuffix$+" "
=Result$+MonthName$+" "+Year$
DATA "January","February","March","April"
DATA "May","June","July","August"
DATA "September","October","November"
DATA "December"
```

Associated
TIME$

DEF FN

Purpose
Create a user defined function.

Syntax
```
DEF FN name({parameter}{,parameter})
  {commands}
=value
```

Description
User defined functions are similar to user defined procedures except that they can return a value. This can be either a number or a character string.

Example
```
REM Function test - print squares
FOR I=1 TO 10 CYCLE
  x=FN square(I)
PRINT I; " squared is "; x
REPEAT
END
DEF FN square(num)
  LOCAL result
  result=num*num
=result
```

Associated
FN, LOCAL

DEF PROC

Purpose
Create a user defined procedure.

Syntax
```
DEF PROC name({parameter}{,parameter})
  {commands}
ENDPROC
```

Description
Allows you to create your own routines that can be called by their label. Once you have written a procedure to do a particular task you can copy it into other programs that require it. Procedures are usually defined after the END of the program.

Example
```
CLS
PROC Hexagon(200,200,100,Red)
UPDATE
END
DEF PROC Hexagon(x,y,l,c)
  PENUP
  MOVETO(x-l*COS(30),y+l/2)
  COLOUR = c
  PENDOWN
  FOR I=1 to 6 CYCLE
   RIGHT(60)
   MOVE(l)
  REPEAT
ENDPROC
```

Associated
LOCAL, PROC

DEFCHAR

Purpose
Define a new font character.

Syntax
DEFCHAR(*char*,*line1 … line10*)

Description
Create a user defined font character. The *char* parameter is the position of the character within the font (0-255) e.g. 65 is the capital letter A in ASCII. The character consists of 10 lines with 8 pixels in each line. These are set by the corresponding bits in each of the *line* parameters. So for example decimal 170 (binary 10101010) would set alternate pixels on the corresponding line of the character.

Example
```
REM Define Chessboard Character
FONTSIZE(10)
DEFCHAR(2,0,85,170,85,170,85,170,85,170,0)
PRINT CHR$(2)
END
```

Associated
CHR$

DEG

Purpose
Set angle units to degrees.

Syntax
DEG

Description
Switches the internal angle system to degree mode. There are 360 degrees in a full circle.

Example
```
REM Draw an ellipse in the screen centre
CLS
DEG
FOR Angle = 0 TO 360 CYCLE
  Xpos=100*COS(Angle)+GWIDTH / 2
  Ypos=50*SIN(Angle)+GHEIGHT / 2
  PLOT(Xpos, Ypos)
REPEAT
END
```

Associated
CLOCK, RAD

DIGITALREAD

Purpose
Read the state of a digital pin on the Raspberry Pi.

Syntax
pinvalue=DIGITALREAD(*pinno*)

Description
Reads the state of a digital pin on the Raspberry Pi. You may need to set the pin mode beforehand to make sure it's configured as an input device. It will return TRUE or FALSE to indicate an input being high or low respectively.

Example
```
REM Set pin 12 to input
PINMODE(12,0)
REM Wait for button to be pushed
UNTIL DIGITALREAD(12) CYCLE
REPEAT
PRINT "Button Pushed"
END
```

Associated
DIGITALWRITE, PINMODE, PWMWRITE

DIGITALWRITE

Purpose
Set the state of a digital pin on the Raspberry Pi.

Syntax
DIGITALWRITE(*pinno*,*pinvalue*)

Description
This procedure sets a digital pin to the supplied value - 0 for off or 1 for on. As with DigitalRead, you may need to set the pin mode (to output) beforehand.

Example
```
REM Flash LED attached to pin2 of GPIO
REM  Set pin 2 to output mode
PINMODE(2,1)
CYCLE
  REM Set output High (on)
  DIGITALWRITE(2,1)
  WAIT(1)
  REM Set output Low (off)
  DIGITALWRITE(2,0)
  WAIT(1)
REPEAT
END
```

Associated
DIGITALREAD, PINMODE, PWMWRITE

DIM

Purpose

Dimension an array of variables.

Syntax

DIM *variable*(dimension{,dimension})

Description

Creates an indexed variable with one or more dimensions. The variable can be either a numeric or character string type (they cannot hold mixed values). The index is a number from 0 to the size of the dimension.
Associative arrays (sometimes called maps) are another way to refer to the individual elements of an array. In the example below we use a number, however strings are also allowed. They can be multi-dimensional and you can freely mix numbers and strings for the array indices.

Example

```
REM Initialise the squares of a chess
REM board to black or white
DIM ChessBoard(8,8)
Count=0
FOR Row=1 TO 8 CYCLE
  FOR Col=1 TO 8 CYCLE
    Count=Count+1
    IF Count MOD 2=1 THEN
      ChessBoard(Row,Col) = Black
    ELSE
      ChessBoard(Row,Col) = White
    ENDIF
  REPEAT
REPEAT
PRINT ChessBoard(1,4)
END
```

DRCANALOGREAD

Purpose
Read an analog channel on a DRC device.

Syntax
voltage=DRCANALOGREAD(*handle*,*pin*)

Description
This function reads an analog channel on a DRC compatible device specified by *handle* and returns the result. The value returned will depend on the hardware you're connected to - for example the Arduino will return a number from 0 to 1023 representing an input voltage between 0 and 5 volts. Other devices may have different ranges.

Example
```
arduino=DRCOPEN("/dev/ttyUSB0")
REM Get voltage on pin 4
voltage=DRCANALOGREAD(arduino, 4)/1023*5
PRINT "Voltage= "; voltage
DRCCLOSE(arduino)
END
```

Associated
DRCCLOSE, DRCDIGITALREAD, DRCDIGITALWRITE, DRCOPEN, DRCPINMODE, DRCPWMWRITE

DRCCLOSE

Purpose
Close a connection to a DRC compatible device.

Syntax
DRCCLOSE(*handle*)

Description
This closes a connection to a DRC device and frees up any resources used by it. It's not strictly necessary to do this when you end your program, but it is considered good practice.

Example
```
arduino = DRCOPEN("/dev/ttyUSB0")
REM Set pin 12 to input
DRCPINMODE(arduino, 12, 0)
CYCLE
REPEAT UNTIL DRCDIGITALREAD(arduino, 12)
PRINT "Button Pushed"
DRCCLOSE(arduino)
END
```

Associated
DRCANALOGREAD, DRCDIGITALREAD, DRCDIGITALWRITE, DRCOPEN, DRCPINMODE, DRCPWMWRITE

DRCDIGITALREAD

Purpose
Read the state of a digital pin on a remote DRC device.

Syntax
state=DRCDIGITALREAD(*handle*,*pin*)

Description
This function allows you to read the state of a digital pin on a DRC device specified by *handle*. You may need to set the pin mode beforehand to make sure it's configured as an input device. It will return TRUE or FALSE to indicate an input being high or low respectively.

Example
```
arduino = DRCOPEN("/dev/ttyUSB0")
REM Set pin 12 to input
DRCPINMODE(arduino, 12, 0)
CYCLE
REPEAT UNTIL DRCDIGITALREAD(arduino, 12)
PRINT "Button Pushed"
DRCCLOSE(arduino)
END
```

Associated
DRCANALOGREAD, DRCCLOSE, DRCDIGITALWRITE, DRCOPEN, DRCPINMODE, DRCPWMWRITE

DRCDIGITALWRITE

Purpose
Set a digital pin on a remote DRC device to the supplied value.

Syntax
DRCDIGITALWRITE(*handle*,*pin*,*value*)

Description
This procedure sets a digital *pin* on a DRC device specified by *handle* to the supplied *value* - 0 for off or 1 for on. As with DrcDigitalRead, you may need to set the pin mode beforehand.

Example
```
arduino=DRCOPEN("/dev/ttyUSB0")
REM  Set pin 2 to output mode
DRCPINMODE(arduino, 2, 1)
REM Set Output High (on)
DRCDIGITALWRITE(arduino, 2, 1)
REM Pause for 1 second
WAIT(1)
REM Set output Low (off)
DRCDIGITALWRITE(arduino, 2, 0)
DRCCLOSE(arduino)
END
```

Associated
DRCANALOGREAD, DRCCLOSE, DRCDIGITALREAD, DRCOPEN, DRCPINMODE, DRCPWMWRITE

DRCOPEN

Purpose
Open a connection to a DRC compatible device.

Syntax
handle=DRCOPEN(*drcdevice*)

Description
This opens a connection to a DRC compatible device and makes it available for our use. It takes the name of the device as an argument and returns a number (the handle) of the device. We can use this handle to reference the device and allow us to open several devices at once. Some implementations may have IO devices with fixed names.

Example
```
arduino = DRCOPEN("/dev/ttyUSB0")
REM Set pin 12 to input
DRCPINMODE(arduino, 12, 0)
CYCLE
REPEAT UNTIL DRCDIGITALREAD(arduino, 12)
PRINT "Button Pushed"
DRCCLOSE(arduino)
END
```

Associated
DRCANALOGREAD, DRCCLOSE, DRCDIGITALREAD, DRCDIGITALWRITE, DRCPINMODE, DRCPWMWRITE

DRCPINMODE

Purpose
Configure the mode of a pin on a remote DRC device.

Syntax
DRCPINMODE(*handle*,*pin*,*mode*)

Description
This configures the mode of a pin on the DRC device specified by *handle*. It takes an argument which specifies the *mode* of the specified *pin* - input, output or PWM output. Other modes may be available, depending on the device and its capabilities. Note that not all devices support all functions. The modes are:

0 pinINPUT

1 pinOUTPUT

2 pinPWM

Example
```
arduino = DRCOPEN("/dev/ttyUSB0")
REM Set pin 12 to input
DRCPINMODE(arduino, 12, 0)
CYCLE
REPEAT UNTIL DRCDIGITALREAD(arduino, 12)
PRINT "Button Pushed"
DRCCLOSE(arduino)
END
```

Associated
DRCANALOGREAD, DRCCLOSE, DRCDIGITALREAD, DRCDIGITALWRITE, DRCOPEN, DRCPWMWRITE

DRCPWMWRITE

Purpose
Output a PWM waveform on the selected pin of a DRC device.

Syntax
DRCPWMWRITE(*handle*,*pin*,*value*)

Description
This procedure outputs a PWM waveform on the specified *pin* of a DRC compatible device specified by *handle*. The pin must be configured for PWM mode beforehand, and depending on the device you are using, then not all pins on a device may support PWM mode. The *value* set should be between 0 and 255.

Example
```
arduino=DRCOPEN("/dev/ttyUSB0")
REM Set pin 11 to PWM output mode
DRCPINMODE(arduino, 11, 2)
DRCPWMWRITE(arduino, 11, 200)
DRCCLOSE(arduino)
END
```

Associated
DRCANALOGREAD, DRCCLOSE, DRCDIGITALREAD, DRCDIGITALWRITE, DRCOPEN, DRCPINMODE

ELLIPSE

Purpose
Draw an ellipse on the screen.

Syntax
ELLIPSE(*xpos*,*ypos*,*xradius*,*yradius*,*fill*)

Description
Draws an ellipse centred at position (*xpos*,*ypos*) with the specified *xradius* and *yradius* in the current foreground COLOUR. The final parameter *fill* is either TRUE or FALSE, and specifies filled (TRUE) or outline (FALSE).

Example
```
CLS
REM Draw a filled red ellipse at
REM location 200,200
COLOUR=red
ELLIPSE(200,200,100,50,TRUE)
UPDATE
END
```

Associated
CIRCLE, RECT, TRIANGLE

ELSE

Purpose
Execute statement(s) when a tested condition is False.

Syntax
```
IF condition THEN
{statements}
ELSE
{statements}
ENDIF
```

Example
```
Number = 13
IF Number MOD 2 = 0 THEN
  PRINT "Number is Even"
ELSE
  PRINT "Number is Odd"
ENDIF
END
```

Associated
ENDIF, IF THEN

END

Purpose
End program execution.

Syntax
```
END
```

Description
Program execution is ended. Programs must terminate with the END or STOP commands or an error will occur.

Example
```
PRINT "Hello World"
END
```

ENDIF

Purpose
Terminate a multiline conditional statement.

Syntax
```
IF condition THEN
{statements}
ENDIF
```

Description
We can extend the IF statement over multiple lines, if required. The way you do this is by making sure there is nothing after the THEN statement and ending it all with the ENDIF statement.

Example
```
DayOfWeek = 5
IF DayOfWeek < 6 THEN
  PRINT "It is a Weekday"
  PRINT "Go to Work!"
ENDIF
END
```

Associated
ELSE, IF THEN

ENDPROC

Purpose
Defines the end of a PROCedure

Syntax
```
ENDPROC
```

Description
End a PROCedure and return to the next command after the procedure was called.

Example
```
CLS
PROC hello
END
DEF PROC hello
PRINTAT (10,10); "Hello"
ENDPROC
```

Associated
PROC

ENVELOPE

Purpose

Emulate the BBC BASIC sound envelope command.

Syntax

ENVELOPE(*N*,*T*,*PI1*,*P12*,*PI3*,*PN1*,*PN2*,*PN3*,*AA*,*AD*,*AS*,*AR*,*ALA*,*ALD*)

Description

NOTE: This is an experimental function. It might not perform entirely as expected. It is also prone to crashing if incorrect values are used. Use with caution!

N 1 to 8 Envelope number.

T 0 to 127 Length of each step in hundredths of a second. Add 128 to cancel auto repeat of the pitch envelope.

PI1 −128 to 127 Pitch change per step in section 3
PI2 −128 to 127 Change of pitch per step in section 2
PI3 −128 to 127 Change of pitch per step in section 1
PN1 0 to 255 Number of steps in section 1
PN2 0 to 255 Number of steps in section 2
PN3 0 to 255 Number of steps in section 3
AA −127 to 127 Change of attack amplitude per step
AD −127 to 127 Change of decay amplitude per step
AS −127 to 0 Change of sustain amplitude per step
AR −127 to 0 Change of release amplitude per step
ALA 0 to 126 Level at end of the attack phase
ALD 0 to 126 Level at end of the decay phase

Example

```
ENVELOPE(1,2,-
2,10,1,80,40,40,127,0,0,0,126,126)
SOUND(1, 1, 53, 64)
END
```

Associated

SOUND

EOF

Purpose
Return true if the end of an input file has been reached.

Syntax
endoffile=EOF(*handle*)

Description
The EOF function returns a TRUE or FALSE indication of the state of the file pointer when reading the file. It is an error to try to read past the end of the file, so if you are reading a file with unknown data in it, then you must check at well defined intervals (e.g. Before each INPUT#).

Example
```
handle=OPEN("eoftest.txt")
FOR r = 0 TO 10 CYCLE
  PRINT# handle, "Record "; r
REPEAT
CLOSE (handle)
handle = OPEN("eoftest.txt")
WHILE NOT EOF (handle) CYCLE
  INPUT# handle, record$
  PRINT record$
REPEAT
CLOSE (handle)
END
```

Associated
CLOSE, FFWD, INPUT#, OPEN, PRINT#, REWIND, SEEK

EXP

Purpose
Return the exponential value of the specified number.

Syntax
exponential=EXP(*number*)

Description
Returns the exponential value of the specified *number*. This is e to the power of *number* where e is the exponential constant (approximately 2.718281828). The exponential function arises whenever a quantity grows or decays at a rate proportional to its current value. This is the opposite of the LOG function i.e. EXP(LOG(X)) = X

Example
```
REM prints 2.718281828
PRINT EXP(1)
REM prints 22026.46579
PRINT EXP(10)
REM prints 10
PRINT LOG(EXP(10))
END
```

Associated
LOG

FADEOFF

Purpose
Fade the display from light to dark

Syntax
FADEOFF

Description
FADEOFF initiates a fade from light to dark. The entire screen display is affected.

