# Welcome to BASin!

BASin is a program that helps you to write programs in BASIC. It is made by a company called Retro Computers Ltd, and with it you can learn how to make your own programs and games.

BASIC is the name of the language that BASin uses. It stands for Beginners Allpurpose Symbolic Instruction Code - which means that people who have never tried to write their own programs before will find it easy to understand and use. This version of BASin is called "BASin for Beginners" because it has some changes that are designed to make learning as easy as possible. While there are other languages that you can use, we would recommend that you start here. The things that you learn with BASin for Beginners will help you when you feel you are ready to learn more about programming on your PC.

Before you start, you may want to read the following topics that explain what BASin does and how the BASIC language started:

- What is BASin?
- What is a Sinclair Spectrum?
- History of Sinclair BASIC.
- How BASin differs from a real ZX Spectrum.

To get help on a particular topic

To learn how to do something in BASin, use the **Contents** and **Index** areas on the left of this help window. You can look for the topic you want, or enter a codeword on the index tab.

### What is **BASin**?

BASin is a what is called a "Development Environment". It was designed to make programming in BASIC easy.

Way back in the early 1980s, a company called Sinclair Research Ltd made a computer called the ZX Spectrum. Nothing quite like it had been made before, at least not as cheap to buy as Sinclair's computer - so a great many children of that time had one in their houses. For the price, it was very powerful - it had colour graphics, a full keyboard (trust me, that was a big deal back then!) and connected to your living room TV.

AND it played games! Lots of them. Throughout the life of the Spectrum, about 10 years, there were thousands of games made and sold. Because this was a new thing at the time (because games hadn't been around for long) most of them were made by people like you and me - who made programs as a hobby.

That is the real genius of Sinclair's computers. They not only had the ability to play games, they also had a secret weapon built-in: A programming language called BASIC. With BASIC you could give the computer instructions that could be used to make it do anything you wanted - including make games.

BASin runs what is called an "emulator" - that is, a program for a PC that pretends to be another computer. This means that BASin will let you program in the same language as the Spectrum, and that's what makes it easy to do. Programming for a PC is a very complex thing to do, and not something an absolute beginner should attempt. Once they have learnt one language though, it makes it easier to program in another. So starting with Sinclair BASIC on a Spectrum, although quite simple, will open the doors to a world that is so vast and full of fun that you will spend the rest of your life exploring it!

BASin's editor looks like that of the Spectrum, with the addition of codeword help, a "Coding teacher" that checks your program as you write it, and development tools such as a UDG editor and a variable viewer/editor. BASin also provides advanced debugging features: conditional breakpoints, watches, and full program flow control at the line/statement level. Don't worry if you don't understand what all that means - you soon will!

Program development in BASin is easier and faster than it was on the original ZX Spectrum. The Spectrum BASIC manual - an excellent introduction not only to the language, but to programming in general - is included here for beginners, and has been modified where necessary to take account of BASin's features.

### What is a Sinclair Spectrum?

Released in 1982 as a direct rival to Acorn's BBC range of educational computers, the Sinclair ZX Spectrum was a true home computer at an affordable price. It loaded games from a standard cassette player and produced a display on the family television set. Unlike its competitors, the Spectrum was small and black, with rubber keys and a small buzzer for making simple beeping noises - and it was amazingly successful.

The hardware

The Spectrum was based on a combination of ULA (uncommitted logic array) and 3.5 MHz Z80 processor, and initially had 16 kilobytes of RAM (usable memory). This was later increased to 48 KB, which rapidly became the standard.

The screen display measured  $256 \times 192$  pixels, a high resolution for its time, and supported eight colours at two brightness levels, totalling fifteen colours (since bright and dull black were identical). The display was divided into 24 rows and 32 columns of "character cells", each 8 × 8 pixels and capable of containing any two of the basic eight colours (or *attributes*), as long as they were at the same intensity. This led to some "attribute clash", one of the most endured and loved features of the hardware: where two coloured objects collided, their attributes would overlap, giving the impression of coloured squares.

Sound was similarly simple and to the point: a buzzer capable of outputting one channel of beeping. The **BEEP** command was used to drive it from BASIC, but clever machine code routines could produce multi-channel effects and music, which no other computer of the time could rival without dedicated sound hardware.

The BASIC

Spectrum BASIC was more or less unique, as was the method of programming. Keys on the keyboard were imprinted with BASIC keywords, which meant that a single keypress (such as P) could produce an entire keyword (**PRINT**). The use of shift keys made further keywords available. It was a quirky system, but

easy to learn. This style of coding was the standard for all Spectrums prior to the 128K machine.

For more information, see History of Sinclair BASIC.

History of the Spectrum

Although intended as a serious business and educational machine, the Spectrum quickly gained a devoted following of gamers. Because Sinclair had wisely incorporated the BASIC programming language, a legion of bedroom programmers arose almost overnight, advertising their home-grown games cheaply in magazines. Many of today's programmers owe their careers to the Spectrum, having moved from BASIC to Z80 assembly language; and top software houses such as Rare and Bullfrog cut their teeth in the home computer market by writing top-quality Spectrum games.

The Spectrum continued to grow, not only with new incarnations, but with an increasing selection of peripherals. You could buy printers, mass storage devices, joysticks, light-pens, and light-guns - all in the same sleek rounded black style. It was a hobbyist's dream.

In 1984, the Spectrum became the Spectrum+, with a typewriter-style keyboard housing for the still popular 48K machine. The new keyboard kept the "Sinclair" feel and was adequate for single-key programming, if not for touch-typing.

The 128K Spectrum followed in 1986. A clever memory paging mechanism gave this machine a full 128 kilobytes of RAM. The BASIC benefited from a "silicon disk" or RAM disk, allowing programs to be saved to temporary memory almost instantly, and a PLAY command which controlled the three channels of the new AY-3-8912 sound chip.

Later in 1986, Sinclair Research was sold to Amstrad Consumer Electronics, who re-released the 128K machine as the Spectrum +2 with a new ROM, a grey case with a more professional keyboard (reminiscent of the Amstrad CPC), and a built-in cassette recorder. Many considered this to be the death of the Sinclair Spectrum, but Amstrad did eventually address issues with the machine's hardware when they produced the Spectrum +3, complete with 3" disk drive.

Commercial software (mostly games) continued to be released for the Spectrum until the early 1990s. Even now, in the twenty-first century, the Spectrum is almost a household word - and the enthusiasm for it seems likely to continue well into the future.

## **History of Sinclair BASIC**

[From *The History of Sinclair BASIC*, by Andrew Owen.]

Sinclair BASIC needs no introduction to enthusiasts of the little rubber-keyed machine, but its evolution is stranger than you might think.

In July 1975, Micro-Soft, as it was then called, shipped BASIC (Beginner's Allpurpose Symbolic Instruction Code) version 2.0 for the MITS Altair 8800 hobbyist computer. This was the first commercial version of the BASIC programming language, originally developed by J G Kemeny and T E Kurtz in 1964 at Dartmouth College in the United States.

By then, Kemeny and Kurtz had addressed the main criticisms of BASIC - that it lacked structure and encouraged bad programming habits - but the 4K and 8K versions for the Altair, written by Paul Allen and Bill Gates, were based on the original Dartmouth BASIC.

Microsoft BASIC became so popular that it made Gates and Allen their first fortune and was subsequently supplied with the majority of 8-bit computers. So, not surprisingly, when the ANSI Standard for Minimal BASIC (X3.60-1978) was launched, it was based mainly on the Microsoft version.

In May 1979, Clive Sinclair's engineers began work on the machine that would become the ZX80. Sinclair was inspired to create the machine after seeing how much his son enjoyed using a TRS-80 but guessing that many people would be put off buying one because of the high price – just under £500.

Unlike the MK14, Sinclair's previous foray into the computer hobbyist market, this machine would ship with BASIC based on the ANSI standard. But the aim was to keep costs down, and that precluded paying a licence fee to Microsoft. To this end, Sinclair had already met with John Grant of Nine Tiles in April to discuss the software requirements of the ZX80.

Given the tiny R&D; budget, Nine Tiles stood to make hardly any money out of the deal, but the feeling was that the project was exciting and worthwhile, and

one the company would benefit from being associated with.

To achieve the launch price of £79.95 in kit form, RAM was limited to 1K and the integer BASIC had to be crammed into a 4K ROM. Grant wrote the bulk of the ROM between June and July. But the resulting program was 5K in length, so Grant spent that August trimming the code.

According to Cambridge mathematician Steven Vickers, who wrote the subsequent versions of Sinclair BASIC: "The ZX80 integer BASIC, written by John Grant, was in Z80 assembly code pure and simple, though it did use the usual stack based techniques for interpreting expressions."

The lack of support for floating-point numbers overshadows Grant's achievement. He laid the path for things to come, introducing many unique features of Sinclair BASIC, such as the way it refuses to allow most syntax errors to be entered into the program, instead pointing out where the error is in the line before it is entered, making it much easier to learn and use than any other version of BASIC.

The kit was launched at a computer fair in the first week of February 1980, and while it was not a massive success by comparison with the ZX Spectrum, it turned Sinclair's fortunes around, eventually earning him a knighthood, and it sold well enough to persuade him to make a new computer: the ZX81.