Example
```
PAPER=0
INK=1
CLS
PRINT "LOADING..."
FADEOFF
WHILE FADING = true CYCLE
    UPDATE
REPEAT
FADEON
WHILE FADING  =  true CYCLE
    UPDATE
REPEAT
END
```

Associated
FADEON, FADING

FADEON

Purpose
Fade the display from dark to light

Syntax
FADEON

Description
FADEON initiates a fade from dark to light. The entire screen display is affected.

Example
```
PAPER=0
INK=1
CLS
PRINT "LOADING..."
FADEOFF
WHILE FADING = true CYCLE
    UPDATE
REPEAT
FADEON
WHILE FADING  =  true CYCLE
    UPDATE
REPEAT
END
```

Associated
FADEOFF, FADING

FADING

Purpose
Check if the display is fading

Syntax
```
FADING
```

Description
Returns either TRUE (if fade in progress) or FALSE (no fade active)

Example
```
PAPER=0
INK=1
CLS
PRINT "LOADING..."
FADEOFF
WHILE FADING = true CYCLE
    UPDATE
REPEAT
FADEON
WHILE FADING  =  true CYCLE
    UPDATE
REPEAT
END
```

Associated
FADEON, FADING

FALSE

Purpose
Represent the logical "false" value.

Syntax
```
FALSE
```

Description
Represents a Boolean value that fails a conditional test. It is equivalent to a numeric value of 0.

Example
```
condition = FALSE
IF condition = FALSE THEN
  PRINT "Condition is FALSE"
ENDIF
IF NOT condition THEN
  PRINT "Condition is FALSE"
ENDIF
PRINT "Condition= ";condition
END
```

Associated
TRUE

FFWD

Purpose
Move the file pointer to the end of a file.

Syntax
FFWD(*handle*)

Description
Move the file pointer back to the end of the file specified by *handle*. If you want to append data to the end of an existing file, then you need to call FFWD before writing the data.

Example
```
handle=OPEN("ffwdtest.txt")
PRINT# handle, "First Line"
CLOSE (handle)
handle = OPEN ("ffwdtest.txt")
FFWD (handle)
PRINT# handle, "Appended line"
CLOSE (handle)
handle = OPEN("ffwdtest.txt")
WHILE NOT EOF (handle) CYCLE
  INPUT# handle, record$
  PRINT record$
REPEAT
CLOSE (handle)
END
```

Associated
CLOSE, EOF, INPUT#, OPEN, PRINT#, REWIND, SEEK

FN

Purpose
Call a user defined function.

Syntax
result=FN *name*({*argument*}{,*argument*})

Description
Calls the specified user defined function called *name* with the specfied *arguments*. The returned *result* can then be used by the program. Once the function has been executed control returns to the command following.

Example
```
PRINT FN SphereVolume(10)
END
REM Function calculate volume of a sphere
REM with radius r
DEF FN SphereVolume(r)
= (4/3)*PI*r*r*r
```

Associated
DEF FN

FONTSIZE

Purpose
Scale the text font.

Syntax
```
FONTSIZE(scale)
```

Description
Change the size of the text font.

Example
```
FOR S=1 TO 10 CYCLE
  FONTSIZE(S)
  PRINT "Hello World"
REPEAT
END
```

Associated
PRINTAT, LOADFONT

FOR REPEAT

Purpose
Loop a specified number of times using a counter.

Syntax
```
FOR count=start TO end [STEP step] CYCLE
  statements
REPEAT
```

Description
The *count* variable is initially set to *start* and changes by *step* each time around the loop until *count* is greater than or equal to *end*. The optional *step*, which defaults to 1 may be less than zero to count backwards. The end of the loop is indicated using the REPEAT.

Example
```
REM year into a string array
DATA "January", "February", "March"
DATA "April", "May", "June"
DATA "July", "August", "September"
DATA "October", "November", "December"
DIM Months$(12)
FOR Month = 1 TO 12 CYCLE
  READ Months$(Month)
REPEAT
PRINT "The seventh month is ";Months$(7)
END
```

Associated
BREAK, CONTINUE, CYCLE, CYCLE REPEAT, REPEAT UNTIL, UNTIL REPEAT, WHILE REPEAT

FREEIMAGE

Purpose
Release an image from memory

Syntax
FREEIMAGE(*handle*)

Description
Frees up the memory space taken up by a stored image

Example
```
handle = LOADIMAGE( "sprite1.bmp" )
PLOTIMAGE( handle, 0, 0 )
UPDATE
FREEIMAGE( handle )
handle = LOADIMAGE( "sprite2.bmp" )
PLOTIMAGE( handle, 100, 100 )
UPDATE
END
```

Associated
LOADIMAGE, PLOTIMAGE

FULLSCREEN

Purpose
Sets the display to full screen mode.

Syntax
FULLSCREEN={TRUE/FALSE}

Description
Switches between full screen or windowed mode. Note this does not set the resolution to the screen display mode so unless you set the mode manually you will get a border.

Example
```
SETMODE(800,600)
FULLSCREEN=0
COLOUR=RED
RECT(0,0,GWIDTH, GHEIGHT,0)
UPDATE
WAIT(2)
FULLSCREEN=1
WAIT(2)
END
```

Associated
SETMODE

GET

Purpose
Get a single character code from the keyboard.

Syntax
asciicode=GET

Description
This pauses program execution and waits for you to type a single character on the keyboard, then returns the value of the key pressed as a numeric variable (ASCII).

Example
```
PRINT "Press a key"
key = GET
PRINT "ASCII Value of key = "; key
END
```

Associated
GET$, INKEY

GET$

Purpose
Get a single character from the keyboard.

Syntax
*key$* = GET$

Description
This pauses program execution and waits for you to type a single character on the keyboard, then returns the key as a string variable.

Example
```
PRINT "Press a key"
key$ = GET$
PRINT "You Pressed Key: "; key$
END
```

Associated
GET, INKEY

GETIMAGEH

Purpose
Get the pixel height of a loaded image.

Syntax
GETIMAGEH(*handle*)

Description
Gets the height in pixels of a loaded image

Example
```
REM Centre image on screen
logo=LOADIMAGE("/usr/share/fuze/logo.bmp")
imageW=GETIMAGEW(logo)
imageH=GETIMAGEH(logo)
X=(GWIDTH-imageW)/2
Y=(GHEIGHT-imageH)/2
PLOTIMAGE(logo,X,Y)
UPDATE
END
```

Associated
GETIMAGEW, LOADIMAGE, PLOTIMAGE

GETIMAGEW

Purpose
Get the pixel width of a loaded image.

Syntax
GETIMAGEW(*handle*)

Description
Gets the width in pixels of an image previously loaded using LOADIMAGE (using the *handle* returned by LOADIMAGE).

Example
```
REM Centre image on screen
logo=LOADIMAGE("/usr/share/fuze/logo.bmp")
imageW=GETIMAGEW(logo)
imageH=GETIMAGEH(logo)
X=(GWIDTH-imageW)/2
Y=(GHEIGHT-imageH)/2
PLOTIMAGE(logo,X,Y)
UPDATE
END
```

Associated
GETIMAGEH, LOADIMAGE, PLOTIMAGE

GETMOUSE

Purpose
Read values from an attached mouse

Syntax
GETMOUSE(*xpos*,*ypos*,*buttons*)

Description
This reads values for the current state of the mouse. *xpos* is the horizontal mouse position, *ypos* is the vertical position and *buttons* is the state of the mouse buttons. You can test whether the left button has been pressed by using the logical & operator to see if bit 0 of the *buttons* value is set: *buttons* & 1 will be TRUE. Likewise if the right button is pressed then bit 3 will be set and *buttons* & 4 will be TRUE.

Example
```
CLS
MOUSEON
CYCLE
  GETMOUSE(x,y,b)
  LINETO(x,y)
  UPDATE
  REM Cycle colour if left button pressed
  IF b & 1 THEN
    COLOUR = COLOUR MOD 16 + 1
  ENDIF
  REM Exit if right button pressed
REPEAT UNTIL b & 4
MOUSEOFF
END
```

Associated
MOUSEOFF, MOUSEON, MOUSEX, MOUSEY, SETMOUSE

GETPIXEL

Purpose
Return the colour of the specified pixel.

Syntax
colour = GETPIXEL(*xpos*, *ypos*)

Description
This returns the internal colour code (0-15) of the pixel at the specified point (*xpos*,*ypos*). This is for the "named" colours - e.g. Red, Green etc. It returns -1 if the pixel colour is not a standard colour - in which case, you need to use GETPIXELRGB.

Example
```
CLS
COLOUR = RED
Xpos = GWIDTH / 2
Ypos = GHEIGHT / 2
PLOT(Xpos, Ypos)
PRINT GETPIXEL(Xpos, Ypos)
END
```

Associated
GETPIXELRGB

GETPIXELRGB

Purpose
Return the RGB colour of the specified pixel.

Syntax
RGBcolour = GETPIXELRGB(*xpos*, *ypos*)

Description
This returns the RGB colour of the pixel at the specified point (*xpos*,*ypos*). This will return a 24-bit value.

Example
```
CLS
RGBCOLOUR(49, 101, 206)
Xpos = GWIDTH / 2
Ypos = GHEIGHT / 2
CIRCLE(Xpos, Ypos, 50, TRUE)
pixel = GETPIXELRGB(Xpos, Ypos)
PRINT "RGB = ";  pixel
PRINT "Red = ";  (pixel >> 16) & 0xFF
PRINT "Green = ";  (pixel >> 8) & 0xFF
PRINT "Blue = ";  (pixel >> 0) & 0xFF
END
```

Associated
GETPIXEL

GETSPRITEANGLE

Purpose
Returns a sprite's current angle

Syntax
GETSPRITEANGLE(*spriteIndex*)

Description
Returns the angle of the sprite with the specified *spriteIndex*.

Example
```
sprite = newSprite (1)
loadSprite ("logo.png", sprite, 0)
angle = 0
plotSprite (sprite, gWidth / 2, gHeight / 2, 0)
CYCLE
cls2
  setSpriteAngle (sprite, angle)
  angle = angle + 1
  if angle>359 THEN angle=0
  printat(0,0);getspriteangle(sprite)
  UPDATE
REPEAT
```

Associated
GETSPRITEW, HIDESPRITE, LOADSPRITE, NEWSPRITE, PLOTSPRITE, SETSPRITETRANS, SPRITECOLLIDE, SPRITECOLLIDEPP

GETSPRITEH

Purpose
Get the pixel height of a sprite.

Syntax
GETSPRITEH(*spriteIndex*)

Description
Returns the height in pixels of the sprite with the specified *spriteIndex*.

Example
```
REM Centre sprite on the screen
index=NEWSPRITE(1)
fuzelogo$="/usr/share/fuze/logo.bmp"
LOADSPRITE(fuzelogo$,index,0)
spriteW=GETSPRITEW(index)
spriteH=GETSPRITEH(index)
X=(GWIDTH-spriteW)/2
Y=(GHEIGHT-spriteH)/2
PLOTSPRITE(index,X,Y,0)
UPDATE
END
```

Associated
GETSPRITEW, HIDESPRITE, LOADSPRITE, NEWSPRITE, PLOTSPRITE, SETSPRITETRANS, SPRITECOLLIDE, SPRITECOLLIDEPP

GETSPRITEW

Purpose
Get the pixel width of a sprite.

Syntax
GETSPRITEW(*spriteIndex*)

Description
Returns the width in pixels of the sprite with the specified *spriteIndex*.

Example
```
REM Centre sprite on the screen
index=NEWSPRITE(1)
fuzelogo$="/usr/share/fuze/logo.bmp"
LOADSPRITE(fuzelogo$,index,0)
spriteW=GETSPRITEW(index)
spriteH=GETSPRITEH(index)
X=(GWIDTH-spriteW)/2
Y=(GHEIGHT-spriteH)/2
PLOTSPRITE(index,X,Y,0)
UPDATE
END
```

Associated
GETSPRITEH, HIDESPRITE, LOADSPRITE, NEWSPRITE, PLOTSPRITE, SETSPRITETRANS, SPRITECOLLIDE, SPRITECOLLIDEPP

GETSPRITEX

Purpose
Get the X position of a sprite.

Syntax
GETSPRITEX(*spriteIndex*)

Description
Returns the X position in pixels of the sprite with the specified *spriteIndex*.

Example
```
updateMode = 0
sprite = newSprite (1)
loadSprite ("logo.png", sprite, 0)
CYCLE
    CLS2
    plotSprite (sprite, gWidth / 2, gHeight / 2, 0)
    PRINT getSpriteX (sprite)
    PRINT getSpriteY (sprite)
    UPDATE
REPEAT
```

Associated
GETSPRITEW, HIDESPRITE, LOADSPRITE, NEWSPRITE, PLOTSPRITE, SETSPRITETRANS, SPRITECOLLIDE, SPRITECOLLIDEPP

GETSPRITEY

Purpose
Get the Y position of a sprite.

Syntax
GETSPRITEY(*spriteIndex*)

Description
Returns the Y position in pixels of the sprite with the specified *spriteIndex*.

Example
```
updateMode = 0
sprite = newSprite (1)
loadSprite ("logo.png", sprite, 0)
CYCLE
    CLS2
    plotSprite (sprite, gWidth / 2, gHeight / 2, 0)
    PRINT getSpriteX (sprite)
    PRINT getSpriteY (sprite)
    UPDATE
REPEAT
```

Associated
GETSPRITEH, HIDESPRITE, LOADSPRITE, NEWSPRITE, PLOTSPRITE, SETSPRITETRANS, SPRITECOLLIDE, SPRITECOLLIDEPP

GHEIGHT

Purpose
Find the current height of the display.

Syntax
height=GHEIGHT

Description
This can be read to find the current height of the display in either high resolution or low resolution pixels.

Example
```
REM Draw a circle in the centre of the
screen
CLS
COLOUR = blue
CIRCLE(GWIDTH/2,GHEIGHT/2,50,TRUE)
UPDATE
END
```

Associated
GWIDTH, ORIGIN, SETMODE

GRABREGION

Purpose
Grab a region of the screen to a temporary buffer

Syntax
handle = GRABREGION(*x, y, width, height*)

Description
Grab a region of the screen with x and y as the location and with *width* and *height* in pixels. The region can be recalled by its handle and pasted using PLOTIMAGE.

Example
```
FOR n = 0 TO 15 CYCLE
RECT(0,n*GHEIGHT/16, GWIDTH, GHEIGHT/16,1)
REPEAT
handle=GRABREGION(0,0,200,200)
CLS
PLOTIMAGE(handle, GWIDTH/2, GHEIGHT/2)
UPDATE
END
```

Associated
COPYREGION, FREEIMAGE, LOADIMAGE, PLOTIMAGE, SAVEREGION, SAVESCREEN, SCROLLDOWN, SCROLLLEFT, SCROLLRIGHT, SCROLLUP

GWIDTH

Purpose
Returns the current width of the display.

Syntax
width=GWIDTH

Description
This can be read to find current width of the display in either high resolution or low resolution pixels.

Example
```
REM Draw a circle in the centre of the
screen
CLS
COLOUR = blue
CIRCLE(GWIDTH/2,GHEIGHT/2,50,TRUE)
UPDATE
END
```

Associated
GHEIGHT, ORIGIN, SETMODE

HIDESPRITE

Purpose
Remove a sprite from the screen.

Syntax
HIDESPRITE(*spriteindex*)

Description
This removes the sprite at the specified *spriteindex* from the screen. You do not have to erase a sprite from the screen when you move it, just call PLOTSPRITE with the new coordinates.