Work on the ZX81 hardware had begun in September 1979, even before the launch of the ZX80, but it was the development of the uncommitted logic array, or ULA, which allowed the machine to go into production. The ULA, produced by Ferranti for Sinclair, reduced the chip count and brought the retail cost of the machine, in kit form, down to £49.95.

Again, Nine Tiles was called on to provide the new BASIC, but this time there was 8K to play with. Vickers, who had joined Nine Tiles in January 1980, wrote the BASIC more or less from scratch, only using some of the ZX80 code, making numerous improvements while managing to maintain backwards compatibility with the ZX80 hardware.

"As far as Clive was concerned, it wasn't a question of what the machine ought to be able to do, but more what could be crammed into the machine given the component budget he'd set his mind on," said Vickers in an interview on July 23, 1985. "The only firm brief for the '81 was that the '80's math package must be improved."

The ROM was almost complete by the end of autumn 1980, but support still had to be added for the ZX Printer. Somewhere between this time and the launch, a bug crept in which caused the square root of 0.25 to be 1.3591409. Nine Tiles quickly fixed the bug, but Sinclair was somewhat tardy in making this version available to people who had already bought the machine.

Despite this problem, the ZX81 was well received and became a massive success, spawning a series of clones, both illegal and licensed by Timex, which was manufacturing the UK models for Sinclair at its Dundee plant. Inspired by the public reaction to the ZX81, and annoyed at not winning the contract to design a computer for the British Broadcasting Corporation, Sinclair decided the market needed a budget colour computer.

The ZX80 and ZX81 hardware had been the primarily the work of one man, Jim Westwood, but he had been moved to the flat-screen television department, so the hardware design job on the ZX82, which became the ZX Spectrum, was given to Richard Altwasser - while Vickers at Nine Tiles was again asked to provide the BASIC.

What started out as an expansion of the ZX81 BASIC soon turned into a large 16K program. Sinclair wanted as few changes to the ZX81 code as possible, but Nine Tiles felt that software designed for a machine with 1K was inappropriate for a machine with 16K and that problems would occur later on. They were right.

"Certainly with the Spectrum we wanted to rewrite the code, but there wasn't the time and there definitely weren't the resources," said Grant in an interview on September 8, 1985. "At every point [in the development of the ZX range] Clive wanted the maximum new facilities for the minimum money."

After the best part of a year's work the BASIC was almost finished. While it was greatly enhanced, it was also depressingly slow, but more problems were to follow. The main problem was providing support for the planned peripherals, because no working prototypes were available to Vickers until near the end of

1981. But then, in February 1982, Nine Tiles began to have financial disagreements with Sinclair over royalties which it became apparent would not be forthcoming.

To make matters worse, Vickers and Altwasser both handed in their resignations in order to form their own company, Cantab, which went on to produce the Jupiter Ace, essentially a ZX80 with the Forth language taking the place of BASIC. The result of the delays these problems caused was that when Sinclair launched the machine, it did so with an incomplete ROM. Nine Tiles continued working on the ROM for three months after the launch in April 1982, but by then too many units had been sold and the program was never finished.

The original plan was to issue only a limited number of Spectrums with the incomplete ROM and provide an upgrade, much in the way the bug in the ZX81 ROM had been handled; except that by the time Sinclair got its act together, around 75,000 units had been sold and the plan became unworkable. This is the reason why the microdrive commands don't work in the standard ROM, and led to the development of the shadow ROM in the Interface 1 in order to handle peripherals which should have been supported directly by BASIC.

### How BASin differs from a real Spectrum

This section is intended for Adults who are familiar with the Sinclair Spectrum to read

BASin, although as functionally close to being a Sinclair Spectrum on a PC as possible, has several differences. Mostly these affect the way you enter programs into the emulated Spectrum. Firstly, the Spectrum ROM used has been patched extensively to allow new features and bug-fixes, though no changes have been made to the ROM that would affect how your programs would run on a real Spectrum. One example of this is the lack of K, L, C, E, and G cursors - because of the addition of PC keyboard support, these are no longer necessary, as is also the case with the Symbol Shift key. The main cursor in **INPUT** commands is now a white and blue flashing square, which hovers over the input rather than nestled between tokens. The editor does support the usual text processing operations such as cut, copy, paste and search/replace functions.

Editor and keyboard

Tokens are handled differently too. You no longer enter entire words using a single key-press - the editor is based on the 128k Spectrum, which means full typing support with the available use of every key on a modern keyboard. You are also free to use the mouse to make selections and get context sensitive help. For more information, consult the section devoted to Using the Editor.

As mentioned before, the keyboard does not function as a 48k Spectrum's keyboard would. This means that retrieving key states by reading the result of a BASIC IN instruction from port \$FE (254) no longer works. INKEY\$ is not affected, however. Machine code instructions that read the keyboard from a Z80 IN opcode will retrieve the correct keys - but the keyboard is significantly different from a PC. The CTRL key is designated to Symbol Shift, the Shift keys act as expected, but bear in mind that CTRL-P will produce a quote (") mark, as opposed to shift-2 on a UK keyboard.

Screen display

The main display output (the Spectrum "screen") sits to the right of the Editor in the main window. When a program is running, this window will be where any program output appears. It behaves exactly the same way as a real Spectrum's display. Because the editor is now in a separate area to the display, runtime errors no longer report to the bottom of the display - they can be redirected to an error dialog which also provides information on that error. This again helps prevent display corruption, but is a feature that can be turned off.

prompts have been enhanced: they use the same blue-and-white cursor as the editor, and they support cut and paste operations.

Loading and saving files

BASin uses text files with the .bas extension to store BASIC code. This code is in raw text format (windows notepad compatible) with escape codes to represent tokens that cannot be rendered in plain text. See the chapter on .BAS File Format for more details. These files can be used just like tape files on the original hardware - they can store runtime variable contents and auto-start when loaded. BASin for Beginners can load any of the formats that BASin supports, but can only save in .szx format. This is to ensure that any program saved by BASin can be run on a Sinclair Spectrum Vega.

See Loading and saving files for more information.

### **BASin and the 128k Spectrum**

Although BASin's emulation is based upon the more reliable and robust 48k Spectrum, it has extra functionality to provide the enhancements to BASIC that the 128k Spectrums brought.

The sound is enhanced over the simple beeper of the 48k, by using an AY-3-8912 sound chip, which enables the BASIC to produce three channel sound and white noise, with full envelope control.

The extra memory of the 128k Spectrum is available to BASIC via the *silicon disc* which is a "virtual disk" system providing 64kb of storage space which is almost instant to access. With careful programming and judicious use of the MERGE command, up to 91kb of BASIC program can be stored at any one time.

128k Commands

BASin will allow you to use the extra sound and silicon disc systems by "branching" the emulation out to the 128k +2 rom and AY-3-8912 sound chip as commands are intepreted. The commands given are...

```
PLAY
LOAD !
SAVE !
MERGE !
CAT
ERASE
```

For instruction on how to use these commands, see the following sections of this help file...

- Chapter 19 Beep and PLAY
- Chapter 20 File storage

The following keyword reference sections may also be useful...

- PLAY
- CAT
- ERASE
- LOAD
- SAVE
- MERGE

Limitations of the 128k BASIC - UDGs

The extra commands, PLAY and SPECTRUM had to be housed somewhere in the character set - all keywords in BASIC are stored in this manner. The designers decided that in order to accomodate the new commands, the last two User-Defined Graphics ("T" and "U") would be given over to them. This means that some programs, when run in 128k mode, will display these graphics characters incorrectly.

Because of this limitation, BASin will only replace these UDGs with their command-replacements if your program makes use of the silicon disc or PLAY command. This is worth bearing in mind if you would like to use them. As an example, the program "frogger" from the BASin examples, displays correctly when run in a 48k mode...



...and as you can see, all the graphics are accounted for and display well. However, placing a PLAY command anywhere in the game produces a quite drastic and undesireable effect...



...where all the instances of UDGs "T" and "U" have been replaced with "SPECTRUM" and "PLAY". This kind of thing was common in BASIC games designed on the 48k spectrum and run on the 128k without modification.

Limitations of the 128k BASIC - Printing

In order to provide a full 128kb of memory in the later Spectrum models, the designers utilised an ingenious method of "paging" - pages of RAM could be swapped into and out of the upper memory at will.

Also, to provide the extra commands for the silicon disc and AY chip, two ROMs were used, which could be swapped in a similar manner to the RAM pages in upper memory. Of course, when one ROM was paged in it would need to know that it had to page itself back out to allow the main ROM to resume control. This was not possible for the 48k ROM that lived behind the newer 128k ROM - it had to be an exact copy of the original 48k ROM for compatibility reasons, and this ROM had not been designed for paging systems.

This meant that an area of RAM had to be allocated which the 48k ROM, when it had finished the job it was called for, would jump back to. This area contains code that swaps the 128k ROM back in. Because the 128k Spectrum used an RS-232 port to communicate with more advanced printers, it was felt that the printer buffer utilised by the 48k ROM was no longer needed, and as such this area was reclaimed for the paging system and some more system variables which contained information about the silicon disk, amongst other things.