Example
```
CLS
fuzelogo$="/usr/share/fuze/logo.bmp"
s1=NEWSPRITE(1)
s2=NEWSPRITE(1)
LOADSPRITE(fuzelogo$,s1,0)
LOADSPRITEfuzelogo$,s2,0)
PLOTSPRITE(s1,100,100,0)
PLOTSPRITE(s2,200,200,0)
UPDATE
WAIT(2)
REM Remove a sprite from the screen
HIDESPRITE(s2)
UPDATE
END
```

Associated
GETSPRITEH, GETSPRITEW, LOADSPRITE, NEWSPRITE, PLOTSPRITE, SPRITECOLLIDE, SPRITECOLLIDEPP, SPRITEOUT

HLINE

Purpose
Draw a horizontal line.

Syntax
HLINE(*xpos1, xpos2, ypos*)

Description
Draws a horizontal line on row y, from column xpos1 to column xpos2.

Example
```
CLS
COLOUR=red
FOR ypos=0 TO GHEIGHT STEP 100 CYCLE
  HLINE(0,GWIDTH,ypos)
REPEAT
UPDATE
END
```

Associated
LINE, LINETO, VLINE

HTAB

Purpose
Set/Read the current text cursor horizontal position.

Syntax
HTAB=*value*
value=HTAB

Description
Set/Read the current text cursor horizontal position.

Example
```
CLS
FOR xpos = 0 TO TWIDTH STEP 2 CYCLE
  HTAB=xpos
  PRINT HTAB
REPEAT
END
```

Associated
HVTAB, VTAB, PRINTAT

HVTAB

Purpose
Move the current text cursor to the specified position.

Syntax
HVTAB(*xpos*,*ypos*)

Description
The cursor is moved to the supplied column *xpos* and line *ypos*. Note that (0,0) is top-left of the text screen.

Example
```
CLS
HVTAB(TWIDTH/2-3,THEIGHT/2)
PRINT "CENTRE"
HVTAB(0,0)
END
```

Associated
HTAB, VTAB, PRINTAT

IF THEN

Purpose
Execute a statement conditionally.

Syntax
IF *condition* THEN {*statement*}

Description
The statement is executed when the condition evaluates to TRUE (not 0). Unlike some implementations of BASIC the THEN is required.

Example
```
PRINT "Press Space Bar to Continue"
CYCLE
   IF INKEY = 32 THEN BREAK
REPEAT
PRINT "Space Bar Pressed"
END
```

Associated
ELSE, ENDIF, SWITCH

INK

Purpose
Set/Read the current text foreground colour.

Syntax
foregroundcolour=INK
INK=foregroundcolour

Description
Set/Read the current text foreground (ink) colour.

Example
```
PAPER=black
CLS
string$="This is multicoloured text"
FOR I=1 TO LEN(string$) CYCLE
  INK=I MOD 15 + 1
  PRINT MID$(string$,I-1,1);
REPEAT
INK=white
PRINT
END
```

Associated
PAPER

INKEY

Purpose
Get a single character code from the keyboard without pausing.

Syntax
asciicode=INKEY

Description
This is similar to GET except that program execution is not paused; If no key is pressed, then -1 is returned. The following constants are predefined to test for special keys: KeyUp, KeyDown, KeyLeft, KeyRight, KeyIns, KeyHome, KeyDel, KeyEnd, KeyPgUp, KeyPgDn, KeyF1, KeyF2, KeyF3, KeyF4, KeyF5, KeyF6, KeyF7, KeyF8, KeyF9, KeyF10, KeyF11, KeyF12.

Example
```
REM Show the ASCII code for the last key
pressed
LastKey = -1
REM Press Esc to Quit
CYCLE
  Key=INKEY
  IF Key<>-1 AND Key<>LastKey THEN
    PRINT "Key Pressed: "; Key
    LastKey = Key
  ENDIF
REPEAT
```

Associated
GET, INPUT, SCANKEYBOARD

INPUT

Purpose
Read data from the keyboard into a variable.

Syntax
```
INPUT [prompt$,] variable
```

Description
When FUZE BASIC encounters the INPUT statement, program execution stops, a question mark(?) is printed and it waits for you to type something. It then assigns what you typed to the variable. If you typed in a string when it was expecting a number, then it will assign zero to the number. To stop it printing the question mark, you can optionally give it a string to print.

Example
```
INPUT "What is your Name? ", Name$
PRINT "Hello " + Name$
END
```

Associated
INKEY

INPUT#

Purpose
Read data from a file.

Syntax
```
INPUT# handle,variable
```

Description
This works similarly to the regular INPUT instruction, but reads from the file identified by the supplied *handle* rather than from the keyboard. Note that unlike the regular keyboard INPUT instruction, INPUT# can only read one variable at a time.

Example
```
handle=OPEN("testfile.txt")
PRINT# handle,"Hello World"
REWIND(handle)
INPUT# handle,record$
PRINT record$
CLOSE (handle)
END
```

Associated
CLOSE, EOF, FFWD, OPEN, PRINT#, REWIND, SEEK

INT

Purpose
Return the integer part of a number.

Syntax
integerpart=INT(*number*)

Description
Returns the integer part of the specified *number*.

Example
```
PRINT "The integer part of PI is ";
PRINT INT(PI)
END
```


LEFT

Purpose
Turns the turtle to the left (counter clockwise) by the given angle.

Syntax
LEFT(*angle*)

Description
Turns the virtual graphics turtle to the left (counter clockwise) by the given *angle* in the current angle units.

Example
```
REM Draw a box using turtle graphics
CLS
COLOUR=RED
MOVE(50)
DEG
LEFT(90)
MOVE(50)
PENDOWN
FOR I = 1 TO 4 CYCLE
  LEFT(90)
  MOVE(100)
REPEAT
UPDATE
END
```

Associated
MOVE, MOVETO, PENDOWN, PENUP, RIGHT, TANGLE

LEFT$

Purpose
Return the specified leftmost number of a characters from a string.

Syntax
*substring$*=LEFT$(*string$*,*number*)

Description
Returns a substring of *string$* with *number* characters from the left (start) of the string. If number is greater than or equal to the length of *string$* then the whole string is returned.

Example
```
string$="The quick brown fox"
FOR I=1 TO 20 CYCLE
  PRINT LEFT$(string$, I)
REPEAT
END
```

Associated
MID$, RIGHT$

LEN

Purpose
Return the length of the specified character string.

Syntax
length=LEN(*string$*)

Description
Returns the number of characters in the specified *string$*.

Example
```
String$="The Quick Brown Fox"
Chars=LEN(String$)
PRINT "String Length is: ";Chars
FOR I=0 TO Chars - 1 CYCLE
  Char$=MID$(String$,I,1)
  PRINT "Character No. ";I;" is "+Char$
REPEAT
END
```

LINE

Purpose
Draw a line between two points

Syntax
LINE(*xpos1*,*ypos1*,*xpos2*,*ypos2*)

Description
Draw a line between point (*xpos1*,*ypos1*) and point (*xpos2*,*ypos2*) in the current COLOUR.

Example
```
CLS
COLOUR = lime
GH=GHEIGHT
GW=GWIDTH
LINE(10,10,10,GH-10)
LINE(10,GH-10,GW-10,GH-10)
LINE(GW-10,GH-10,GW-10,10)
LINE(GW-10,10,10,10)
UPDATE
END
```

Associated
HLINE, LINETO, VLINE

LINETO

Purpose
Draw a line from the last point plotted.

Syntax
LINETO(*xpos1*,*ypos1*)

Description
Draws a line from the last point plotted (by the PLOT or LINE procedures) to point (*xpos1*,*ypos1*).

Example
```
CLS
COLOUR = yellow
ORIGIN(10,10)
GH=GHEIGHT
GW=GWIDTH
LINETO(0,GH-20)
LINETO(GW-20,GH-20)
LINETO(GW-20, 0)
LINETO(0,0)
UPDATE
END
```

Associated
HLINE, LINE, VLINE

LOADIMAGE

Purpose
Load an image file into memory.

Syntax
handle=LOADIMAGE(*filename*)

Description
Load an image from a file with the specified *filename*. The returned *handle* can then be used to plot it on the screen with PLOTIMAGE.

Example
```
COLOUR=RED
RECT(0,0,50,50,TRUE)
COLOUR=WHITE
LINE(0,50,50,50)
LINE(0,25,50,25)
LINE(0,25,0,50)
LINE(50,25,50,50)
LINE(25,0,25,25)
LINE(0,0,50,0)
SAVEREGION("bricks.bmp",0,0,50,50)
handle=LOADIMAGE("bricks.bmp")
FOR X=0 TO GWIDTH STEP 50 CYCLE
  FOR Y=0 TO GHEIGHT STEP 50 CYCLE
    PLOTIMAGE(handle,X,Y)
  REPEAT
REPEAT
UPDATE
END
```

Associated
GETIMAGEH, GETIMAGEW, PLOTIMAGE, FREEIMAGE

LOADMUSIC

Purpose
Load a music file into memory ready to be played.

Syntax
handle=LOADMUSIC(*filename*)

Description
Loads an uncompressed music file in wav format (file extension .wav) from the file *filename* into memory and returns a *handle* which can then be used to play the music using the PLAYMUSIC function.

Example
```
handle=LOADMUSIC("takeoff.wav")
SETMUSICVOL(70)
PLAYMUSIC(handle,1)
END
```

Associated
PAUSEMUSIC, RESUMEMUSIC, SETMUSICVOL, STOPMUSIC

LOADSAMPLE

Purpose
Load a sound sample into memory ready to be played.

Syntax
handle=LOADSAMPLE(*filename*)

Description
The LOADSAMPLE function loads a sound sample from the uncompressed WAV format file called *filename* and returns a *handle* to it so that it can be played using the PLAYSAMPLE function. You can load up to 32 sound samples into memory at the same time.

Example
```
channel=0
volume=70
SETCHANVOL(channel,volume)
intro=LOADSAMPLE("pacman_intro.wav")
PLAYSAMPLE(intro,channel,0)
WAIT(4.5)
END
```

Associated
PAUSECHAN, PLAYSAMPLE, RESUMECHAN, SETCHANVOL, STOPCHAN

LOADSPRITE

Purpose
Load a sprite from a file into memory.

Syntax
LOADSPRITE(*filename$,index,subindex*)

Description
This loads a sprite from the supplied *filename$* into memory and associates it with the given sprite *index* and *subindex*. The *index* is the handle returned by a call to NewSprite and the *subindex* is the version of the sprite to allow for animation. The first *subindex* is 0.

Example
```
CLS
REM Create a new sprite with 1 version
SpriteIndex=NEWSPRITE(1)
REM Load a sprite from a file
fuzelogo$="/usr/share/fuze/logo.bmp"
LOADSPRITE(fuzelogo$,SpriteIndex,0)
REM Draw the sprite on the screen
PLOTSPRITE(SpriteIndex,200,200,0)
UPDATE
END
```

Associated
GETSPRITEH, GETSPRITEW, HIDESPRITE, NEWSPRITE, PLOTSPRITE, SETSPRITETRANS, SPRITECOLLIDE, SPRITECOLLIDEPP

LOCAL

Purpose
Define variables to be local to a user defined procedure or function.

Syntax
LOCAL *variable*

Description
Allows a variable name to be reused in a procedure or function without affecting its value in the calling program.

Example
```
X=10
PRINT "Global X="+STR$(X)
Proc Test()
PRINT "Global X="+STR$(X)
END
DEF PROC Test()
  PRINT "Global X="+STR$(X)
  LOCAL X
  X=5
  PRINT "Local X="+STR$(X)
ENDPROC
```

Associated
DEF FN, DEF PROC

LOG

Purpose
Return the natural logarithm of the specified number.

Syntax
naturallogarithm = LOG(*number*)

Description
Returns the natural logarithm of the specified *number*. This is the opposite of the EXP function i.e. LOG(EXP(X)) = X. Logarithms are used in science to solve exponential radioactive decay problems and in finance to solve problems involving compound interest.

Example
```
X = EXP(10)
PRINT X // Will print 22026.46579
PRINT LOG(X) // Will print 10
END
```

Associated
EXP

MAX

Purpose
Returns the larger of two numbers.

Syntax
maxvalue=MAX(*number1*,*number2*)

Description
Returns the larger (highest value) of *number1* or *number2*.

Example
```
REM Prints the value of number2
number1=12.26
number2=27.45
PRINT MAX(number1, number2)
END
```

Associated
MIN

MID$

Purpose
Return characters from the middle of a string.

Syntax
MID$(*string$*, start, length)

Description
Returns the middle *length* characters of *string$* starting from position *start*. The first character of the string is position number 0.

Example
```
REM Prints Quick
string$="The Quick Brown Fox"
PRINT MID$(string$,4,5)
END
```

Associated
LEFT$, RIGHT$

MIN

Purpose
Returns the smaller of two numbers.

Syntax
minvalue=MIN(*number1*,*number2*)

Description
Returns the smaller (lowest value) of *number1* or *number2*.

Example
```
REM Prints the value of number1
number1=12.26
number2=27.45
PRINT MIN(number1, number2)
END
```

Associated
MAX

MOUSEOFF

Purpose
Make the mouse cursor invisible.

Syntax
MOUSEOFF

Description
Make the mouse cursor invisible within the FUZE BASIC window. This is the default value.

Example
```
CLS
MOUSEON
CYCLE
  GETMOUSE(x,y,b)
  LINETO(x,y)
  UPDATE
  REM Cycle colour if left button pressed
  IF b & 1 THEN
    COLOUR = COLOUR MOD 16 + 1
  ENDIF
  REM Exit if right button pressed
REPEAT UNTIL b & 4
MOUSEOFF
END
```

Associated
GETMOUSE, MOUSEON, MOUSEX, MOUSEY, SETMOUSE

MOUSEON

Purpose
Make the mouse cursor visible.

Syntax
```
MOUSEON
```

Description
Make the mouse cursor visible within the FUZE BASIC window. It is invisible by default.

Example
```
CLS
MOUSEON
CYCLE
  GETMOUSE(x,y,b)
  LINETO(x,y)
  UPDATE
  REM Cycle colour if left button pressed
  IF b & 1 THEN
    COLOUR = COLOUR MOD 16 + 1
  ENDIF
  REM Exit if right button pressed
REPEAT UNTIL b & 4
MOUSEOFF
END
```

Associated
GETMOUSE, MOUSEOFF, MOUSEX, MOUSEY, SETMOUSE

MOUSEX

Purpose
To find the mouse X position

Syntax
```
value = MOUSEX
```

Description
Returns the X position of the current mouse location

Example
```
CYCLE
PRINTAT(0,0); "Mouse X Position="; MOUSEX
;"    "
REPEAT
```

Associated
GETMOUSE, MOUSEOFF, MOUSEON, MOUSEY, SETMOUSE

MOUSEY

Purpose
To find the mouse Y position

Syntax
value = MOUSEY

Description
Returns the Y position of the current mouse location

Example
```
CYCLE
PRINTAT(0,0); "Mouse Y Position="; MOUSEY
;"    "
REPEAT
```

Associated
GETMOUSE, MOUSEOFF, MOUSEON, MOUSEX, SETMOUSE

MOVE

Purpose
Move the graphics turtle forward

Syntax
MOVE(*distance*)

Description
This causes the virtual graphics turtle to move forwards *distance* in screen pixels. A line will be drawn if the pen is
down.

Example
```
CLS
COLOUR=RED
MOVE(50)
DEG
LEFT(90)
MOVE(50)
PENDOWN
FOR I = 1 TO 4 CYCLE
  LEFT(90)
  MOVE(100)
REPEAT
UPDATE
END
```

Associated
LEFT, MOVETO, PENDOWN, PENUP, RIGHT, TANGLE

MOVETO

Purpose
Move the graphics turtle to a point on the screen.