Because BASin allows the user to use the old 48k printer routines, then the use of the commands for it will try to read (and write) to the buffer region maintained by the paging system. The repercussions of this are that if any printing commands (LPRINT, LLIST) are used after a 128k command, at best you will see garbage on your printout, and at worst it may crash the emulated spectrum. If you use the printer commands with silicon disk commands, you may find that the silicon disc becomes corrupted, which may in turn also crash the emulation.

So if you're going to use 128k commands, then please avoid the use of the ZX Printer support.

BASin's 48k/128k program state indicator

To help remind you which computer your program is compatible with, BASin provides a small indicator at the bottom of the editor window. This is updated as you create or edit your program, and appears like this...



As you enter commands into BASin, this indicator will remain in "48k mode" until you enter a 128k command which utilises the enhanced sound (PLAY) or the silicon disc (LOAD !, SAVE !, MERGE !, CAT ! and ERASE !). When BASin detects these, the indicator will change...



...and when this happens you may also notice that any instances of the UDGs "T" and "U" will change to their small-character hexadecimal equivalents.

Removing 128k commands from your program will revert it back to 48k mode.

The Program Information Window will also display compatibility information based on your usage of 128k features.

**Note:** Snapshots saved by BASin will differ in behaviour if they are 128k hardware-based. See the section on Loading and saving snapshots for more information.

# **Using BASin**

BASin is easy to use - just start it and get typing! There are lessons online for you to try.

This section will give you an overview of what BASin can do, and what all the options and windows you can open actually do. BASin is quite a complex application, but is very user-friendly and helpful. You can call up help for any of the dialogs that BASin presents to you by using the F1 key (except in the Editor window, where this is used for fast context sensitive BASIC Keyword help).

Choose a section to get help for:

- Starting BASin
- System requirements
- Getting help
- The BASIC editor
- The display window
- Loading and saving files
- Running and debugging programs
- Debugging your programs
- Configuring BASin
- BASin's tools
- Sinclair BASIC reference

## **Starting BASin**

BASin has been made easy to use, and starting is no different. Start BASin by using the icon supplied (BASin.exe) in the Start Menu or in the folder BASin was installed into. You will firstly be greeted by the Editor window, which will remain unresponsive until the emulated Spectrum has been initialised.

This is a process whereby the original 48k ROM is loaded into an emulated memory model and executed from address zero. The ROM will perform a self-test, which is to fill all available RAM with the number two, and then read that back, whilst clearing the RAM back to zeros. Originally, this was a useful test which could tell the ROM how much memory is available back in the days of 16k and 48k Spectrums. It's not essential to BASin's operation, but is a nice historical effect. The net result is that the Display Window will display a white border around an animated black rectangle, with faint red vertical lines, before clearing to white with the message

### © 2015 Sinclair Research Ltd

at the bottom of the display. This indicates that all is well, and BASin is ready to start.

The Editor will then gain focus and pop to the front, with a blank white editing area, and a flashing blue/white cursor similar to the 128k Spectrum's Editor. The gutter at the left is for line numbers, and is not a bug in the Editor.

## **System requirements**

BASin is designed to perform well regardless of PC hardware specification. The emulation has been tested using a 50 MHz 486 DX/2, and (with sound disabled) runs very well on that specification. The editor is more resource-hungry, but the BASin tab allows certain features to be disabled if necessary.

Sound

DirectX 5 or higher must be installed in order for sounds to play. DirectX is not required for the display, but performance in this case will vary depending on your display driver software and computer usage by other programs.

Help file

HTML Help requires Microsoft Internet Explorer 5 or higher.

Spectrum ROM image

BASin requires unmodified Spectrum 48K and +2 ROM images, which are included in the installation package. Amstrad has graciously given permission for Spectrum ROM images to be freely distributed and included with emulators.

## **Getting help**

BASin has a very detailed help file, to allow you to easily find help on any topic. You can open the main help page using Ctrl-F1 or by using the Help Menu in the Editor.

The Help file also includes a complete BASIC programming course, adapted from the original Sinclair BASIC manual. This also includes specific help on any of the codewords found in Sinclair BASIC. Just hover your mouse over the keyword you want help with, and click your right mouse button to get a Codeword Context Menu for that codeword, which includes, amongst other options, help for that codeword. You can also use the Editor cursor to move to the codeword you want help with, and press F1.

To get help on an item in a tool, click the Help button in the window you want help with, or use the Help menu if available.

As a last resort, you can fnd contact details for the Author in the Contact chapter.

## The editor

The editor is the area where you enter and modify your BASIC program.

BASin's editor is where you enter your program.

Program lines are automatically arranged in numerical order, no matter where they are entered.

Keyboard and mouse operations

BASin supports many standard navigation keys.

Like any other Notepad program, you can use the mouse and/or shift keys to make selections and cut/copy them to the windows clipboard. They will be stored in .bas format which can be pasted into any text editor. Pasting is a little different. A single line of text pasted in will be sent directly to the editor, but more than one line will be sent to the Add Code Window - where you can view the lines and change them as you wish. Multiple line pastes must each have a line number, and contain no syntax errors. If an error occurs when pasting the lines in, the offending line will be ignored and will not appear in your program. You can see the status of any errors in pasting by viewing the Log Window. This facility allows you to copy a BASIC program from say, a web page or text document, and paste it directly into BASin.

Right-clicking in the editor produces a context menu with commands relating to the item that was clicked.

How your program is formatted

The darker area at the left-hand side is the **gutter**, where line numbers are displayed. To aid debugging and make things easier to read, program lines that contain more than one statement are split on the : (colon) character, so that each

statement begins on a new line of the display.

How BASin handles program lines

Program lines, as mentioned earlier, are stored in the editor in order, sorted by their line number. This makes the program easier to follow for the programmer.

BASin has a number of features to help you with your programming:

#### • The Coding Teacher

The coding teacher is a bar that appears at the bottom of the editor window, which tracks your code as you type and offers hints as to what the current codeword actually does, and it's syntax.



If you type an error, it will colour the relevant part of the syntax list red. The syntax that is currently expected is coloured blue, and any return types of codewords is in green. Error messages will appear at the bottom of the helper bar which explains what has gone wrong.

For more information, there is a chapter dedicated to the coding teacher.

#### • The Character Ruler

This option is available in the View menu, rather than the Options dialog.

The character ruler helps you to make long strings for display on-screen. Due to the nature of the **PRINT** keyword, a string longer than 32 characters will "wrap" round to the beginning of the next line down. This can be useful for displaying many lines of text with one **PRINT**. The character ruler will show you at a glance how long your string is, and a red marker will appear at any point where the 32 character boundary is reached. This activity will

be activated whenever the cursor enters a string area in the BASIC text.

More information is available in the Character Ruler chapter.

#### • Line Overwrite Protection

BASin will not allow you to enter a line that contains a syntax error, and when you make one the cursor will move to the error and if you have sounds enabled, will emit a low pitched beep to alert you.

```
10 PRINT "Welcome!"
10 PRINT "Hello!"
```

This behaviour is also used by the line-overwrite protection system which you can configure in the Options window, on the BASin tab. This, when enabled, will alert you when you try to enter a line for which a line already exists with the same number. The cursor will turn green, and the same low beep can be heard. In this instance, however, you do not have to correct the line number - pressing Return a second time will accept the new line and replace the old one.

#### • Auto line numbering

BASin has a shortcut (the "." key) which will create a line number for you. Pressing the "." key at the start of a new line will choose a line number roughly halfway between the previous line number (of the line directly above the current cursor position - Line 0 is assumed if you're at the start of the program) and the next line number if it exists. If no next line is present, then 10 is added to the previous line number. The "." will not be displayed - instead, your new line number will be typed out. This behaviour can be enabled or disabled in the Options window.

#### • Auto-Bracket completion

Finally, a word about typing expressions of the form

(a+(b/c))\*3

Which make use of brackets - BASin has an option which can make

counting many brackets easier - BASin can either insert a closing bracket as you type, ahead of the cursor, or can complete any open brackets when you press enter to finish a line.

#### • Syntax Highlighting

The text of your program can optionally colour in the text according to the type of words displayed.

1	REM This is a FloodFill
10	CIRCLE 128,88,40:
	CIRCLE 128,88,30
20	LET x=100:
	LET y=100
30	GO SUB 1000:
	STOP
1000	PLOT X, y

This is called "syntax highlighting", and can make a program a lot easier to read and follow. Keywords, user defined functions, comments, numbers, symbols, strings, variables (BASin will colour variable names that exist in VARS memory space differently to those that do not, which can quickly where show typing errors they may occur) and up the background/foreground colours can all be altered in appearance, also to be set to bold or italics in style. Be aware that this requires quite a lot more effort to display by your CPU, so may slow the editor down on older PCs.

#### • Predictive typing

This option will display any keyword or variable name in dark grey to the right of your cursor as you type.



If you continue to type, the search for the correct word will be narrowed down - when you see the correct word displayed, hit the right cursor key, or the "." key to accept it. You can, of course, continue typing over the top. If you do not accept a word and press return, any predicted text will be removed.

#### • Source Markers

Source markers are a method of moving quickly around a large program. By inserting a source marker, you associate that marker with a particular line or statement. By recalling it later, you can jump instantly to that line. This makes navigation between important sections of code much easier. You can access source markers from the Search menu.