Syntax
MOVETO(*xpos*,*ypos*)

Description
This moves the virtual graphics turtle to the absolute location (*xpos*,*ypos*). A line will be drawn if the pen is down.

Example
```
REM Draw a spiral in the centre of the
screen
CLS
COLOUR=RED
PENUP
MOVETO(GWIDTH/2,GHEIGHT/2)
PENDOWN
FOR I=2 TO GWIDTH CYCLE
  MOVE(I)
  RIGHT(30)
REPEAT
UPDATE
END
```

Associated
LEFT, MOVE, PENDOWN, PENUP, RIGHT, TANGLE

NEWSPRITE

Purpose
Create a new sprite.

Syntax
index=NEWSPRITE(*count*)

Description
This returns an index (or handle) to the internal sprite data. You need to use the index returned in all future sprite handling functions/procedures. The *count* argument specifies the number of different versions of the sprite.

Example overleaf...

Example

```
CLS
COLOUR=YELLOW
CIRCLE(100,100,50,TRUE)
SAVEREGION("pac1.bmp",50,50,101,101)
COLOUR=BLACK
TRIANGLE(100,100,150,125,150,75,TRUE)
SAVEREGION("pac2.bmp",50,50,101,101)
TRIANGLE(100,100,150,150,150,50,TRUE)
SAVEREGION("pac3.bmp",50,50,101,101)
CLS
pacman=NEWSPRITE(3)
LOADSPRITE("pac1.bmp",pacman,0)
LOADSPRITE("pac2.bmp",pacman,1)
LOADSPRITE("pac3.bmp",pacman,2)
FOR X=1 TO GWIDTH STEP 25 CYCLE
  FOR S=0 TO 2 CYCLE
    PLOTSPRITE(pacman,X,GHEIGHT/2,S)
    UPDATE
    WAIT(.1)
  REPEAT
REPEAT
HIDESPRITE(pacman)
END
```

Associated

GETSPRITEH, GETSPRITEW, HIDESPRITE, LOADSPRITE, PLOTSPRITE, SETSPRITETRANS, SPRITECOLLIDE, SPRITECOLLIDEPP

NUMFORMAT

Purpose

Control how numbers are formatted.

Syntax

NUMFORMAT(*width*,*decimals*)

Description

You can affect the way numbers are printed using the NUMFORMAT procedure. This takes 2 arguments, the *width* specifying the total number of characters to print and the *decimals* the number of characters after the decimal point. Numbers printed this way are right-justified with leading spaces inserted if required. NUMFORMAT (0,0) restores the output to the general purpose format used by default.

Example

```
NUMFORMAT(6,4)
REM Prints 3.1416
PRINT PI
NUMFORMAT(0,0)
REM Prints 3.141592654
PRINT PI
END
```

Associated

PRINT, PRINTAT

OPEN

Purpose
Open a file for read or write.

Syntax
handle=OPEN(*filename$*)

Description
The OPEN function opens a file and makes it available for reading or writing and returns the numeric *handle* associated with the file. The file is created if it doesn't exist, or if it does exist the file pointer is positioned at the start of the file.

Example
```
handle = OPEN("testfile.txt")
PRINT# handle, "Colin"
PRINT# handle, 47
CLOSE(handle)
handle = OPEN("testfile.txt")
INPUT# handle, Name$
INPUT# handle, Age
CLOSE(handle)
PRINT "Name: " + Name$
PRINT "Age: "; Age
END
```

Associated
CLOSE, EOF, FFWD, INPUT#, PRINT#, REWIND, SEEK

ORIGIN

Purpose
Move the graphics origin.

Syntax
ORIGIN(*xpos*,*ypos*)

Description
This changes the graphics origin for the Cartesian plotting procedures. The *xpos*, *ypos* coordinates are always absolute coordinates with (0,0) being bottom left (the default).

Example
```
CLS
COLOUR=yellow
REM Move the origin to the screen centre
ORIGIN(GWIDTH/2, GHEIGHT/2)
PLOT(-100,-100)
LINETO(-100,100)
LINETO(100,100)
LINETO(100,-100)
LINETO(-100,-100)
UPDATE
ORIGIN(0,0)
END
```

Associated
GHEIGHT, GWIDTH

PAPER

Purpose
Set/Read the current text background colour.

Syntax
backgroundcolour=PAPER
PAPER=backgroundcolour

Description
Set/Read the current text background (paper) colour. Clear screen (CLS) will set the entire background to this colour.

Example
```
PRINT "Text background colour "; PAPER
PAPER=RED
PRINT "Text background colour "; PAPER
PAPER=7
PRINT "Text background colour "; PAPER
END
```

Associated
INK, PAPERON, PAPEROFF

PAPEROFF

Purpose
Switches the text background colour off.

Syntax
PAPEROFF

Description
This function switches the background text colour off. It can be turned on or off so that the text background does not obscure whatever is behind it.

Example
```
CYCLE
  INK = Orange
  PaperOn
  fontSize (RND (10) + 1)
  printAt (RND(tWidth), RND(tHeight - 1)); "ON";
  WAIT(.3)
  INK = Yellow
  PaperOff
  fontSize (RND (10) + 1)
  printAt (RND(tWidth), RND(tHeight - 1)); "OFF";
  WAIT(.3)
REPEAT
```

Associated
INK, PAPER, PAPERON

PAPERON

Purpose
Sets text background colour to display

Syntax
PAPERON

Description
This function switches the background text colour on. It can be turned on or off so that the text background does not obscure whatever is behind it.

Example
```
CYCLE
  INK = Orange
  PaperOn
  fontSize (RND (10) + 1)
  printAt (RND(tWidth), RND(tHeight - 1)); "ON";
  WAIT(.3)
  INK = Yellow
  PaperOff
  fontSize (RND (10) + 1)
  printAt (RND(tWidth), RND(tHeight - 1)); "OFF";
  WAIT(.3)
REPEAT
```

Associated
INK, PAPER, PAPEROFF

PAUSECHAN

Purpose
Pause the playing of a sound sample.

Syntax
PAUSECHAN(*handle*)

Description
This function pauses the playing of the sound sample associated with the *handle* returned by *LOADSAMPLE* that has been started using *PLAYSAMPLE*. The sample can be resumed where it left off using *RESUMECHAN*

Example
```
channel=0
volume=70
SETCHANVOL(channel,volume)
intro=LOADSAMPLE("pacman_intro.wav")
PLAYSAMPLE(intro,channel,0)
WAIT(3)
PAUSECHAN(intro)
Wait(2)
RESUMECHAN(intro)
END
```

Associated
LOADSAMPLE, PLAYSAMPLE, RESUMECHAN, SETCHANVOL, STOPCHAN

PAUSEMUSIC

Purpose
Pause a playing music file

Syntax
PAUSEMUSIC

Description
Pauses a playing music track which can them be restarted using RESUMEMUSIC.

Example
```
handle=LOADMUSIC("takeoff.wav")
SETMUSICVOL(70)
PLAYMUSIC(handle,1)
PAUSEMUSIC
```

Associated
LOADMUSIC, RESUMEMUSIC, SETMUSICVOL, STOPMUSIC

PENDOWN

Purpose
Start drawing using the graphics turtle.

Syntax
PENDOWN

Description
This lowers the “pen” that the virtual graphics turtle is using to draw with. Nothing will be drawn until you execute this procedure.

Example
```
REM Draw a spiral in the centre of the
screen
CLS
COLOUR=RED
PENUP
MOVETO(GWIDTH/2,GHEIGHT/2)
PENDOWN
FOR I=2 TO GWIDTH CYCLE
  MOVE(I)
  RIGHT(30)
REPEAT
UPDATE
END
```

Associated
LEFT, MOVE, MOVETO, PENUP, RIGHT, TANGLE

PENUP

Purpose
Stop drawing using the graphics turtle.

Syntax
PENUP

Description
This lifts the “pen” that the virtual graphics turtle uses to draw. You can move the turtle without drawing while the pen is up.

Example
```
REM Draw a spiral in the centre of the
screen
CLS
COLOUR=RED
PENUP
MOVETO(GWIDTH/2,GHEIGHT/2)
PENDOWN
FOR I=2 TO GWIDTH CYCLE
  MOVE(I)
  RIGHT(30)
REPEAT
UPDATE
END
```

Associated
LEFT, MOVE, MOVETO, PENDOWN, RIGHT, TANGLE

PI

Purpose
Returns the value of the constant pi.

Syntax
valueofpi=PI

Description
Returns an approximation of the value of the constant pi which is the ratio of a circle’s circumference to its diameter (approximately 3.141592654) which is widely used in mathematics, specifically trigonometry and geometry.

Example
```
PRINT FN AreaOfCircle(12)
END
DEF FN AreaOfCircle(withRadius)
  LOCAL result
  result=PI*withRadius*withRadius
= result
```

PINMODE

Purpose
Configure the mode of a pin on the Pi's GPIO.

Syntax
PINMODE(*pinno*,*pinmode*)

Description
Configures the mode of a pin on the Pi's GPIO. It takes an argument which specifies the mode of the pin - input, output or PWM output. The modes are:
0 pinINPUT
1 pinOUTPUT
2 pinPWM

Example
```
REM Set pin 12 to input
PINMODE(12,1)
REM Wait for button to be pushed
UNTIL DIGITALREAD(12) CYCLE
REPEAT
PRINT "Button Pushed"
END
```

Associated
DIGITALREAD, DIGITALWRITE, PWMWRITE, SOFTPWMWRITE

PLAYMUSIC

Purpose
Start playing a music track.

Syntax
PLAYMUSIC(*handle*,*repeats*)

Description
This function plays a music track previously loaded using the *LOADMUSIC* function which returns the *handle*. The *repeats* are the number of times to play the track.

Example
```
handle=LOADMUSIC("takeoff.wav")
SETMUSICVOL(70)
PLAYMUSIC(handle,1)
END
```

Associated
LOADMUSIC, PAUSEMUSIC, RESUMEMUSIC, SETMUSICVOL, STOPMUSIC

PLAYSAMPLE

Purpose
Start playing a sound sample.

Syntax
PLAYSAMPLE(*handle*,*channel*,*loops*)

Description
This function plays a sound previously loaded using the *LOADSAMPLE* function which returns the *handle*. The *channel* is 0, 1, 2 or 3 which lets you play up to 4 concurrent samples. The *loops* parameter is different to the *repeats* one in the *PLAYMUSIC* function. Here it means the number of times to loop the sample – zero means no loops which means play it ONCE.

Example
```
channel=0
volume=70
SETCHANVOL(channel,volume)
intro=LOADSAMPLE("pacman_intro.wav")
PLAYSAMPLE(intro,channel,0)
WAIT(4.5)
END
```

Associated
LOADSAMPLE, PAUSECHAN, RESUMECHAN, SETCHANVOL, STOPCHAN

PLOT

Purpose
Draw a single point on the screen.

Syntax
PLOT(*xpos*,*ypos*)

Description
This plots a single pixel at screen location (*xpos*,*ypos*) in the selected graphics mode in the selected colour. Note that (0,0) is bottom left by default.

Example
```
CLS
CYCLE
  IF INKEY<>-1 THEN BREAK
  xpos=RND(GWIDTH)
  ypos=RND(GHEIGHT)
  COLOUR=RND(16)
  PLOT(xpos, ypos)
  UPDATE
REPEAT
END
```

PLOTIMAGE

Purpose
Display a loaded image on the screen.

Syntax
PLOTIMAGE(*handle*,*xpos*,*ypos*)

Description
Plot an image previously loaded using LOADIMAGE (using the *handle* returned by LOADIMAGE) on the screen at coordinates (*xpos*,*ypos*).

Example
```
COLOUR=RED
RECT(0,0,50,50,TRUE)
COLOUR=WHITE
LINE(0,50,50,50)
LINE(0,25,50,25)
LINE(0,25,0,50)
LINE(50,25,50,50)
LINE(25,0,25,25)
LINE(0,0,50,0)
SAVEREGION("bricks.bmp",0,0,50,50)
handle=LOADIMAGE("bricks.bmp")
FOR X=0 TO GWIDTH STEP 50 CYCLE
  FOR Y=0 TO GHEIGHT STEP 50 CYCLE
    PLOTIMAGE(handle,X,Y)
  REPEAT
REPEAT
UPDATE
END
```

Associated
GETIMAGEH, GETIMAGEW, LOADIMAGE, FREEIMAGE

PLOTSPRITE

Purpose
Draw a sprite on the screen.

Syntax
PLOTSPRITE(*index*,*xpos*,*ypos*,*subindex*)

Description
This plots the sprite *index* and version *subindex* at the coordinates (*xpos*, *ypos*). The coordinates specify the bottom-left corner of the bounding rectangle of the sprite.

Example
```
CLS
REM Create a new sprite with 1 version
SpriteIndex=NEWSPRITE(1)
REM Load a sprite from a file
fuzelogo$="/usr/share/fuze/logo.bmp"
LOADSPRITE(fuzelogo$,SpriteIndex,0)
REM Draw the sprite on the screen
PLOTSPRITE(SpriteIndex,200,200,0)
UPDATE
END
```

Associated
GETSPRITEH, GETSPRITEW, HIDESPRITE, LOADSPRITE, NEWSPRITE, SETSPRITEALPHA, SETSPRITEANGLE, SETSPRITEFLIP, SETSPRITESIZE, SETSPRITETRANS, SPRITECOLLIDE, SPRITECOLLIDEPP, SPRITEOUT

POLYEND

Purpose
Draw the filled polygon started by PolyStart.

Syntax
POLYEND

Description
This marks the end of drawing a polygon. When this is called, the stored points will be plotted on the screen and the polygon will be filled.

Example
```
CLS
PROC Hexagon(200,200,50,Red)
UPDATE
END
DEF Proc Hexagon(x,y,l,c)
  COLOUR=c
  POLYSTART
  POLYPLOT(x+l,y)
  POLYPLOT(x+l/2,y-l*SQRT(3/2))
  POLYPLOT(x-l/2,y-l*SQRT(3/2))
  POLYPLOT(x-l,y)
  POLYPLOT(x-l/2,y+l*SQRT(3/2))
  POLYPLOT(x+l/2,y+l*SQRT(3/2))
  POLYEND
ENDPROC
```

Associated
POLYPLOT, POLYSTART

POLYPLOT

Purpose
Add a point to a filled polygon.

Syntax
POLYPLOT(*xpos*,*ypos*)

Description
This remembers the given *xpos*,*ypos* coordinates as part of a filled polygon. Nothing is actually drawn on the screen until the PolyEnd instruction is executed. Polygons can have a maximum of 64 points.

Example
```
CLS
PROC Hexagon(200,200,50,Red)
UPDATE
END
DEF Proc Hexagon(x,y,l,c)
  COLOUR=c
  POLYSTART
  POLYPLOT(x+l,y)
  POLYPLOT(x+l/2,y-l*SQRT(3/2))
  POLYPLOT(x-l/2,y-l*SQRT(3/2))
  POLYPLOT(x-l,y)
  POLYPLOT(x-l/2,y+l*SQRT(3/2))
  POLYPLOT(x+l/2,y+l*SQRT(3/2))
  POLYEND
ENDPROC
```

Associated
POLYEND, POLYSTART

POLYSTART

Purpose
Start drawing a filled polygon.

Syntax
```
POLYSTART
```

Description
This marks the start of drawing a filled polygon.

Example
```
CLS
PROC Hexagon(200,200,50,Red)
UPDATE
END
DEF Proc Hexagon(x,y,l,c)
  COLOUR=c
  POLYSTART
  POLYPLOT(x+l,y)
  POLYPLOT(x+l/2,y-l*SQRT(3/2))
  POLYPLOT(x-l/2,y-l*SQRT(3/2))
  POLYPLOT(x-l,y)
  POLYPLOT(x-l/2,y+l*SQRT(3/2))
  POLYPLOT(x+l/2,y+l*SQRT(3/2))
  POLYEND
ENDPROC
```

Associated
POLYEND, POLYPLOT

PLOTTEXT

Purpose
Display text using graphic coordinates.

Syntax
```
PLOTTEXT(“text”, xpos, ypos)
```

Description
The PRINT command uses the cursor coordinates to display text whereas PLOTTEXT can position text at a specified pixel location

Example
```
CYCLE
  INK = RND (30)
  fontSize (RND (10) + 1)
  plotText("HELLO", RND(gWidth), RND(gHeight-1))
  UPDATE
REPEAT
```

Associated
PRINT, PRINTAT

PRINT

Purpose
Output text to the screen

Syntax
```
PRINT {text}{;text}
```

Description
Outputting text to the screen is done via the PRINT command. The PRINT command is quite versatile and will print any combination of numbers and strings separated by the semi-colon (;). A trailing semi-colon will suppress the printing of a new line.

Example
```
PRINT "Hello Colin"
PRINT "Hello ";
name$="Colin"
PRINT name$
PRINT "Hello "; name$
END
```

Associated
NUMFORMAT, PRINTAT

PRINT#

Purpose
Print data to a file.

Syntax
```
PRINT# handle,data
```

Description
The PRINT# instruction acts just like the regular PRINT instruction except that it sends data to the file identified by the supplied file-handle rather than to the screen. Numbers are formatted according to the settings of NUMFORMAT. It is strongly recommended to only print one item per line if you are going to read those items back into a FUZE BASIC program again.

Example
```
handle = OPEN("testfile.txt")
PRINT# handle, "Hello World"
CLOSE (handle)
END
```

Associated
CLOSE, EOF, FFWD, INPUT#, OPEN, REWIND, SEEK

PRINTAT

Purpose
Set the text cursor position and print

Syntax
PRINTAT(*x*, *y*); "*text*"

Description
Use to position the text cursor at the specified location and print. Useful for laying out text and or variables at any preferred screen location.

Example
```
name$ = "Sam"
age = 10
PRINTAT( 0, 5 ); "My name is "; name$
PRINTAT( 5, 10 ); "I am "; age ; " years
old"
END
```

Associated
CHR$, HTAB, HVTAB, INK,
PAPER, PRINT, PRINT, THEIGHT, TWIDTH, VTAB

PROC

Purpose
Call a user defined procedure.

Syntax
PROC *name*({*argument*}{,*argument*})

Description
Calls the specified user defined procedure called *name* with the specified *arguments*. Once the procedure has been executed control returns to the command following.

Example
```
CLS
CYCLE
  x=RND(TWIDTH)
  y=RND(THEIGHT)
  c=RND(15)
  Text$=”Blossom”
  PROC text(text$, x, y, c)
REPEAT
END
DEF PROC text(text$, x, y, c)
  INK=c
  PRINTAT(x,y); text$
ENDPROC
```

Associated
DEF PROC, ENDPROC

PWMWRITE

Purpose
Output a PWM waveform on the selected pin.

Syntax
PWMWRITE(*pinno*,*pinvalue*)

Description
This procedure outputs a PWM waveform on the selected pin. The pin must be configured for PWM mode beforehand. The value set should be between 0 and 100.

Example
```
REM Set pin 1 to PWM output mode
PINMODE(1,2)
PWMWRITE (1,50)
END
```

Associated
DIGITALREAD, DIGITALWRITE, PINMODE

RAD

Purpose
Set angle units to radians.

Syntax
RAD

Description
Switches the internal angle system to radians. There are 2 * PI radians in a full circle.

Example
```
REM Draw an ellipse in the screen centre
CLS
RAD
FOR Angle=0 TO 2 * PI STEP 0.01 CYCLE
  Xpos=100*COS(Angle)+GWIDTH / 2
  Ypos=50*SIN(Angle)+GHEIGHT / 2
  PLOT(Xpos,Ypos)
REPEAT
END
```

Associated
CLOCK, DEG

READ

Purpose
Read data into program variables.

Syntax
```
READ variable {,variable}
```

Description
To get data into your program variables, we use the READ instruction. We can read one, or many items of data at a time.

Example
```
REM Load the name of the days of the
REM week into a string array
DATA "Monday", "Tuesday", "Wednesday"
DATA "Thursday", "Friday", "Saturday"
DATA "Sunday"
DIM DaysOfWeek$(7)
FOR Day = 1 TO 7 CYCLE
  READ DaysOfWeek$(Day)
REPEAT
PRINT "The third day of the week is ";
PRINT DaysOfWeek$(3)
END
```

Associated
DATA, RESTORE

RECT

Purpose
Draw a rectangle on the screen.

Syntax
```
RECT(xpos,ypos,width,height,fill)
```

Description
Draws a rectangle at position (*xpos*,*ypos*) with *width* and *height*. The final parameter, *fill* is either TRUE or FALSE, and specifies filled (TRUE) or outline (FALSE).

Example
```
CLS
CYCLE
  COLOUR=RND(16)
  x=RND(GWIDTH)
  y=RND(GHEIGHT)
  w=RND(GWIDTH / 4)
  h=RND(GHEIGHT / 4)
  f=RND(2)
  RECT(x,y,w,h,f)
  UPDATE
  IF INKEY<>-1 THEN BREAK
REPEAT
END
```

Associated
CIRCLE, ELLIPSE, TRIANGLE

REPEAT UNTIL

Purpose
Loop until the specified condition is met.

Syntax
```
CYCLE
  {statements}
REPEAT UNTIL condition
```

Description
Execute the *statements* one or more times until the *condition* is TRUE (not 0).

Example
```
Number=INT(RND(10)) + 1
Guess=0
REM Guessing Game
PRINT "Guess a Number Between 1 and 10"
CYCLE
  INPUT "Enter Your Guess: ", Guess
  IF (Number<>Guess) THEN
    PRINT "Incorrect Guess Again"
  ENDIF
REPEAT UNTIL Number = Guess
PRINT "You are Correct!"
END
```

Associated
BREAK, CONTINUE, CYCLE, CYCLE REPEAT, FOR REPEAT, UNTIL REPEAT, WHILE REPEAT

RESTORE

Purpose
Reset the DATA pointer.

Syntax
```
RESTORE
```

Description
With no *lineno* specified resets the READ command to the very first DATA statement in the program.

Example
```
DATA "Monday", "Tuesday", "Wednesday"
DATA "Thursday", "Friday", "Saturday",
"Sunday"
FOR Day = 1 TO 3 CYCLE
  READ DayOfWeek$
REPEAT
PRINT DayOfWeek$
RESTORE
FOR Day = 1 TO 4 CYCLE
  READ DayOfWeek$
REPEAT
PRINT DayOfWeek$
END
```

Associated
DATA, READ

RESUMECHAN

Purpose
Resume the playing of a sound sample.

Syntax
RESUMECHAN(*handle*)

Description
This function resumes the playing of the sound sample associated with the *handle* returned by *LOADSAMPLE* that has been started using *PLAYSAMPLE* and paused using *PAUSECHAN*.

Example
```
channel=0
volume=70
SETCHANVOL(channel,volume)
intro=LOADSAMPLE("pacman_intro.wav")
PLAYSAMPLE(intro,channel,0)
WAIT(3)
PAUSECHAN(intro)
WAIT(2)
RESUMECHAN(intro)
END
```

Associated
LOADSAMPLE, PAUSECHAN, PLAYSAMPLE, SETCHANVOL, STOPCHAN

RESUMEMUSIC

Purpose
Resumes music playing after it has been paused.

Syntax
RESUMEMUSIC

Description
Resumes the playing of a music track previously paused using PAUSEMUSIC.

Example
```
handle=LOADMUSIC("takeoff.wav")
SETMUSICVOL(70)
PLAYMUSIC(handle,1)
PAUSEMUSIC
WAIT(1)
RESUMEMUSIC
```

Associated
LOADMUSIC, PAUSEMUSIC, SETMUSICVOL, STOPMUSIC

REWIND

Purpose
Move the file pointer to the start of a file.

Syntax
REWIND(*handle*)

Description
Move the file pointer to the start of the file specified by *handle*.

Example
```
handle=OPEN ("rewindtest.txt")
PRINT# handle,"First Record"
PRINT# handle,"Second Record"
CLOSE(handle)
handle=OPEN("rewindtest.txt")
INPUT# handle, record$
PRINT record$
REWIND(handle)
REM reads the first record again
INPUT# handle, record$
PRINT record$
CLOSE (handle)
END
```

Associated
CLOSE, EOF, FFWD, INPUT#, OPEN, PRINT#, SEEK

RGBCOLOUR

Purpose
Set the current graphical plot colour to an RGB (Red,Green, Blue) value.

Syntax
RGBCOLOUR(*red,green,blue*)

Description
This sets the current graphical plot colour to an RGB (Red,Green, Blue) value. The values should be from 0 to 255.

Example
```
CLS
PRINT "Draw Spectrum"
FOR v = 0 TO 255 CYCLE
  RGBCOLOUR(255,v,0)
  LINE(v,300,v,400)
  RGBCOLOUR(v,255,0)
  LINE(511-v,300,511-v,400)
  RGBCOLOUR(0,255-v,v)
  LINE(512+v,300,512+v,400)
  RGBCOLOUR(0,v,255)
  LINE(768+v,300,768+v,400)
REPEAT
UPDATE
END
```

Associated
COLOUR, INK, PAPER

RIGHT

Purpose
Turns the turtle to the right (clockwise) by the given angle.

Syntax
RIGHT(*angle*)

Description
Turns the virtual graphics turtle to the right (clockwise) by the given *angle* in the current angle units.

Example
```
CLS
PRINT "Draw Pink Hexagon"
PENUP
COLOUR = PINK
PENDOWN
FOR I = 1 TO 6 CYCLE
  RIGHT(60)
  MOVE(100)
REPEAT
END
```

Associated
LEFT, MOVE, MOVETO, PENDOWN, PENUP, TANGLE

RIGHT$

Purpose
Return the specified rightmost number of a characters from a string.

Syntax
*substring$*=RIGHT$(*string$*,*number*)

Description
Returns a substring of *string$* with *number* characters from the right (end) of the string. If number is greater than or equal to the length of *string$* then the whole string is returned.

Example
```
String$="The quick brown fox"
FOR I=1 TO 20 CYCLE
  PRINT RIGHT$(String$, I)
REPEAT
END
```

Associated
LEFT$, MID$

ROTATEIMAGE

Purpose
Return the specified rightmost number of a characters from a string.

Syntax
*substring$*=RIGHT$(*string$*,*number*)

Description
Returns a substring of *string$* with *number* characters from the right (end) of the string. If number is greater than or equal to the length of *string$* then the whole string is returned.

Example
```
image = loadImage ("screen.png")
plotImage (image, 0, 0)
UPDATE
WAIT (2)
CLS
rotateImage (image, 90)
plotImage (image, 0, 0)
UPDATE
WAIT (2)
END
```

Associated
LOADIMAGE, SCALEIMAGE

RND

Purpose
Generate a random number in a given range.

Syntax
random=RND(*range*)

Description
This function returns a random number based on the value of *range*. If *range* is zero, then the last random number generated is returned, if *range* is 1, then a random number from 0 to 1 is returned, otherwise a random number from 0 up to, but not including *range* is returned.

Example
```
DiceRoll=RND(6)+1
PRINT "Dice Roll: "; DiceRoll
CoinToss=RND(2)
IF CoinToss=0 THEN
  PRINT "Heads"
ELSE
  PRINT "Tails"
ENDIF
END
```

Associated
SEED

SAVEREGION

Purpose
Save a snapshot of an area of the screen to an image file.

Syntax
SAVEREGION(*file$,xpos,ypos,width,height*)

Description
This takes a snapshot of an area of the current screen, specified by the rectangle with bottom left at coordinates (*xpos*,*ypos*) of specified *width* and *height*, and saves it to the file named *file$* in a bitmap (.bmp) format.

Example (overleaf)

SAVEREGION Example

```
CLS
COLOUR=YELLOW
CIRCLE(100,100,50,TRUE)
SAVEREGION("pac1.bmp",50,50,101,101)
COLOUR=BLACK
TRIANGLE(100,100,150,125,150,75,TRUE)
SAVEREGION("pac2.bmp",50,50,101,101)
TRIANGLE(100,100,150,150,150,50,TRUE)
SAVEREGION("pac3.bmp",50,50,101,101)
CLS
pacman=NEWSPRITE(3)
LOADSPRITE("pac1.bmp",pacman,0)
LOADSPRITE("pac2.bmp",pacman,1)
LOADSPRITE("pac3.bmp",pacman,2)
FOR X=1 TO GWIDTH STEP 25 CYCLE
  FOR S=0 TO 2 CYCLE
    PLOTSPRITE(pacman,X,GHEIGHT/2,S)
    UPDATE
    WAIT(.1)
  REPEAT
REPEAT
HideSprite(pacman)
END
```

Associated
SAVESCREEN

SAVESCREEN

Purpose
Save a snapshot of the screen to an image file.

Syntax
SAVESCREEN(*filename$*)

Description
This takes a snapshot of the current screen and saves it to the filename given in a bitmap (.bmp) format file.

Example
```
CLS
PRINT "Draw Spectrum"
FOR v = 0 TO 255 CYCLE
  RGBCOLOUR(255,v,0)
  LINE(v,300,v,400)
  RGBCOLOUR(v,255,0)
  LINE(511-v,300,511-v,400)
  RGBCOLOUR(0,255-v,v)
  LINE(512+v,300,512+v,400)
  RGBCOLOUR(0,v,255)
  LINE(768+v,300,768+v,400)
REPEAT
UPDATE
SAVESCREEN("screenshot.bmp")
END
```

Associated
SAVEREGION

SCALEIMAGE

Purpose
Resize a loaded image.

Syntax
SCALEIMAGE(*handle, percent*)

Description
Scales a preloaded image by a specified percentage.

Example
```
image = loadImage ("screen.png")
plotImage (image, 0, 0)
UPDATE
WAIT (2)
CLS
scaleImage (image, 50)
plotImage (image, 0, 0)
UPDATE
WAIT (2)
END
```

Associated
LOADIMAGE, ROTATEIMAGE

SCANKEYBOARD

Purpose
Scan for a key pressed down.

Syntax
SCANKEYBOARD(*keycode*)

Description
Allows you to detect that any of the keys have been pressed (including special keys) and also to detect mutiple keys pressed at the same time. The *keycode* parameter indicates the key press to be scanned for e.g scanSpace is the space bar. See the end of this guide for a full list of SCANKEYBOARD keycodes.

Example
```
PRINT "Press Ctrl-Alt-Delete"
CYCLE
  LCtrl = SCANKEYBOARD(scanLCtrl)
  LAlt  = SCANKEYBOARD(scanLAlt)
  Delete = SCANKEYBOARD(scanDelete)
  Reboot = LCtrl AND LAlt AND Delete
REPEAT UNTIL Reboot
PRINT "Rebooting…"
CLEARKEYBOARD
END
```

Associated
CLEARKEYBOARD, GET, INKEY, INPUT

SCLOSE

Purpose
Close an open serial port.

Syntax
SCLOSE(*handle*)

Description
This closes a serial port and frees up any resources used by it. It's not strictly necessary to do this when you end your program, but it is considered good practice.