#### • Mode indicator

The mode indicator is a black strip that appears at the bottom of the editing area. It shows which type of spectrum your program is compatible with, based on commands that your program uses. It shows "48 BASIC" by default, until you use a 128k command, at which point it will show "128 BASIC". The Mode Indicator is visible at all times, and cannot be turned off.

Editor features

- The toolbar.
- The character ruler.
- The coding teacher.
- The status bar.
- Variable ToolTips.
- Source Markers
- Expression Evaluator.

### The status bar

The status bar is located at the bottom of the editor, , and shows some information that can be useful when programming in BASin. It can be enabled or disabled from the View Menu.

40 Kb available to BASIC (0 Bytes Used)	Insert
---	--------

Syntax state indicator

The white-on-black letter at the left-hand side indicates the type of syntax element that BASin expects you to type next. (This is roughly equivalent to the old-style edit cursor on a 48K Spectrum.)

M	A keyword is required (e.g. at the start of a line).
	Characters will be entered in lower case.
0	Characters will be entered in upper case.
6	Characters will be entered in graphics mode.

Hint area

The hint area usually displays the size of your program and the amount of free memory in kilobytes. A Spectrum has about 38 kilobytes of memory for BASIC programs.

When a menu is open, the hint area displays a short description of the selected menu command.

Keyboard state indicator

The area on the right displays Insert or Overwrite, depending on whether

overtype mode is enabled.

## The keyboard

BASin makes full use of the PC keyboard.

The editor uses a standard set of keys for editing, as well as some special ones that are unique to Spectrum BASIC.

<b>← → ☆ ↓</b>	Moves left, right, up, or down by one character.
Shift + ← → ↑ ↓	Extends the current selection.
Home, End	Jumps to the beginning or end of the current line.
Ctrl + Home, End	Jumps to the beginning or end of the entire program.
Page Up, Page Down	Moves up or down by one screenful of text.
Insert	Toggles overtype mode, where newly typed characters replace old ones.
Delete	Deletes the character to the right of the cursor.
Backspace	Deletes the character to the left of the cursor.
Tab	Inserts three spaces. (Actual tab characters are not supported.)
¬ or €	Inserts the © copyright symbol.
^	Inserts the $\uparrow$ exponentiation (raise to a power) maths symbol.

When you attempt to move the cursor away from a modified line, BASin behaves as though the line had been entered, checking it for mistakes before storing it in the program.

#### Graphics characters

Spectrum BASIC programs can contain two kinds of graphics characters: builtin chequered graphics and user-defined graphics (UDGs). These characters are described in Chapter 14 of the manual.

In BASin, you can toggle graphics mode using **Num Lock** or **Scroll Lock**, or (under Windows NT and XP) enable it temporarily by holding down **Alt Gr**. The key to be used can be configured on the BASin tab.

Control characters

Spectrum BASIC programs can contain special characters, called control characters, that affect the display of the listing.

You can insert control characters using either the Token Table window or the function keys F1 to F8.

In BASin, the appearance of "control characters" depends on whether syntax highlighting is enabled; and the flash attribute has no visible effect in a program line.

**Note:** Control codes will appear in the program as small hex numbers, two per character square, when Syntax Highlighting is enabled - when this is disabled, then BASin will show the text with any colour modifications as it would appear on a real spectrum, with the exception of the FLASH attribute.

Keystroke	Graphics	Non-graphics
F1	Blue	
F2	Red	
F3	Magenta	Inverse Video on
F4	Green	Inverse Video off

F5	Cyan	Bright on
F6	Yellow	Bright off
F7	White	Flash on
F8	Black	Flash off

In graphics mode, a function key sets the ink colour, and a function key with **Shift** sets the paper (background) colour.

## The toolbar

The toolbar is located at the top of the editor.

It provides quick access to some of the more commonly used menu commands.



#### Buttons

From the File menu:

	<b>New</b> - This button will clear the current program and variables from memory, allowing you to start afresh. You will be prompted to save any changes you may have made to the current program.
<b>6</b> 3	<b>LOAD</b> "" - Brings up the Load Program dialog. The same as using the FileLoad "" options from the main menu. You can load either .bas files, or Spectrum Snapshots.
2	<b>Save</b> - Saves the current program in memory. If this program is new, and has not been saved before ("Untitled Project") then the save program dialog will be displayed prompting you for a filename. This saves either .bas files or Spectrum Snapshots.

#### From the **Run menu**:

▶ 11	<b>Run</b> / <b>Pause</b> - If the current program is not running, then this starts the program from the beginning, as if the user had typed " <b>RUN</b> " in the editor as a Direct Command. If the program is already running then execution will be suspended when the current statement has been executed. From there you can use the single step and other debugger functions.
0	<b>GO TO cursor</b> - This will start program execution from the line and statement that the editor cursor is currently positioned at. Similar to issuing a GO TO command from the editor.
•	<b>System BREAK</b> / <b>Continue</b> - This will either stop the current program execution by sending a BREAK message to the emulation, or <b>CONTINUE</b> from where it left off. See the <b>CONTINUE</b> keyword help in the <b>BASIC</b> manual for more on how <b>CONTINUE</b> operates.

9 <mark>9</mark> 9	<b>Single-step statement</b> - When program execution is suspended, you can step through the program one statement at a time, pausing after each so you can inspect the program flow and variables. See the Single Step tool description for more information.
4	<b>Step over statement</b> - Will, when the program is suspended, run through the current statement until the next statement in the program is encountered. This allows you to step past subroutines that you don't want to have to step through.
R	<b>Run to cursor</b> - Will start execution and not stop until the program reaches the statement that the editor cursor is resting on. The statement will not be executed - the program will be suspended before it starts.
8	<b>Add Breakpoint at cursor</b> - Will toggle a breakpoint at the current statement which the editor cursor lies upon. It's the same effect as double clicking a line in the editor. See the Breakpoints section for more on breakpoints. This will open the Breakpoint Properties window so you can edit the breakpoint's conditions.
Ø	<b>Add Watch</b> - Opens the Watch Properties window to allow you to add a Watch to the current Watchlist. See the entry under Watches for more information.

### From the View menu:

### The character ruler

The character ruler is located at the bottom of the editor. It helps making long strings for display on-screen. Due to how the PRINT codeword works, a string longer than 32 characters will "wrap" round to the beginning of the next line down. This can be useful for displaying many lines of text on one command. The character ruler will show you at a glance how long your string is, and a red marker will appear at any point where the 32 character boundary is reached. This happen whenever the cursor enters a quote delimited string area in the BASIC text.

To show or hide the character ruler, choose **Character Ruler** on the View menu.

Edit cursor position

Each mark on the ruler represents one character. Every eighth mark has a number that indicates the character count from the start of the program line. These numbers increase as you scroll to the right.

The small black arrow ( • ) shows the location of the edit cursor.

A grey vertical bar shows the position of the mouse pointer.

String lengths

The character ruler also indicates the lengths of strings. (A string is a piece of program text enclosed in quotation marks, such as "Welcome".)

While the edit cursor is inside a string, the character ruler displays the length with a green bar, with a red arrow after every 32nd character. The Spectrum screen display is 32 characters wide, so this shows where a line of text will wrap.

```
PRINT "Sele<mark>c</mark>t an option:"
```

See also the section on the editor context menu for information on wrapping long strings around the screen with word-wrap and other effects.

# The coding teacher

The coding teacher is located at the bottom of the editor.



To show or hide the coding teacher, choose **Debug Windows** > **Coding teacher** on the View menu.

If you don't want the coding teacher to appear automatically when BASin starts, clear the **Show coding teacher on startup** check box on the BASin tab.

The coding teacher displays a continuously updated reminder of the correct syntax for the keyword you are typing. It can help you identify and fix any problems with a program line before you enter it.

The top line of the coding teacher is a colour-coded template that illustrates the syntax of the statement being edited.

- The **return type** from a function is displayed in **dark green**.
- The **expected item** the syntax element that comes next is **dark blue**.
- Any **syntax error** or missing item in the current statement is **dark red**.

The bottom line of the coding teacher provides a more detailed explanation of any errors.
# Variable ToolTips

Hovering over a variable in the editor produces a floating ToolTip that shows its name, type, and contents.



**Note:** Variable ToolTips only appear for variables that are already present in memory. This means that a variable must have been made (using a **DIM** or **LET** statement) before it will produce a ToolTip.

# **Source Markers**

Source Markers are available from the Search menu.

	170	LET	h=.5:
		LET	rd=PI/180
	180	LET	c = k * COS (w * cd):
	400	나타	s=K*SIN (w*rd)
	190	나타	dx =2:
		나타	af = a / 93
	200	DIH	h (256)
	210	FOR	L=1 TO 256
2	220	LET	h(l)=-1000
	230	NEXT	ГІ
	240	CLS	

Source Markers are a method of managing large BASIC programs easily. You can use these to "mark" important sections of code, and once marked, can jump back to them from anywhere in your program. Lines and statements that have been marked have a small green number next to them in the gutter to the left of the line, as in the illustration above.

Setting a source marker

To set a source marker, use the Search Menu. You can also move the editor cursor to the line or statement you want to mark, and then press CTRL-SHIFT and a number from 0 to 9.