Example
```
REM Read a character from a serial port
arduino=SOPEN("/dev/ttyUSB0", 115200)
char$=SGET$(arduino)
PRINT char$
SCLOSE(arduino)
END
```

Associated
SGET, SGET$, SOPEN, SPUT, SPUT$, SREADY

SCROLLDOWN
SCROLLLEFT
SCROLLRIGHT
SCROLLUP

Purpose
Scroll a region of the screen down.

Syntax
SCROLLDOWN(*xpos*,*ypos*,*width*,*height*,*pixels*)

Description
Scroll the region of the screen specified by the rectangle at position (*xpos*,*ypos*) with dimensions *width* X *height* down by the specified number of *pixels*.

Example overleaf

Example
```
CLS
W=100 // Width
S=2 // Step Size
X=(GWIDTH-W)/2
Y=(GHEIGHT-W)/2
COLOUR=WHITE
RECT(X,Y,W,W,TRUE)
RECT(X+W,Y+W,W,W,TRUE)
RECT(X-2,Y-2,W*2+4,W*2+4,FALSE)
UPDATE
COLOUR=BLACK
FOR I=1 TO W STEP S CYCLE
  SCROLLUP(X,Y,W,W*2,S)
  SCROLLDOWN(X+W,Y,W,W*2,S)
  UPDATE
  WAIT(0.01)
REPEAT
FOR I = 1 TO W STEP S CYCLE
  SCROLLRIGHT(X,Y+W,W*2,W,S)
  SCROLLLEFT(X,Y,W*2,W,S)
  WAIT(0.01)
UPDATE
REPEAT
END
```

Associated
SCROLLLEFT, SCROLLRIGHT, SCROLLUP

SEED

Purpose
Seed the random number generator.

Syntax
SEED=*value*

Description
This can be assigned to initialise the random number generator, or you can read it to find the current seed.

Example
```
SEED=10
PRINT RND(100)
SEED=10
REM Will print the same number
PRINT RND(100)
REM Will print a different number
PRINT RND(100)
END
```

Associated
RND

SEEK

Purpose
Move the file pointer to any place in the file.

Syntax
SEEK(*handle*,*offset*)

Description
The SEEK instruction moves the file pointer to any place in the file. It can even move the file pointer beyond the end of the file in which case the file is extended. The argument supplied to SEEK is an absolute number of bytes from the start of the file. If you are using random access files and want to access the 7th record in the file, then you need to multiply your record size by 7 to get the final location to seek to.

Example
```
handle=OPEN("TestFile.txt")
recSize = 20
FOR recNo=0 TO 10 CYCLE
  record$ = "Record " + STR$(recNo)
  pad = recSize - LEN(record$)
  PRINT# handle,record$;SPACE$(pad)
REPEAT
REM read the 7th record
SEEK (handle,(recSize+1)*7)
INPUT# handle,record$
PRINT record$
CLOSE(handle)
END
```

Associated
CLOSE, EOF, FFWD, INPUT#, OPEN, PRINT#, REWIND

SENSEACCELX / Y / Z

Purpose
Returns the value of the Raspberry Pi senseHAT accelerometer.

Syntax
```
value=SENSEACCELX
value=SENSEACCELY
value=SENSEACCELZ
```

Description

The Raspberry Pi senseHAT has a number of built in sensors. The accelerometer can be accessed with this function.

Example
```
CLS
LOOP
PRINT “Sense Accelerometer X=”; senseAccelX
PRINT “Sense Accelerometer Y=”; senseAccelY
PRINT “Sense Accelerometer Z=”; senseAccelZ
REPEAT
END
```

Associated
senseCompass, senseGyro

SENSECLS

Purpose
Sets all of the LEDs on the Raspberry Pi senseHAT to off.

Syntax
```
SENSECLS
```

Description

Sets an RGB value of 0, 0, 0 to all of the matrix LEDs thereby clearing the display.

Example
```
CLS
sensePlot(2,2)
WAIT (1)
senseCls
END
```

Associated
sensePlot, senseRect, senseScroll, senseHFlip, senseVflip, senseRGBcolour, senseGetRGB, senseLine

SENSECOMPASSX / Y / Z

Purpose
Returns the value of the Raspberry Pi senseHAT compass.

Syntax
```
value=SENSECOMPASSX
value=SENSECOMPASSY
value=SENSECOMPASSZ
```

Description

The Raspberry Pi senseHAT has a number of built in sensors. The compass can be accessed with this function.

Example
```
CLS
LOOP
PRINT “Sense Compass X=”; senseCompassX
PRINT “Sense Compass Y=”; senseCompassY
PRINT “Sense Compass Z=”; senseCompassZ
REPEAT
END
```

Associated
senseAccel, senseGyro,

SENSEGETRGB

Purpose
Return the values used by a Raspberry Pi senseHAT LED

Syntax
SENSEGETRGB(*xpos, ypos, red, green, blue*)

Description

Returns the Red, Green and Blue values from a given LED at the specified matrix coordinates. It is possible to use this for collision detection in games using the senseHAT.

Example
```
CLS
senseRGBcolour (0, 0, 255)
sensePlot (0, 0)
SenseGetRGB (0, 0, R, G, B)
PRINT “Red=”; R
PRINT “Green=”; G
PRINT “Blue=”; B
END
```

Associated
senseCls, sensePlot, senseRect, senseScroll, senseHFlip, senseVflip, senseRGBcolour, , senseLine

SENSEGYROX / Y / Z

Purpose
Returns the value of the Raspberry Pi senseHAT gyro.

Syntax
```
value=SENSEGYROX
value=SENSEGYROY
value=SENSEGYROZ
```

Description

The Raspberry Pi senseHAT has a number of built in sensors. The gyro can be accessed with this function.

Example
```
CLS
LOOP
PRINT “Sense Gyro X=”; senseGyroX
PRINT “Sense Gyro Y=”; senseGyroY
PRINT “Sense Gyro Z=”; senseGyroZ
REPEAT
END
```

Associated
senseAccel, senseCompass

SENSEHEIGHT

Purpose
Returns the value of the Raspberry Pi senseHAT height sensor.

Syntax
```
value=SENSEHEIGHT
```

Description

The Raspberry Pi senseHAT has a number of built in sensors. Height above sea level can be accessed with this function. Note, this does not work well indoors.

Example
```
CLS
LOOP
PRINT “Height”; senseHeight
REPEAT
END
```

Associated
senseHumidity, sensePressure, senseTemperature

SENSEHFLIP

Purpose
Horizontally flips the Raspberry Pi senseHAT LED matrix.

Syntax
```
SENSEHFLIP
```

Description

Reverses (flips) the LED matrix display horizontally.

Example
```
CLS
SenseRGBcolour(255,0,0)
senseLine(0,0,0,7)
SenseRGBcolour(0,255,0)
senseLine(0,7,7,7)
LOOP
senseHflip
WAIT(1)
SenseVflip
WAIT(1)
END
```

Associated
senseCls, sensePlot, senseRect, senseScroll, senseVflip, senseRGBcolour, senseGetRGB, , senseLine

SENSEHUMIDITY

Purpose
Returns the value of the Raspberry Pi senseHAT humidity sensor.

Syntax
```
value=SENSEHUMIDITY
```

Description

The Raspberry Pi senseHAT has a number of built in sensors. Humidity can be accessed with this function.

Example
```
CLS
LOOP
PRINT “Humidity”; senseHumidity
REPEAT
END
```

Associated
senseHeight, sensePressure, senseTemperature

SENSELINE

Purpose
Lights a line on the Raspberry Pi senseHAT LED matrix.

Syntax
SENSELINE(*x1, y1, x2, y2*)

Description

Sets the RGB values, defined by senseRGBcolour, to a line of LEDs on the Raspberry Pi senseHAT LED matrix. The line is displayed from x1, y1 to x2, y2.

Example
```
CLS
SenseRGBcolour(255,0,0)
senseLine(0,0,0,7)
SenseRGBcolour(0,255,0)
senseLine(0,7,7,7)
LOOP
senseHflip
WAIT(1)
SenseVflip
WAIT(1)
END
```

Associated
senseCls, sensePlot, senseRect, senseScroll, senseVflip, senseRGBcolour, senseGetRGB

SENSEPLOT

Purpose
Lights an LED on the Raspberry Pi senseHAT LED matrix.

Syntax
SENSEPLOT(*xpos, ypos*)

Description

Sets the RGB values, defined by senseRGBcolour, to a single LED on the Raspberry Pi senseHAT LED matrix.

Example
```
CLS
SenseRGBcolour(255,0,0)
sensePlot(0,0)
SenseRGBcolour(0,255,0)
sensePlot(7,7)
END
```

Associated
senseCls, senseHflip, senseRect, senseScroll, senseVflip, senseRGBcolour, senseGetRGB, senseLine

SENSEPRESSURE

Purpose
Returns the value of the Raspberry Pi senseHAT air pressure sensor.

Syntax
```
value=SENSEPRESSURE
```

Description

The Raspberry Pi senseHAT has a number of built in sensors. Air pressure can be accessed with this function.

Example
```
CLS
LOOP
PRINT “Pressure”; sensePressure
REPEAT
END
```

Associated
senseHumidity, senseHeight, senseTemperature

SENSERECT

Purpose
Lights a rectangle on the Raspberry Pi senseHAT LED matrix.

Syntax
SENSERECT(*xpos, ypos, width, height, fill*)

Description

Sets the RGB values, defined by senseRGBcolour, to a rectangle of LEDs on the Raspberry Pi senseHAT LED matrix. The rectangle is displayed from xpos, ypos with a width and height as specified. Fill can be either 0 for an outline or 1 for filled in.

Example
```
CLS
SenseRGBcolour(255,0,0)
senseRect(0,0,7,7,1)
SenseRGBcolour(0,255,0)
senseRect(0,0,7,7,0)
END
```

Associated
senseCls, sensePlot, senseLine, senseScroll, senseVflip, senseRGBcolour, senseGetRGB

SENSERGBCOLOUR

Purpose

Set the Red, Green and Blue values used by a Raspberry Pi senseHAT LED.

Syntax

SENSERGBCOLOUR(*red, green, blue*)

Description

Sets the Red, Green and Blue values to be used by the senseHAT FUZE BASIC drawing commands.

Example

```
CLS
senseRGBcolour (255, 0, 0)
sensePlot (0, 0)
senseRGBcolour (0, 255, 0)
sensePlot (3, 3)
senseRGBcolour (0, 0, 255)
sensePlot (7, 7)
END
```

Associated

senseCls, sensePlot, senseRect, senseScroll, senseHFlip, senseVflip, senseLine, senseGetRGB

SENSESCROLL

Purpose

Scrolls the Raspberry Pi senseHAT LED matrix.

Syntax

SENSESCROLL(*direction, direction*)

Description

Shifts the Raspberry Pi senseHAT LED matrix in the specified direction by the number of pixels indicated.

Example

```
senseCLS
sensePlot (2, 2)
sensePlot (2, 3)
LOOP
SenseScroll (0, 2) // two up
WAIT(0.1)
SenseScroll (2, 0) // two right
WAIT(0.1)
SenseScroll (0, -2) // two down
WAIT(0.1)
SenseScroll (-2, 0) // two left
WAIT(0.1)
REPEAT
END
```

Associated

senseCls, sensePlot, senseRect, senseRGBcolour, senseHFlip, senseVflip, senseLine, senseGetRGB

SENSETEMPERATURE

Purpose
Returns the value of the Raspberry Pi senseHAT heat sensor.

Syntax
```
value=SENSETEMPERATURE
```

Description

The Raspberry Pi senseHAT has a number of built in sensors. Temperature, in degrees, can be accessed with this function.

Example
```
CLS
LOOP
PRINT “Temperature”; senseTemperature
REPEAT
END
```

Associated
senseHumidity, senseHeight, sensePressure

SENSEVFLIP

Purpose
Vertically flips the Raspberry Pi senseHAT LED matrix.

Syntax
```
SENSEHFLIP
```

Description

Reverses (flips) the LED matrix display vertically.

Example
```
CLS
SenseRGBcolour(255,0,0)
senseLine(0,0,0,7)
SenseRGBcolour(0,255,0)
senseLine(0,7,7,7)
LOOP
senseHflip
WAIT(1)
SenseVflip
WAIT(1)
END
```

Associated
senseCls, sensePlot, senseRect, senseScroll, senseVflip, senseRGBcolour, senseGetRGB, senseLine

SETCHANVOL

Purpose
Set the volume of a sound sample.

Syntax
SETCHANVOL(*channel*,*volume*)

Description
Sets the sound sample playback volume on the specified *channel* where *volume* is a percentage of the maximum (0-100)

Example
```
channel=0
chomp=LOADSAMPLE("pacman_chomp.wav")
FOR volume=10 TO 100 STEP 10 CYCLE
  SETCHANVOL(channel,volume)
  PLAYSAMPLE(chomp,0,0)
  WAIT(1)
REPEAT
END
```

Associated
LOADSAMPLE, PAUSECHAN, PLAYSAMPLE, RESUMECHAN, STOPCHAN

SETMODE

Purpose
Set display width and height

Syntax
SETMODE(*width, height*)

Description
Sets the display width and height to the specified. It is generally sensible to use standard screen display sizes

Example
```
SETMODE( 1280, 720 )
PRINT "Hello World"
WAIT( 2 )
SETMODE( 640, 480 )
PRINT "Hello Another World"
END
```

Associated
GHEIGHT, GWIDTH

SETMOUSE

Purpose
Move the mouse pointer to the specified point.

Syntax
SETMOUSE(*xpos*,*ypos*)

Description
Moves the mouse pointer to the screen coordinate (*xpos*,*ypos*)

Example
```
CLS
MOUSEON
COLOUR=WHITE
RECT(100,100,150,50,TRUE)
INK=BLACK
PAPER=WHITE
PRINTAT(7,38); PRINT "Click Me"
UPDATE
Clicked=FALSE
CYCLE
 GETMOUSE(X,Y,Z)
 IF Z <> 0 THEN
  IF (X > 100 AND X < 250) THEN
    IF (Y > 100 AND Y < 150) THEN
      Clicked=TRUE
    ENDIF
  ENDIF
 ENDIF
REPEAT UNTIL Clicked
SETMOUSE(GWIDTH/2,GHEIGHT/2)
END
```

Associated
GETMOUSE, MOUSEOFF, MOUSEON, MOUSEX, MOUSEY

SETMUSICVOL

Purpose
Sets the music playback volume.

Syntax
SETMUSICVOL(*level*)

Description
Sets the music playback volume where *level* is a percentage of the maximum (0-100)

Example
```
handle=LOADMUSIC("takeoff.wav")
SETMUSICVOL(70)
PLAYMUSIC(handle,1)
```

Associated
LOADMUSIC, PAUSEMUSIC, RESUMEMUSIC, STOPMUSIC

SETSPRITEALPHA

Purpose
Sets the transparency of a sprite

Syntax
SETSPRITEALPHA(*sprite, alpha*)

Description
Sets how transparent a sprite is. An alpha of 0 means it's invisible and 255 means it's completely opaque, or solid.

Example
```
pic = NEWSPRITE( 1 )
LOADSPRITE( "/usr/share/fuze/logo.bmp",
pic, 0 )
FOR alpha = 0 TO 255 CYCLE
SETSPRITEALPHA( pic, alpha )
PLOTSPRITE( pic, GWIDTH / 2, GHEIGHT / 2, 0
)
UPDATE
REPEAT
END
```

Associated
GETSPRITEH, GETSPRITEW, LOADSPRITE, NEWSPRITE, PLOTSPRITE, SETSPRITEANGLE, SETSPRITEFLIP, SETSPRITEORIGIN, GETSPRITEX, GETSPRITEY

SETSPRITEANGLE

Purpose
Rotate a sprite to the given angle

Syntax
SETSPRITEANGLE(*sprite, angle*)

Description
Use to rotate the specified sprite to the given angle in degrees. 0 is default.

Example
```
pic = NEWSPRITE( 1 )
LOADSPRITE( "/usr/share/fuze/logo.bmp",
pic, 0 )
FOR angle = 0 TO 360 CYCLE
SETSPRITEANGLE( pic, angle )
PLOTSPRITE( pic, GWIDTH / 2, GHEIGHT / 2, 0
)
UPDATE
REPEAT
END
```

Associated
GETSPRITEH, GETSPRITEW, LOADSPRITE, NEWSPRITE, PLOTSPRITE, SETSPRITESIZE, SETSPRITEFLIP, SETSPRITEORIGIN, GETSPRITEX, GETSPRITEY

SETSPRITEFLIP

Purpose
Mirror a sprite in the specified direction

Syntax
SETSPRITEFLIP(*sprite, flip*)

Description
Graphically mirrors (flips) the specified sprite.

0 Reset to default
1 mirrored vertically
2 mirrored horizontally
3 mirrored vertically & horizontally

Example
```
pic = NEWSPRITE( 1 )
LOADSPRITE("/usr/share/fuze/logo.bmp",pic, 0)
PLOTSPRITE(pic, gWidth / 2, gHeight / 2, 0)
FOR a=3 TO 0 step -1 cycle
SETSPRITEFLIP( pic, a )
UPDATE
WAIT( 1 )
REPEAT
```

Associated
GETSPRITEH, GETSPRITEW, LOADSPRITE, NEWSPRITE, PLOTSPRITE, SETSPRITEANGLE, SETSPRITESIZE, SETSPRITEORIGIN, GETSPRITEX, GETSPRITEY

SETSPRITEORIGIN

Purpose
Sets the anchor point of a sprite

Syntax
SETSPRITEorigin(*sprite, xpos, ypos*)

Description
Use to set the origin of a specified sprite. When plotting a sprite its default origin is bottom left. You can change this to any point on the sprite. This example sets it to the middle.

Example
```
pic = NEWSPRITE( 1 )
LOADSPRITE( "logo.png", pic, 0 )
LOOP
SETSPRITEORIGIN( pic, 0, 0 )
Plotsprite( pic, 0, 0, 0)
Update
Wait(1)
MiddleX=GETSPRITEW(pic)/2
MiddleY=GETSPRITEH(pic)/2
SETSPRITEORIGIN( pic, MiddleX, MiddleY )
PLOTSPRITE( pic, 0, 0, 0)
UPDATE
REPEAT
```

Associated
GETSPRITEH, GETSPRITEW, LOADSPRITE, NEWSPRITE, PLOTSPRITE, SETSPRITEANGLE, SETSPRITEFLIP, SETSPRITESIZE, GETSPRITEX, GETSPRITEY

SETSPRITESIZE

Purpose
Change the size of a sprite

Syntax
SETSPRITESIZE(*sprite, size*)

Description
Sets the sprite to the specified *size* in percent. 100 is the default, therefore 50 is half the size and 300 is three times as big as the original.

Example
```
pic = NEWSPRITE( 1 )
LOADSPRITE("/usr/share/fuze/logo.bmp",pic, 0)
FOR angle = 50 TO 200 CYCLE
SETSPRITESIZE( pic, size )
PLOTSPRITE( pic, GWIDTH / 2, GHEIGHT / 2, 0)
UPDATE
REPEAT
END
```

Associated
GETSPRITEH, GETSPRITEW, LOADSPRITE, NEWSPRITE, PLOTSPRITE, SETSPRITEANGLE, SETSPRITEFLIP, SETSPRITEORIGIN, GETSPRITEX, GETSPRITEY

SETSPRITETRANS

Purpose
Set a transparency colour for a given sprite and its sub-sprites.

Syntax
SETSPRITETRANS(*spriteId, Red, Green, Blue*)

Description
If you specify a transparency colour for a sprite then any pixels in the sprite that are this colour will be transparent i.e. They will show as the background colour. This allows a sprite to pass over a background image without blocking it out. You need to load the sprite files first before setting the transparency colour.

Example (overleaf)

SETSPRITETRANS Example

```
CLS
RGBCOLOUR(247, 247, 247)
RECT(50, 50, 101, 101, TRUE)
COLOUR = YELLOW
CIRCLE(100, 100, 50, TRUE)
SAVEREGION("s3.bmp", 50, 50, 101, 101)
CLS2
COLOUR = RED
MidY = GHEIGHT / 2
FOR X = 0 TO GWIDTH STEP 100 CYCLE
  RECT(X, MidY - 50, 50, 200, TRUE)
REPEAT
COLOUR = Black
s1 = NEWSPRITE(1)
LOADSPRITE("s3.bmp", s1, 0)
SETSPRITETRANS(s1, 247, 247, 247)
FOR X = 0 TO gWidth STEP 10 CYCLE
  PLOTSPRITE(s1, X, GHEIGHT / 2, 0)
  UPDATE
  WAIT (0.0005)
REPEAT
END
```

Associated

GETSPRITEH, GETSPRITEW, LOADSPRITE, NEWSPRITE, PLOTSPRITE, RGBCOLOUR

SGET

Purpose

Read a byte from a serial port.

Syntax

byte=SGET(*handle*)

Description

Fetch a single byte of data from an open serial port and return the data as a number. This function will pause program execution for up to 5 seconds if no data is available. If there is still not data after 5 seconds, the function will return -1.

Example

```
REM Read a byte from a serial port
arduino=SOPEN("/dev/ttyUSB0", 115200)
byte=SGET(arduino)
PRINT byte
SCLOSE(arduino)
END
```

Associated

SCLOSE, SGET$, SOPEN, SPUT, SPUT$, SREADY

SGET$

Purpose
Read a character from a serial port.

Syntax
character=SGET$(*handle*)

Description
Fetch a single byte of data from an open serial port and return the data as a single character string. This function will pause program execution for up to 5 seconds if no data is available. If there is still not data after 5 seconds, the function will return an empty string.

Example
```
REM Read a character from a serial port
arduino=SOPEN("/dev/ttyUSB0", 115200)
char$=SGET$(arduino)
PRINT char$
SCLOSE(arduino)
END
```

Associated
SCLOSE, SGET, SOPEN, SPUT, SPUT$, SREADY

SGN

Purpose
Returns the sign of the specified number.

Syntax
sign=SGN(*number*)

Description
Returns -1 if the number is negative, 1 otherwise. (Zero is considered positive)

Example
```
REM Prints 1
PRINT SGN(100)
PRINT SGN(0)
REM Print -1
PRINT SGN(-5)
END
```

Associated
ABS

SHOWKEYS

Purpose
List the function key definitions

Syntax
SHOWKEYS

Description
It is possible to set the 12 function keys at the top of the keyboard to user defined values. This command will show what the current definitions are. By default function key **F2** is defined to enter the **EDIT** command and **F3** the **RUN** command but it is possible to overwrite these.

Example
```
REM Set key F5 to clear screen
keyF5$ = "CLS\n"
SHOWKEYS
```


SIN

Purpose
Returns the sine of the given angle.

Syntax
sine=SIN(*angle*)

Description
Returns the sine of the argument *angle* in radians. This is the ratio of the side of a right angled triangle, that is opposite to the angle, to the hypotenuse (the longest side).

Example
```
REM Draw an ellipse in the screen centre
CLS
DEG
FOR Angle=0 TO 360 CYCLE
  Xpos=100*COS(Angle)+GWIDTH/2
  Ypos=50*SIN(Angle)+GHEIGHT/2
  PLOT(Xpos,Ypos)
REPEAT
END
```

Associated
ASIN, ATAN, COS

SOFTPWMWRITE

Purpose
Synthesize Pulse Wave Modulation to the specified GPIO pin.

Syntax
SOFTPWMWRITE(*pinNo*,*value*)

Description
This enables you to simulate an analog output. For example instead of an LED being just on or off you can make it appear brighter or dimmer. The *pinNo* parameter is the number of the GPIO output pin. This must first be set to *pinSoftPwm*. The *value* parameter is a percentage (0-100).

Example
```
REM Connect an LED to GPIO pin 0
PINMODE (0, PINSOFTPWM)
FOR I=0 TO 100 CYCLE
  SOFTPWMWRITE (0,I)
  WAIT(0.1)
REPEAT
FOR I=100 to 0 STEP -1 CYCLE
  SOFTPWMWRITE (0,I)
  WAIT(0.1)
REPEAT
END
```

Associated
PINMODE

SOPEN

Purpose
Opens a serial device and makes it available for use.

Syntax
handle=SOPEN(*$device*,*speed*)

Description
This opens a serial device and makes it available for our use. It takes the name of the serial port and the speed as an argument and returns a number (the handle) of the device. We can use this handle to reference the device and allow us to open several devices at once.
The following baud rates are recognised: 50, 75, 110, 134, 150,200, 300, 600, 1200, 1800, 2400, 19200, 38400 57600, 115200 and 230400, but do check your local PC and devices capabilities. The device is always opened with the data format set to 8 data bits, 1 stop bit and no parity. All handshaking is turned off.

Example
```
REM Read a byte from a serial port
arduino=SOPEN("/dev/ttyUSB0", 115200)
byte=SGET(arduino)
SCLOSE(arduino)
END
```

Associated
SCLOSE, SGET, SGET$, SPUT, SPUT$, SREADY

SOUND

Purpose
Play a synthesised sound with the specified parameters (emulates BBC BASIC SOUND command).

Syntax
sound(*channel,amplitude,pitch,duration*)

Description
The *channel* is 0 to 3 with 0 being for white noise (static). The *amplitude* goes from 0 to -15 where 0 is silent and -15 is full volume (-7 being half volume and so on). The *duration* is in 20ths of a second, so 20 represents one second, 10 half a second and so on. The *pitch* values are taken from a table which can be found at the end of the manual but, middle C has a value of 53 and each note is 4 away from the next. Note that when used with a sound envelope *amplitude* is replaced by the envelope number.

Example
```
Channel=1
Volume=-10
FOR Note=1 TO 5 CYCLE
  READ Frequency,Duration
  SOUND(Channel,Volume,Frequency,Duration)
REPEAT
END
DATA 89, 20, 97, 20, 81, 20, 33, 20, 61, 40
```

Associated
ENVELOPE, TONE

SPACE$

Purpose
Returns a blank string of the specified length.

Syntax
blankstring=SPACE$(*number*)

Description
Returns a string of blank spaces *number* characters long.

Example
```
Blank$ = SPACE$(100)
FOR I=1 TO 20 CYCLE
  PRINT Blank$
REPEAT
END
```

SPRITECOLLIDE

Purpose
Detect a sprite collision (fast bounding box)

Syntax
collision=SPRITECOLLIDE(*target*)

Description
Returns the sprite index of the first sprite that the sprite index *target* has collided with, or -1 if there is no collision. It only checks the current sprite location and this is only updated after a screen update cycle so it is possible to call PLOTSPRITE() and have a sprite overlap but it not be detected until after the update has happened on screen.

Example
```
CLS
RGBCOLOUR(254,254,254)
RECT(50,50,101,101,TRUE)
COLOUR=YELLOW
CIRCLE(100,100,50,TRUE)
SAVEREGION("s1.bmp",50,50,101,101)
COLOUR=RED
CIRCLE(100,100,50,TRUE)
SAVEREGION("s2.bmp",50,50,101,101)
CLS2
s1=NEWSPRITE(1)
s2=NEWSPRITE(1)
LOADSPRITE("s1.bmp",s1,0)
LOADSPRITE("s2.bmp",s2,0)
FOR X=0 TO GWIDTH STEP 1 CYCLE
  PLOTSPRITE(s1,X,200,0)
  PLOTSPRITE(s2,GWIDTH-X-100,200,0)
  IF SPRITECOLLIDE (s2) <> -1 THEN BREAK
  UPDATE
  WAIT(0.0005)
REPEAT
END
```

Associated
GETSPRITEH, GETSPRITEW, HIDESPRITE, LOADSPRITE, NEWSPRITE, PLOTSPRITE, SPRITECOLLIDEPP

SPRITECOLLIDEPP

Purpose
Detect a sprite collision (pixel perfect)

Syntax
collision=SPRITECOLLIDEPP(*target*,*accuracy*)

Description
This is a slower but more accurate version of SpriteCollide. It first does do a simple bounding box test then checks row at a time. The *accuracy* parameter is how many pixels to skip both horizontally and vertically. This is from 1 to 16, where 1 is "perfect" and greater than 1 is less accurate, but faster. This will affect the amount of visible overlap you get before a collision is detected.

Example
```
CLS
RECT(50,50,101,101,TRUE)
COLOUR=YELLOW
CIRCLE(100,100,50,TRUE)
SAVEREGION("s1.bmp",50,50,101,101)
COLOUR=RED
CIRCLE(100,100,50,TRUE)
SAVEREGION("s2.bmp",50,50,101,101)
CLS2
s1=NEWSPRITE(1)
s2=NEWSPRITE(1)
LOADSPRITE("s1.bmp",s1,0)
LOADSPRITE("s2.bmp",s2,0)
FOR X=0 TO GWIDTH STEP 1 CYCLE
  PLOTSPRITE(s1,X,200,0)
  PLOTSPRITE(s2,GWIDTH-X-100,200,0)
  IF SPRITECOLLIDEPP(s2,1) <> -1 THEN BREAK
  UPDATE
REPEAT
END
```

Associated
GETSPRITEH, GETSPRITEW, HIDESPRITE, LOADSPRITE, NEWSPRITE, PLOTSPRITE, SPRITECOLLIDE

SPRITEOUT

Purpose
Find out if a sprite is off screen

Syntax
result = SPRITEOUT(*sprite*)

Description
If the sprite is off screen then result is true otherwise it is false.

Example
```
pic = NEWSPRITE( 1 )
LOADSPRITE("/usr/share/fuze/logo.bmp",pic, 0)
PLOTSPRITE( pic, GWIDTH / 2, GHEIGHT / 2, 0)
WHILE NOT SPRITEOUT( pic ) CYCLE
ADVANCESPRITE( pic, 2 )
UPDATE
REPEAT
PRINT "GONE!"
END
```

Associated
HIDESPRITE, PLOTSPRITE

SPUT

Purpose
Send a byte to an open serial port.

Syntax
SPUT(*arduino*,*byte*)

Description
Send a single byte of data to an open serial port.

Example
```
REM Write a byte to a serial port
arduino=SOPEN("/dev/ttyUSB0", 115200)
SPUT(arduino,52)
SPUT(arduino,50)
SCLOSE(arduino)
END
```

Associated
SCLOSE, SGET, SGET$, SOPEN, SPUT$, SREADY

SPUT$

Purpose
Send a character string to an open serial port.

Syntax
SPUT$(*arduino*,*string*)

Description
Send a string of characters of data to an open serial port.

Example
```
REM Write a byte to a serial port
arduino=SOPEN("/dev/ttyUSB0", 115200)
SPUT$(arduino,"Hello")
SCLOSE(arduino)
END
```

Associated
SCLOSE, SGET, SGET$, SOPEN, SPUT, SREADY

SQRT

Purpose
Return the square root of the specified number.

Syntax
squareroot=SQRT(*number*)

Description
Returns the square root of the argument *number*. This is the opposite of multiplying a number by itself i.e. X = SQRT(X * X)

Example
```
foursquared=4*4
PRINT SQRT(foursquared)
END
```

SREADY

Purpose
Get the number of characters available to be read on an open serial port.

Syntax
count=SREADY(*handle*)

Description
Returns the number of characters available to be read from an open serial port. This can be used to poll the device to avoid stalling your program when there is no data available to be read.