Jumping to a source marker

You can retrieve a source marker that has previously been set by using the Search menu. You can also use CTRL and the number of the source marker you wish to jump to.

**Note:** you cannot jump to a source marker that has not been previously been defined, for obvious reasons.

Clearing source markers

You can clear all the source markers by using the Clear all option on the Search menu. Once cleared, none of the source markers will be available.

**Note:** If you clear your program using **NEW** or by loading a new program, the source markers will be automatically cleared.

# **Expression Evaluator window**

The Expression Evaluator window is available from the View menu.

Evaluate Expression	
Expression :	
Davilla	*
Result :	
Evaluate Add as Watch	Close

This window, accessed from the View Menu, provides a simple calculator for Sinclair BASIC expressions. At the top of the window is an edit field where you can type in your expression, which also incorporates a drop-down list consisting of the "history" of all expressions you have typed. The expression can contain any keywords or functions accessible to Sinclair BASIC, including any variables you have declared either programmatically at runtime, or by Direct Command.

Upon pressing Return, or by clicking the "Evaluate" button, the expression will be parsed for errors and executed. This is performed by starting a new instance of Spectrum Emulation, and allowing the ROM to process the expression as though it had the keyword **PRINT** inserted before it. The resulting text which would be displayed on-screen is taken by BASin and sent to the larger "Result" field below the edit field.

**Note:** expressions by their nature can contain functions such as the USR function which can execute a section of memory as machine code. Although it's a valid function (and returns the value in BC as the code performs it's final RET) it can get stuck in infinite loops, or take an age to complete. This will mean that the Evaluate Expression Window will not respond to requests for more expressions, either at all or for a long time. To this end, you can press the "Evaluate" button again to halt the current expression. Any result returned will be displayed. Closing the Evaluate Expression Window will also have this effect.

Also, related to this, the Evaluation will use the same "engine" to produce the

result as the Watches (and other evaluated expressions, such as Find Window parameters) and as such any expressions taking a long time to evaluate will "stall" other expressions waiting to execute. However, none of them will affect the progress of the main system emulation.

There is also the option, via the "Add as Watch" button, to add this expression as an Expression Watch to the Watches Window.

# **Program Information**

The Program Information window is available from the View menu.



The Program Information window is provided simply to give an "overview" of your program - how many lines and statements you have used and how many variables you have declared.

It also gives information as to how much memory has been used in total, how much of that is program code, and how much has been used by the variables. It also displays the amount of free memory remaining to BASIC. This figure is calculated using the RAMTOP system variable and so represents how much you can use currently, and may not be taking the full memory of the Spectrum into account.

The final line informs you which of the two flavours of Spectrum your program is compatible with - 128k or 48k.

If you use the extended 128k commands then the pogram will be 128k only.

If you use the UDGs "T" and "U" inside strings, then it will be 48k only (as these UDGs were not available on the 128k).

If you use neither then the program is deemed compatible with both, with one exception - if you have files currently residing on the silicon disc, and the silicon disc has been used then the window will warn you that the program *may* not be compatible with the 48k Spectrum should you decide to utilise these files in the

future. However, as silicon disk contents are not saved to inside .bas files, and also not to tape files, then compatibility might not be an issue.

When saving snapshots the contents of the silicon disc will be stored, and as such if the silicon disc is in use you will not be able to save 48k snapshots.

# The 48k/128k mode indicator

To help remind you which computer your program is compatible with, BASin provides a small indicator at the bottom of the editor window. This is updated as you create or edit your program, and appears like this...



As you enter commands into BASin, this indicator will remain in "48k mode" until you enter a 128k command which utilises the enhanced sound (PLAY) or the silicon disc (LOAD !, SAVE !, MERGE !, CAT ! and ERASE !). When BASin detects these, the indicator will change...



...and when this happens you may also notice that any instances of the UDGs "T" and "U" will change to their small-character hexadecimal equivalents.

Removing 128k commands from your program will revert it back to 48k mode.

# **Find window**

The Find window is available from the Search menu.

Find				
Text to find:	×			
Direction	Origin			
<ul> <li>Forward</li> </ul>	<ul> <li>Start of BASIC</li> </ul>			
O Backward	<ul> <li>Current Program Cursor</li> </ul>			
Options				
Case Sensitive				
Expression				
	Find Cancel			

This window allows you to search for text within your program. It works in a similar way to other text editor's Find... dialogs, but has some useful behaviours pertinent to Sinclair BASIC.

You can type the text you wish to search for into the edit field at the top of the dialog, and set options for the search:

#### • Direction

This sets which way BASin will search for your text - either forward towards the end of your program, or backwards towards the beginning. Searching backwards only makes sense when you search from the current cursor position as the origin.

• Origin

This sets where in your program BASin will begin the search - either from the current Editor cursor position, or from the start of the program.

Because BASin has the ability to evaluate Sinclair BASIC expressions independently from the main emulation, the text can be flagged as an expression rather than just a string of characters to be searched for. If you set the

"Expression" checkbox, then BASin will evaluate the search text as an expression, and use the result as the text to search for - for example, if A\$ is set to "Hello" then entering "A\$" as the search text and then checking the "Expression" box will set BASin to search your program for the string "Hello".

Lastly, you can use the "**Case Sensitive**" checkbox to search for an exact match of your text, including capital or lowercase letters, or to search regardless of letter case.

# **Replace window**

The Replace window is available from the Search menu.

Replace				
Text to find:	~			
Replace With:	~			
Direction Forward Backward	Origin Start of BASIC Current Program Cursor			
Options Case Sensitive Find Expression Replace Expression Cancel Cancel				

The Replace Dialog is functionally very similar to the Find Text Dialog, but allows you to replace the found text with new text of your choosing. To this end, there is the usual Find... options, such as the edit field for your search text, but an additional edit field with which you can specify the text you want to replace the found text with.

As with the Find Text Dialog (the chapter for which gives more information about searching for text), you can specify that the text to search for is an expression with the "Find Expression" checkbox, and also that the text to replace it with is also an expression with the "Replace Expression" checkbox.

The Direction and Origin options are the same as for the Find Dialog, as is the "Case Sensitive" option.

Lastly, there are the two buttons at the bottom of the window:

#### • Replace All

This button will, regardless of direction and origin, replace all occurrences of the search text with the replace text in your program.

### • Replace

Will perform one replace operation at a time, subsequent presses of this button will repeat the process with the next occurrence of your search text.

# **The Editor Menu strip**

The Editor menu strip is the main method of getting at BASin's more advanced features. Below is a list of options on each of the menu items, and a brief explanation of each can be found by clicking the corresponding link.

File Menu New Load '''' Re-Load Save Save As... Print Exit

Edit Menu Undo Redo Cut Copy Paste Delete Copy Listing

View Menu Status Bar Toolbar Syntax Helper Character Ruler Program Information Command History Last Error Debug Windows Variables Breakpoints Watches Log Window Expression Evaluator

Search Menu Find... Replace... Find Next Replace Next Source Markers Set Marker Get Marker Clear All Go to Line Number... Go to Error

Run Menu Run/Pause/Resume Break/Continue Go to Cursor Enable Profiling Force Break Trace Execution Single Step Statement Step Over Statement Run To Cursor Toggle Breakpoint Add Breakpoint... Add Watch...

Tools Menu BASin Options... Token Table... UDG Editor... Renumber...

Help Menu Contents Command Help Sinclair BASIC Manual Error Help About

# File menu

The File menu provides commands relating to the current BASIC program and the associated file on disk.

New

Clears the current program and variables from the editor, and also clears them from the emulated Spectrum's memory. You will be prompted to save any changes you may have made to your program since your last save, but this applies only to the program itself and not any variables that may have been altered.

LOAD ""

Opens a saved program or snapshot file.

Re-LOAD

Opens a recently used program from a list of up to 10 programs, or (if you choose **Previous Session**) restores the program and emulated Spectrum state from your last BASin session. To restore your session automatically whenever BASin starts, select **Restore previous session** on the BASin tab.

Save

Saves changes to the program, in the format used to load it. If no format was specified (if the program was created from scratch) then the default .bas format will be used.

Save As

Prompts for a filename and file type, and saves the program to disk - either as a .bas file, or as a snapshot file. The default method is to save as a .bas file, other types of file are specified by the extension of the filename supplied by you. Saving "myprog.sna" will instruct BASin to use the .sna format, for example

Print

Opens the **Print Preview window**, ready to send the program listing to a printer.

Exit

Quits BASin, prompting to save changes if necessary.

# Edit menu

The Edit menu provides commands for manipulating text in the editor.

BASin uses the **. bas** file format to store program text on the Clipboard.

Undo

Reverses changes you have made to your program, one at a time.

Redo

Repeats the changes that you have undone. This command is only available immediately after an undo.

Cut

Removes the selected text and places it on the Clipboard. The clipped text is stored as .bas format, so any control characters in the text, or characters from the Spectrum ASCII set will be converted to escape characters.

You can then Paste this text into say, a notepad or an email.

Сору

Copies the selected text to the Clipboard. As with the cut option, text is stored in the .bas format.

Paste

Inserts any text found on the Windows system clipboard into the program at the current cursor position. The text will be processed for .bas format escape codes, and then will be inserted using one of two methods:

If the text contains no Return (CHR\$ 13) characters, then it will be inserted as one line.

If the text contains more than one line, then the Add Code Window will be shown, with the clipboard text in the edit area as plain text. You can change this text if you like, and it will then be checked for mistakes before being added line by line into your program. If any of the lines will overwrite any current program lines, then (assuming that you have enabled Line Overwrite Protection in the relevant Options Window page) BASin will warn you with the option to stop, ignore the line, or replace the old line in the program.

#### Delete

Removes the selected text. Does not store a copy of the text on the Windows clipboard. You will lose the text forever - use with care!

#### Copy Listing

Copies the entire program listing to the Clipboard. Unlike a normal save in .bas format, this does not include any variables in memory.

### **View menu**

The View menu provides commands relating to information about your program and to show the various debugging tools.

Toolbar

Shows or hides the toolbar which contains shortcut buttons to the more common of BASin's editor operations.

Status Bar

Shows or hides the status bar at the bottom of the editor window.

Syntax Helper

Shows or hides the code teacher. This is a useful tool which evaluates the "correctness" of your program line as you type - errors are highlighted in red, enabling you to spot errors before you enter a program line.

Character Ruler

Shows or hides the character ruler.

**Program Information** 

This window will display information about your program - how large it is in memory, how many variables you currently have declared and their usage of available memory. Also how much memory remains available to you for programming. Bear in mind that the system variable RAMTOP is used to calculate this figure.

**Command History** 

Shows or hides the Command History window - contains a list of all direct commands which have been entered into BASin for this session.

Last Error

Displays the most recent BASIC error message and its description, in the error dialog.

Debug Windows

Opens a submenu that allows you to show or hide the debug windows. See the section entitled "debugging your programs" for a more in-depth discussion of this powerful aspect of BASin.

- Variables: Shows or hides the Variables window, which lists all variables currently declared in memory. Also provides the means to alter a variable's contents whilst debugging a program, and to set a watch a variable for changes.
- **Breakpoints**: Shows or hides the **Breakpoints** window, where you can manage any breakpoints you might have set up.
- Watches: Shows or hides the Watch List, which like the **Breapoints** option above allows you to manage any watches you have set.
- Log Window: Shows or hides the Log window. Certain operations can produce log messages on completion, and one of the REM Directives can produce user-configurable log entres. You can view them in the **log window**.

Expression Evaluator

Shows or hides the Expression Evaluator window, where you can enter a BASIC expression and view its result.

# Search menu

The Search menu provides tools for finding text in your program.

Find

Displays the Find window, where you can search the current program for a word or phrase.

Find Next

Finds the next single occurrence of the word or phrase after the cursor position.

Replace

Displays the Replace window, where you can search the current program for a word or phrase and replace it with another word or phrase.

**Replace** Next

Replaces the next single occurrence of the word or phrase after the cursor position.

Source Markers

This menu opens a submenu which you can use to manipulate Source Markers.

Source Markers are like bookmarks in your program, and can be used to quickly move around a very big program.

- **GetMarker** opens a submenu containing ten options to set a marker at the current cursor position. A small green number corresponding to the source maker's number will appear in the gutter to the left of the statement.
- **SetMarker** opens a submenu with a list of the ten markers available. Any that have been set will be enabled. Any source markers that have not been selected will not be available. Clicking one of these will send the editor

cursor to the statement or line associated with it.

• **Clear All** - this option will clear any source markers you have set.

#### Go to Line Number

Opens a small requester which allows you to type in a line number to jump to in the Editor. This line number does not have to be a number - you could specify an expression (which must end up as a number, like "2+4" or "a+6") which can be specified using any Sinclair BASIC codewords, such as functions or variable names. For example, By setting variable "A" to 10, then typing "A" into the edit box, BASin will evaluate the result as 10, and move the cursor to line 10.

#### Go to Error

Moves to the program line and statement that caused the last error.

### Run menu

This menu is the main control for running your program, and most of the debugging options are in here. Most have keyboard shortcuts. More information on debugging can be found in the chapter headed Debugging your programs.

Run / Pause / Resume

This item will start your program if it is not already running, and has not been paused. Selecting this option again will put your program into pause, or debug mode. A further click on this item will start your program again. Stopping your program this way is **not** the same as using the **BREAK** key (the Escape key on your keyboard). Whilst the program is stopped, it cannot be edited but various debugging methods can be carried out.

**Note:** When using this item whilst the program is neither running nor paused, you should be aware that the program will be started using a **RUN** direct command, which will perform a **CLEAR** codeword as part of the startup. This will remove any previously declared variables from memory.

System BREAK / Continue

Sends a BREAK keypress to the running program, or continues running with a **CONTINUE** statement.

**Note:** You can only break into a program between statements. This command has no effect until the current statement finishes, even if it is a statement which might not end soon (such as **INPUT** or **PAUSE**).

Go to cursor

Continues the program from the location of the edit cursor. This has the same effect as typing

```
GO TO [line number]
```

and will not clear your currently declared variables.

#### Force BREAK

Aborts the running program and returns to the editor as it was before the last direct command was started. Unlike System BREAK, this command has immediate effect and works even when the running program cannot see the Escape key. This is because BASin stores the state of your program just before any **direct command** is executed, and this menu item will restore that state when clicked.

Trace Execution

Enables or disables tracing, which illustrates program flow by highlighting each statement in your program as it is executed. This can take a considerable amount of CPU time away from your program, which may run significantly slower as a result.

Single Step Statement

When the program is paused, or has triggered a **Breakpoint**, this item can be used to move through the program one statement at a time - the program will resume execution, and when the next statement has been executed, will pause again. Single-stepping is one of the most powerful debugging tools.

Step Over Statement

Executes the next statement in the program, waits for the program to reach the statement after that, and then pauses. In the case of a **GO SUB** statement, for example, whereas **Single Step Statement** would jump to the first line of the subroutine, this command will execute the entire subroutine and move to the next statement on the calling line.

Run to Cursor

Runs the program up to the location of the edit cursor, and then pauses.

Toggle breakpoint

Toggles a breakpoint on the line that the cursor is currently resting upon. This will be a simple breakpoint which will only trigger when program execution reaches that line.

Add Breakpoint

Opens the Add Breakpoint window, where you can specify confitions and other properties for a custom breakpoint.

Add Watch

Opens the Add Watch window.

# **Tools menu**

The Tools menu provides built-in utilities and access to program settings.

For more information, see **BASin's tools**.

**BASin Options** 

Opens the Options window, where you can configure the behaviour and appearance of BASin.

Token Table

Opens the Token Table window, which gives access to a chart of the ASCII set used by Sinclair BASIC, including a visual selection of colour control codes, UDGs, and keywords.

UDG Editor

Opens the UDG Editor window, which provides an easy to use way to make User Defined Graphics in your program.

Renumber

Opens the Renumber window, where you can change the line numbering of the current program.

# Help menu

The Help menu provides help and information about BASin and Spectrum BASIC.

Contents

Displays this help file.

Command Help

Displays help on the BASIC keyword under the cursor. Pressing F1 in the editor has the same effect.

Sinclair BASIC Manual

Displays the Spectrum BASIC manual, as corrected and modified for BASin.

Error Help

Displays the Error Help window.

About

Displays program information, version number, copyright, and credits.

# The context menu

The context menu appears when you right-click the program listing in the editor.

When using the editor, it can be useful to get help with a codeword or things with a variable. BASin allows you to click your right mouse button within the Editor which will bring up the a menu. Depending upon what you were pointing at when you clicked, the menu will give you different options. Bear in mind that if you bring up the menu for a variable, it must have been given a value by the LET codeword before it will be recognised by BASin.

#### Codewords

These options are available when you select a codeword (such as **PRINT**).

#### Token Help

Opens the Spectrum BASIC manual at the entry for that codeword.

#### Tokenise

The Spectrum character set includes a single character, called a token, for each codeword. It is possible (but never necessary) to include a token in a string. This command converts the selected keyword to its token, indicated in the program listing by a very tiny value, that can be made of letters or numbers.

**Note:** If you need to store a token in a string, it is usually better to use the CHR\$ codeword.

You can also tokenise entire strings by using the "String Operation" menu when right-clicking on them.

#### **De-Tokenise**

Converts a token, represented by its tiny-text value, back to the original codeword. When called from the "String Operation" Menu, when a string literal has been right-clicked, you can un-tokenise an entire string.

#### Find Line

Any codeword which requires a line number as it's parameter such as RUN, GO TO, GO SUB etc, where the line number is present will have this option available. Using this, you can jump to the line number indicated by the parameter. This also applies to line numbers that are calculated, i.e. GO TO 5+5 will move the editor cursor to line 10. As this is evaluated, you can use any Sinclair BASIC expression to specify the line number. This also applies to FOR variables, which will allow you to jump to the loop point (the statement directly after the FOR statement). This will not find the relevant NEXT statement line, as you can use more than one NEXT for a FOR statement.

#### Variables

These commands are available when you select a variable.

**Note:** The context menu only recognises variables that are already present in memory. This means that a variable must have been declared (using a **DIM** or **LET** statement) before it will show up in the context menu.