Example
```
REM Read a character from a serial port
arduino=SOPEN("/dev/ttyUSB0", 115200)
IF SREADY(arduino) THEN
  char$=SGET$(arduino)
  PRINT char$
ENDIF
SCLOSE(arduino)
END
```

Associated
SCLOSE, SGET, SGET$, SOPEN, SPUT, SPUT$

STOP

Purpose
Stop a running program.

Syntax
STOP

Description
Program execution is stopped with a message indicating the current line number.

Example
```
INPUT "Enter the Password: ", pass$
IF pass$<>"wibble" THEN
  PRINT "Password Incorrect"
  STOP
ENDIF
PRINT "Password Correct"
END
```

Associated
CONT

STOPCHAN

Purpose
Stop the playing of a sound sample.

Syntax
STOPCHAN(*handle*)

Description
This function stops the playing of the sound sample associated with the *handle* returned by *LOADSAMPLE* that has been started using *PLAYSAMPLE*. It cannot be resumed once stopped.

Example
```
channel=0
volume=70
SETCHANVOL(channel,volume)
intro=LOADSAMPLE("pacman_intro.wav")
PLAYSAMPLE(intro,channel,0)
WAIT(3)
STOPCHAN(intro)
END
```

Associated
LOADSAMPLE, PAUSECHAN, PLAYSAMPLE, RESUMECHAN, SETCHANVOL

STOPMUSIC

Purpose
Stop music playing completely.

Syntax
STOPMUSIC

Description
Stops a playing music track which cannot then be resumed.

Example
```
handle=LOADMUSIC("takeoff.wav")
SETMUSICVOL(70)
PLAYMUSIC(handle,1)
STOPMUSIC
```

Associated
LOADMUSIC, PAUSEMUSIC, RESUMEMUSIC, SETMUSICVOL

STR$

Purpose
Returns a string version of the supplied number.

Syntax
*string$*=STR$(*number*)

Description
Returns a string in the decimal (base 10) representation of *number*. This is useful if you want to append a number to a string. This is the opposite of the VAL function.

Example
```
PRINT "The Answer is "+STR$(42)
END
```

Associated
VAL

SWAP

Purpose
Swap the value of two variables.

Syntax
SWAP(*value1*,*value2*)

Description
This swaps the value of the 2 variables round. Both arguments must be the same type - i.e. Both numeric or both string.

Example
```
lowest=99
highest=0
IF lowest>highest THEN
  SWAP(highest,lowest)
ENDIF
PRINT "Lowest "; lowest
PRINT "Highest ";highest
END
```

SWITCH

Purpose

Test a value against many different values and execute different code.

Syntax

```
SWITCH (variable)
  { CASE value {, value }
    commands
  ENDCASE }
  [ DEFAULT
    commands
  ENDCASE ]
ENDSWITCH
```

Description

Simplify the writing of multiple IF. . . THEN. . . ELSE statements. Rules

- Every SWITCH must have a matching ENDSWITCH.
- Every CASE or DEFAULT statement must have a matching ENDCASE.
- Statements after a CASE statement must not run-into another CASE.
- The constants after the CASE statement (and the expression in the SWITCH statement) can be either numbers or strings, but you can't mix both.

Example

```
INPUT a
SWITCH(a)
  CASE 1,2
    PRINT "You entered 1 or 2"
  ENDCASE
  CASE 7
    PRINT "You entered 7"
  ENDCASE
  DEFAULT
    PRINT "You entered something else"
  ENDCASE
ENDSWITCH
END
```

Associated

ELSE, IF THEN

TAN

Purpose
Return the tangent of the given angle.

Syntax
tangent=TAN(*angle*)

Description
Returns the tangent of the *angle* in the current angle units. In a right-angled triangle the tangent of an angle is the ratio of the length of the opposite side to the length of the adjacent side. This is a measure of the steepness of an angle.

Example
```
DEG
PRINT "Tangent of 45 degrees: "; TAN(45)
PRINT "ArcTangent of 1: "; ATAN(1)
END
```

Associated
ACOS, ASIN, ATAN, COS

TANGLE

Purpose
Read or set the current angle of the turtle.

Syntax
angle=TANGLE
TANGLE=*angle*

Description
This can be read or assigned to and represents the current angle of the turtle when using turtle graphics mode (in the current angle units)

Example
```
CLS
ORIGIN(GWIDTH/2,GHEIGHT/2)
CLOCK
FOR I = 1 TO 60 CYCLE
  TANGLE = I
  MOVETO(0,0)
  PENDOWN
  MOVE(100)
REPEAT
END
```

Associated
LEFT, MOVE, MOVETO, PENDOWN, PENUP, RIGHT

THEIGHT

Purpose
The height in characters of the display.

Syntax
height=THEIGHT

Description
The height in characters of the display.

Example
```
CLS
text$="This text is centred in the screen"
HVTAB((TWIDTH-LEN(text$))/2,THEIGHT/2)
PRINT text$
END
```

Associated
TWIDTH

TIME

Purpose
Find out how long the program has been running.

Syntax
time=TIME

Description
This returns a number which represents the time that your program has been running in milliseconds.

Example
```
REM Simple reaction timer
WAIT(2)
REM Make sure no key pressed
WHILE INKEY<>-1 CYCLE
REPEAT
stime=TIME
PRINT "Go!"
WHILE INKEY=-1 CYCLE
REPEAT
etime = TIME
PRINT "Your reaction time is ";
PRINT etime-stime; " milliseconds"
END
```

TIME$

Purpose
Returns a string with the current time.

Syntax
*now$*=TIME$

Description
This returns a string with the current time in the following format: HH:MM:SS. For example: 18:05:45.

Example
```
PRINT "The time now is ";
PRINT TIME$
END
```

Associated
DATE$

TONE

Purpose
Play a tone with the specified parameters.

Syntax
TONE(*channel*,*volume*,*frequency*,*duration*)

Description
This plays a simple tone of the given *frequency* (1 to 5000Hz), *volume* (%) and *duration* (0.01 to 20 seconds) on the given *channel*.
You can play multiple tones by playing them one after the other and up to three tones can be played simultaneously on different channels.
Channel 0 is white noise and the frequency has no bearing on the sound produced.

Example
```
Channel=1
Volume=70
FOR Note=1 TO 5 CYCLE
  READ Frequency,Duration
  TONE(Channel,Volume,Frequency,Duration)
REPEAT
END
DATA 440, 1, 493, 1, 392, 1, 196, 1, 294 ,2
```

Associated
SOUND

TRIANGLE

Purpose
Draw a triangle on the screen.

Syntax
TRIANGLE (*xpos1, ypos1, xpos2, ypos2, xpos3, ypos3, fill*)

Description
Draws a triangle with its corners at the three given points. The final parameter, *fill* is either TRUE or FALSE, and specifies filled (TRUE) or outline (FALSE).

Example
```
CYCLE
  COLOUR=RND(16)
  x1=RND(GWIDTH)
  x2=RND(GWIDTH)
  x3=RND(GWIDTH)
  y1=RND(GHEIGHT)
  y2=RND(GHEIGHT)
  y3=RND(GHEIGHT)
  f=RND(2)
  TRIANGLE (x1,y1,x2,y2,x3,y3,f)
  UPDATE
  IF INKEY<>-1 THEN BREAK
REPEAT
END
```

Associated
CIRCLE, ELLIPSE, RECT

TRUE

Purpose
Represents the logical "true" value.

Syntax
TRUE

Description
Represents a Boolean value that succeeds a conditional test. It is equivalent to a numeric value of 1 (in fact anything other than 0 evaluates to TRUE)

Example
```
condition=TRUE
IF condition=TRUE THEN
  PRINT "Condition is TRUE"
ENDIF
PRINT condition
END
```

Associated
FALSE

TWIDTH

Purpose
The width in characters of the display.

Syntax
width=TWIDTH

Description
The width in characters of the display.

Example
```
text$="This text is centred horizontally"
HTAB=(TWIDTH-LEN(text$))/2
PRINT text$
END
```

Associated
THEIGHT

UPDATEMODE

Purpose
Set the video update mode.

Syntax
UPDATEMODE=*mode*

Description (overleaf)

UPDATEMODE Description
The updateMode determines when the screen is redrawn. Redrawing the screen takes a little time and will slow down a program if you do it too often. The value of the *mode* parameter can be **0**, **1**, or **2** as follows:
0- automatic updates do not happen. Nothing will be drawn on the screen until the UPDATE command is issued.
1-This is the default mode whereby an UPDATE happens automatically when you output a new line, or the screen scrolls.
2-The screen is updated after every PRINT instruction whether it takes a new line or not.

Example
```
CLS
INPUT "Update Mode? ",mode
IF mode>=0 AND mode<=2 THEN
  PRINT "Press space to exit"
  WAIT(1)
  UPDATEMODE=mode
  CYCLE
    PRINT "Hello World ";
  REPEAT UNTIL INKEY=32
  UPDATE
ELSE
  PRINT "Invalid Update Mode"
ENDIF
WAIT(1)
END
```

Associated
UPDATE

UNTIL REPEAT

Purpose
Loop until the specified condition is met.

Syntax
```
UNTIL condition CYCLE
  {statements}
REPEAT
```

Description
Execute the *statements* zero or more times until the *condition* is TRUE (Not 0).
Because the test is done at the start of the loop the *statements* may not be executed at all

Example
```
REM Print 1 to 10
count=1
UNTIL count>10 CYCLE
  PRINT count
  count=count + 1
REPEAT
END
```

Associated
BREAK, CONTINUE, CYCLE, CYCLE REPEAT, FOR REPEAT, REPEAT UNTIL, WHILE REPEAT

UPDATE

Purpose
Update screen graphics.

Syntax
```
UPDATE
```

Description
Graphics are drawn to a temporary screen buffer rather than the visible screen. The UPDATE command copies the working area to the main display. An update is also performed if your program stops for input, or when you PRINT a new line.

Example
```
REM Moire patterns
CYCLE
  CLS
  COLOUR=RND(15)+1
  x=RND(GWIDTH)
  y=RND(GHEIGHT)
  FOR w=0 TO GWIDTH-1 STEP 3 CYCLE
    LINE(x,y,w,0)
    LINE(x,y,w,GHEIGHT-1)
  REPEAT
  FOR h=0 TO GHEIGHT-1 STEP 3 CYCLE
    LINE(x,y,0,h)
    LINE(x,y,GWIDTH-1,h)
  REPEAT
  UPDATE
REPEAT
END
```

Associated
UPDATEMODE

VAL

Purpose
Returns the number represented by a character string.

Syntax
number=VAL(*string$*)

Description
Returns the *number* represented by *string$*. This is the opposite of the STR$ function.

Example
```
now$ =TIME$
hh=VAL(LEFT$(now$, 2))
mm=VAL(MID$(now$, 3, 2))
ss=VAL(RIGHT$(now$, 2))
elapsed=hh*3600+mm*60+ss
PRINT "Seconds since midnight: "; elapsed
END
```

Associated
STR$

VLINE

Purpose
Draws a vertical line.

Syntax
VLINE (*ypos1*,*ypos2*,*xpos*)

Description
Draws a vertical line on column *xpos*, from row *ypos1* to row *ypos2*.

Example
```
CLS
COLOUR=red
FOR xpos=0 TO GWIDTH STEP 100 CYCLE
  VLINE(0, GHEIGHT, xpos)
REPEAT
UPDATE
END
```

Associated
HLINE, LINE, LINETO

VTAB

Purpose
Set/Read the current text cursor vertical position.

Syntax
VTAB=*value*
value=VTAB

Description
Set/Read the current text cursor vertical position.

Example
```
CLS
FOR ypos = 0 TO THEIGHT CYCLE
  VTAB=ypos
  PRINT VTAB
REPEAT
VTAB=0
END
```

Associated
HTAB, HVTAB

WAIT

Purpose
Waits for the specified time to elapse.

Syntax
WAIT(*time*)

Description
This waits (does nothing) for *time* seconds. This may be a fractional number, but the accuracy will depend on the computer you are running it on, however delays down to 1/100th of a second should be achievable.

Example
```
REM COUNT 10 Seconds
CLS
Seconds = 0
FOR I=1 TO 10 CYCLE
  WAIT(1)
  Seconds=Seconds + 1
  HVTAB(10,10)
  PRINT Seconds
REPEAT
PRINT "Elapsed "; TIME/1000
END
```

WHILE REPEAT

Purpose
Loop while the specified condition is met.

Syntax
```
WHILE condition CYCLE
  {statements}
REPEAT
```

Description
Execute the *statements* zero or more times while the *condition* is TRUE (Not 0). Because the test is done at the start of the loop the *statements* may not be executed at all.

Example
```
handle=OPEN("whiletest.txt")
FOR r=0 TO 10 CYCLE
  PRINT# handle,"Record ";r
REPEAT
CLOSE(handle)
handle = OPEN("whiletest.txt")
WHILE NOT EOF(handle) CYCLE
  INPUT# handle, record$
  PRINT record$
REPEAT
CLOSE (handle)
END
```

Associated
BREAK, CONTINUE, CYCLE, CYCLE REPEAT, FOR REPEAT, REPEAT UNTIL, UNTIL REPEAT

ScanKeyboard values

scanBackspace
scanTab
scanClear
scanReturn
scanPause
scanEscape
scanSpace
scanExclaim
scanQuoteDbl
scanHash
scanDollar
scanAmpersand
scanQuote
scanLeftParen
scanRightParen
scanAsterisk
scanPlus
scanComma
scanMinus
scanPeriod
scanSlash
scan0
scan1
scan2
scan3
scan4
scan5
scan6
scan7
scan8
scan9
scanColon
scanSemiColon
ScanLess
ScanEquals
scanGreater
scanQuestion
scanAt
scanLeftBracket
scanBackSlash
scanRightBracket
scanCaret
ScanUnderscore
scanBackQuote
scanA
scanB
scanC
scanD
scanE
scanF
scanG
scanH
scanI
scanJ
scanK
scanL
scanM
scanN
scanO
scanP
scanQ
scanR
scanS
scanT
scanU
scanV
scanV
scanW
scanX
scanY
ScanZ
scanDelete
scanKP0
scanKP1
scanKP2
scanKP3
scanKP4
scanKP5
scanKP6
scanKP7
scanKP8
scanKP9
scanKpPeriod
scanKpDivide
scanKpMultiply
scanKpMinus
scanKpPlus
ScanKpEnter
ScanKpEquals
scanUp
scanDown
scanRight
scanLeft
scanInsert
scanHome
scanEnd
scanPageup
scanPagedown
scanF1
scanF2
scanF3
scanF4
scanF5
scanF6
scanF7
ScanF8
scanF9
scanF10
scanF11
scanF12
scanF13
scanF14
ScanF15
scanNumLock
scanCapsLock
scanScrolLock
scanRShift
scanLShift
scanRCtrl
scanLCtrl
scanRAlt
scanLAlt
scanRMeta
scanLMeta
scanLSuper
scanRSuper
scanMode
scanCompose
scanHelp
scanPrint
scanSysReq
scanBreak
scanMenu
scanPower
scanEuro
scanUndo

Notes & Octaves for Sound function

Note	Octave Number						
	1	2	3	4	5	6	7
B	0	49	97	145	193	241	
A#	0	45	93	141	189	237	
A		41	89	137	185	233	
G#		37	85	133	181	229	
G		33	81	129	177	225	
F#		29	77	125	173	221	
F		25	73	121	169	217	
E		21	69	117	165	213	
D#		17	65	113	161	209	
D		13	61	109	157	205	253
C#		9	57	105	153	201	249
C		5	53	101	149	197	245

FUZE BASIC
Programmer's reference guide
£14.99 / $24.99 / €21.99

FUZE BASIC

Programmer's Reference Guide

fuze.co.uk