#### Edit Variable

Opens the Variables window for the selected variable, so that you can change its value.

#### String Wordwrapping

These options are available from the "**Wordwrap String**" when you activate the menu on a long string - that is, one that is longer than 32 characters.

Wordwrapping is useful when a string has to be printed to the screen, but is too long to fit into the standard 32 character wide display. By default, the text would flow over the edge of the screen, cutting off halfway through a word if necessary:

10 PRINT "This is a very long string and will w

Will display as

# This is a very long string and w ill wrap

Clearly, this looks unacceptable. So instead, there are word wrapping options available for these strings on the Context-Menu.

**Split at 32 Chars** - This option will split your string into many strings, seperated by the ' character (the line-feed **PRINT** item). The split will occur at the first space charecter *before* the point where the split would occur normally.

Hence, our long string now looks like this in code:

### 10 PRINT "this is a very long string and"'"will

**Insert Spaces** - This option will, instead of splitting the string, insert space characters before any words that will be split to force them over the 32 character boundary. Having applied this method, our line of code now looks like this:

### 10 PRINT "this is a very long string and will v

Note the extra space between "and" and "will". This makes sure that the word "will" begins on a new line. It does however make your lines even longer.

Standard commands

These commands are always available on the context menu.

#### Cut

Removes the selected text (if any) and places a copy on the windows clipboard. The clipped text is converted to .bas format, so any control characters in the text, or tokens from the Spectrum ASCII set will be converted to escape characters.

#### Сору

Copies the selected text (if any) to the Windows system clipboard in text format.

As with the Cut operation, the text will be converted to .bas format before being placed on the clipboard.

### Paste

Inserts any text found on the Windows system clipboard into the program at the current cursor position. The text will be processed for .bas format escape codes, and then will be inserted using one of two methods:

If the text contains no Return (CHR\$ 13) characters, then it will be inserted as one line.

If the text contains more than one line, then the Add Code Window will be shown, with the clipboard text in the edit field as plain text. You can edit this text to your liking, and it will then be tested for syntax validity before being inserted line by line into your program. If any of the lines will overwrite any current program lines, then (assuming that you have enabled Line Overwrite Protection in the relevant Options Window page) BASin will warn you with the option to abort the operation, ignore the line, or replace the old line in the program.

#### Debug

This submenu provides access to common debugging features.

- **Toggle Breakpoint** creates or removes a breakpoint at the editor cursor position.
- **Run to Line** runs the program and pauses at the statement indicated by the edit cursor.
- **Go to Line** runs the program from the line and statement at the edit cursor position.
- Watch Variable If the context menu is called for a

variable, you can Watch the variable as program execution runs. The watch will be added by opening the Watch Properties window, which will be pre-filled with data about the variable.

# Loading and saving files

This section is more for adults and those with experience of files on a computer.

Traditionally, the Spectrum used cassette tapes for loading and saving programs and data.

With BASin, the commands LOAD, SAVE, MERGE, and VERIFY refer to files on your actual computer's hard disk. Full Windows paths are acceptable (such as C:\programs\fruit.bas), but be aware that programs that load or save with long paths will not work correctly on an original Spectrum. When no path is specified in a filename, the most recent path is used.

You can specify an empty filename if you wish to locate the file manually instead of typing its full path. For example, LOAD "" displays a dialogue box where you can select a file to open.

Note that although BASin for Beginners can LOAD any of the following formats, you can only save in the .szx format (a snapshot format) which will allow you to run your program on a real Spectrum Vega.

Programs are saved and loaded in .bas format or Spectrum Emulation Snapshot format. .bas format is a text representation of a BASIC program. Escape characters can be used to indicate embedded colour commands and UDG or graphics characters. In all load and save dialogs relating to programs, the .bas extension is assumed, and will be added if the user omits it in the filename. In all respects the emulated Spectrum receives the data from a .bas or snapshot file as if it were a tape block - program files are encoded into tokenised BASIC before being passed to the ROM for loading into the emulated memory space.

File formats

See the following topics for a detailed description of the file formats that BASin supports:

- The . bas format, which is BASin's preferred format for programs.
- The **. bsc** format, for code blocks.
- The **. bsd** format, for array variable data.
- The **. SCr** format, for Spectrum screen images.
- Snapshot files, which describe the complete state of the emulated Spectrum.
- Tape files, which contain sequential data blocks like a cassette tape.

### The .bas file format

A .bas (BASin program) file contains a plain-text representation of a Spectrum BASIC program and variables. This is the preferred file format for programs developed in BASin.

The **.bas** format is an extension of the format used by **zmakebas**, a tool written by Russell Marks that converts plain-text BASIC programs into **.tap** files for loading into a Spectrum emulator.

Variables

A **. bas** file begins with an optional variables section, which describes the BASIC variables that were resident in memory when the program was saved.

Each variable declaration will take the form:

• Var [v]:[Type] = [Value]

Where v is the variable name (a\$, a etc). Type would be one of Num, Str, NumArray(), StrArray(), NumFOR. Strings would be Quote (") delimited, with normal BASIC quote-in-quote rules applied.

There are five different types of entry:

Numeric	Var name:Num=value
String	Var name\$:Str="value"
Numeric array	<pre>Var name(dim1, dim2,):NumArray=value1, value2,</pre>
String array	<pre>Var name\$(dim1, dim2,):StrArray="value1", "value2",</pre>
Loop counter	Var name:NumFOR=current, limit, step, loop-line, loop-statement
Multi-dimensional arrays are stored in a linear fashion, so that all the elements of the first dimension precede all those of the second dimension, and so on.

For a loop counter variable, *loop-line* and *loop-statement* represent the statement immediately *after* the FOR statement, i.e. the one to which program flow will jump when the corresponding NEXT is encountered.

Auto-run line number

The variables section is followed by an optional Auto declaration, which specifies the line number at which the BASIC program starts running. This corresponds to the SAVE . . . LINE syntax, and takes the form:

Auto line-number

Program

The next part of the .bas file is the actual program listing. Keywords are stored as plain text (so e.g. **PRINT** is five letters), not the non-standard keyword tokens from the Spectrum character set.

If you have selected **Break multi-statement lines** on BASin's Files tab, any program line containing more than one statement is split across multiple lines in the file, one for each statement. This is indicated by a line ending with a  $\setminus$  backslash, e.g.

# 10 PRINT "this is the first statem PRINT "this is the second statemen

Special characters

Spectrum BASIC supports some non-standard control characters (for colours and text formatting) and graphics characters. In **.bas** files, these special characters are represented by escape sequences, so that the files can be easily

viewed and modified in a text editor. Escape sequences begin with a  $\backslash$  backslash.

The escape sequences for control characters are as follows:

$\{in\}$	Ink colour <i>n</i> , where <i>n</i> is in the range 0 to 7.	
\{pn}	Paper colour <i>n</i> , where <i>n</i> is in the range 0 to 7.	
\{bn}	Bright <i>n</i> , where <i>n</i> is 0 or 1.	
$\{fn\}$	Flash $n$ , where $n$ is 0 or 1.	

Note: Control character escape sequences can be combined, so that for instance  $\{i6p1f1\}$  is equivalent to  $\{i6\} \setminus \{p1\} \setminus \{f1\}$  - flashing yellow ink on blue paper.

The escape sequences for graphics characters are as follows (note the spaces!):

١	The character.	١.	The ■ character.
χ'	The ■ character.	۸.'	The ∎ <sup>■</sup> character.
λ'	The character.	١:	The character.
λ''	The 🗖 character.	\:'	The F character.
١.	The ■ character.	١	The 🗕 character.
\ :	The ∎ character.	١.:	The 📕 character.
Λ'.	The ■ character.	<u>\:</u> .	The 🖿 character.

ν':	The 🗖 character.	\::	The character.
∖letter	A user-defined graphics character, where <i>letter</i> is in the range <b>a</b> to <b>l</b>		

Other escape sequences are as follows:

\@	The @ at symbol.
\\	The $\lambda$ backslash symbol.
ν.	The $f{E}$ pound sterling symbol.
\#nnn	Any character, where <i>nnn</i> is a decimal number in the range 000 to 255.

### **The .bsc file format**

A **.bsc** (BASin code) file contains the contents of a block of memory, along with a descriptive header block. To save a code block, use the **SAVE** . . . **CODE** syntax.

**Note:** The filename extension determines the format in which a code file is saved. If the extension is **.bsc**, BASin saves a **.bsc** file. If the extension is anything else (or missing altogether), BASin saves a raw binary file with no header.

The LOAD . . . CODE syntax allows you to specify a start address in memory and an optional length (in bytes). If the file is too short, BASin will load as much code as is available. If it is too long, BASin will stop loading after the specified length or when the top of memory is reached.

The **.bsc** format begins with a simple 17-byte header, identical to the header saved by a real Spectrum. The header takes the form:

Byte offset	Contents
0	Data type indicator: 3 for a code file.
1-10	Filename, padded with trailing spaces.
11-12	Total length of data block (little-endian).
13-14	Start address in memory (little-endian).
15-16	Reserved.

Bytes 17 onwards are the raw binary data that was saved.

BASin can save memory dumps as either .bsc, which includes the header described above, or as raw bytes if the .bsc extension is absent from the filename - e.g. SAVE "c:\BASin\Screen.scr" CODE 16384, 6912 would save a raw .Scr dump of the current display memory, with no header. This can be loaded into any emulator which supports the loading of Binary files.

### **The .bsd file format**

A . bsd (BASin data) file contains the name and contents of a BASIC array variable. To save variable data, use the SAVE . . . DATA syntax.

**Note:** It is only possible to save array variables with single-letter names, such as **a\$**. This was a limitation of Spectrum BASIC and is preserved in BASin for compatibility.

The .bsd format resembles the .bsc format in that it comprises a standard Spectrum header followed by the raw binary data for the variable. For a discussion of how variables are represented in memory, see Chapter 24 of the manual.

The header takes the form:

Byte offset	Contents
0	Data type indicator: 1 for a numeric array, 2 for a string array.
1-10	Filename, padded with trailing spaces.
11-12	Total length of data block (little-endian).
13	Variable name (a single letter A to Z).
14-16	Reserved.

Bytes 17 onwards are the raw binary data that was saved.

### The .scr file format

A . SCr (screen) file is a memory dump of the Spectrum's *display file*, or screen memory. This file format is supported by many emulators.

The display file starts at address 16384 and is 6912 bytes long. The first 6144 bytes represent the actual screen bitmap (which pixels are set and which are not), and the remaining 768 bytes represent the attributes, or colour information, which can be imagined as a transparent overlay.

The layout of the display file is somewhat unusual. It is split into vertical thirds, each comprising eight rows of characters, and each character is stored in eight (non-sequential!) bytes. The attribute data is stored in a more straightforward way, "reading" the character cells from the top left to the bottom right.

This simple program illustrates the layout of the screen display by filling it with consecutive bytes:

```
10 FOR f=0 TO 6911
20 POKE f+16384, 255
30 NEXT f
```

### **Snapshot file formats**

BASin can open and save **. Sna** and **. Z80** files. These are the most popular snapshot file formats used by Spectrum emulators.

If you don't need to load your programs into other emulators, you should use the .bas file format instead.

A snapshot describes the complete state of the emulated Spectrum at the time it was saved, including the memory contents, port states, and register contents. When you open a snapshot in BASin, the BASIC program, variables, and user-defined graphics are extracted from it and converted into a tape block which the ROM loader understands.

**Note:** BASin saves 128k snapshots in a different manner to 48k.

When saving a snapshot, BASin checks to see which hardware it is compatible with - by testing for 128k specific commands, and also for the use of the UDGs "T" and "U" - which would denote a 48k snapshot for the majority of cases. If there is a clear case for one or either of the two hardware models, BASin will choose that format for saving. If there could reasonably be a choice, then BASin will ask you (assuming you have the relevant options set) which you want to use.

48k Snapshots work as expected - they are a dump of the memory and the state of the emulation as it was when the snapshot was saved. However, due to the fact that BASin is primarily a 48k emulation, 128k snapshots are created by inserting your program and silicon disc contents into a 128k snapshot. This means that when you load the snapshot, you will find yourself at the main 128k menu. To examine your program or run it, you must choose "128 BASIC" from this menu.

**Note:** The . **Z80** file format, developed by Gerton Lunter for his **Z80** Spectrum emulator, has gone through a series of changes, making newer files incompatible with older versions of the emulator. You can choose your preferred version on BASin's Files tab.

### **Running a program**

Programs, when run, will send all their output to the Display Area. This area is analogous to the original Television display utilised by the real Sinclair Spectrum. The PC keyboard operates in a similar manner to the Editor window - and thanks to more ROM modifications, any INPUT statements will be able to use a much more friendly method of entering data into the program.

All programs and direct commands will return to the editor with an error message, even if it is just to say that no errors have occurred. You can get further help with error messages - the Error window and Error Help window provide complete information about BASIC errors

#### **Direct commands**

Program lines can be typed in with a line number and will be accepted and stored for later as part of a program. However, if you type a line with no line number, it will be executed immediately. This is a "Direct Command". You can also use the editor as a simple calculator - type in an expression, such as 2+3/(4-2) and press Return. The result will be displayed in the display area.

All direct commands are "logged" in the Command History Window where they can be re-called for repeated use.

When the command has terminated, or the expression has been evaluated, then usually a "0 Ok" message dialog will appear, indicating that all went well. Closing the "Error Message" dialog will return you to the editor.

# **Command History window**

The Command History window is available from the View menu.

This window keeps a log of all Direct Commands executed by you in the Editor.

Command History	
LET x=10 PRINT x LET x=x+10: PRINT x	
	Send To Edit Close

You can inspect the list by scrolling through it, and recall any of the commands by highlighting it, and clicking the "Send To Edit" button. This will discard any un-stored line that you may have been typing and replace it with the command you chose. The "Close" button will hide the Command History window.

### **Error messages**

When an error message (which includes error "0 Ok" which means that no error happened!) BASin will pop up an Error Dialog which contains the error, along with a short explanation of why it might have happened. Clicking the Okay button will hide the dialog, and return you to the editor. The Continue button will issue a CONTINUE codeword as a Direct Command and resume program execution if possible.

You can turn this behaviour off with the checkbox at the bottom of the Error Dialog or by using the relevant Options page.

# **Error window**

The Error window is displayed when an error occurs in Spectrum BASIC, either while your program is running or as a result of a direct command in the editor.



This window is also used for reports that are not technically errors, such as 0 OK and 9 STOP statement.

For a complete list of possible reports and an explanation of the numbers, see Appendix B.

Click **Continue** to resume execution of the program. This is equivalent to the **CONTINUE** statement, and is only available if the error occurred in a running program.

Closing the window will return you to the Editor window, and if you have asked BASin to do so in the Options dialog, then the editor cursor will automatically be moved to the program line/statement that caused the error

Uncheck the **Notify me of this error type** box if you do not want to see the Error window when this error occurs in future. If you choose to suppress the window for a particular report, it will overwrite the bottom two lines of the screen instead - as on an original Spectrum. You can re-enable the Error window for specific errors on the Error Reporting tab.

**Note:** In the unlikely event that the error-handling routine in Spectrum ROM is called without a legitimate error value in the ERR NR system variable, BASin will display the error message ? Unknown error.

# **Error Help window**

The Error Help window is available from the Help menu.



Use the Error Help window to browse the Spectrum BASIC error messages and their descriptions.

For more information about error reports, see Appendix B.

# Debugging

"Debugging" is a term used in computer programming which means to remove or fix mistakes in your program. These mistakes are called "bugs" because a very long time ago, when computers used actual "thing" like valves to work (rather than a chip) someone once found an actual insect in their computer which was causing it to work in an odd manner.

BASin provides many ways to help debug your programs. Debugging is the process by which you determine where a program is not doing what it should, and narrow down the exact "fix" needed to get it working properly. Most of the time, this is due to an error on the your part - a spelling error, or a misunderstanding of how certain codewords do their jobs.

This chapter deals with the methods you can use to track down bugs, and fix them. All of these are ways to get information about your program and its variables as the program is running. By watching this information and following your code slowly, you can (hopefully) get an idea of what is going wrong.

There is one thing worth mentioning - the more debugging systems you enable, the slower a program will execute. BASin will have to take time away from the emulation of the Spectrum BASIC in order to update the various sources of information you have asked for, which can result in choppy sound and inconsistent program speed even on the fastest of systems.

#### Stepping and tracing

One of the most simple debugging tools is the Program Trace. This can be enabled from the Run Menu in the Editor, and will update the editor window as statements execute in real time. This is quite fast and lines flash by very quickly, but some people find that it's necessary to be able to "watch" how their program runs.

Another good method to use is to pause the operation of your program, without actually stopping it. Your program won't know that it's stopped, so it won't affect it.

This can be useful - once the program is "halted" then you can step through the code one statement at a time, and monitor the program flow and variables as you do. BASin provides three methods of doing this:

- **Single Stepping** which will advance one line at a time each statement will be executed and then control will be returned to the user with the program halted once more.
- **Step Over** which will run emulation until the next statement is ready to run. Although on the surface this looks very much like **Single-Stepping**, the difference can be seen when you step over a **GO SUB** statement. The entire subroutine will be executed as normal, and the program halted when the subroutine **RETURNs**.
- **Run To** which will allow the program to run until the line that the editor cursor is currently resting on is ready to execute.

For more information, see the Run menu.

#### Breakpoints

The user can suspend operation using the Run Menu in the Editor, but you'd be hard pressed to accurately stop just where you want to. This is where Breakpoints and Watches come in very handy indeed.

At their simplest, breakpoints are "flags" attached to a statement which, when the statement is about to be executed, tell the emulation to suspend operations and return control to the user. You can then use the various other tools to examine what is going on - look at variables, for instance. Breakpoints in the editor show up as a red bar on a statement:



Breakpoints in BASin can get a lot more complex than this, however.

You can create a breakpoint by simply double-clicking a statement in the Editor. However, if you open the Breakpoints Window, you can get advanced properties of the breakpoint and attach conditions and logging options to it. With a condition attached, you can tell BASin to suspend program operation only when that condition evaluates to "true", or a non-zero result. This is for more advanced users, and can be confusing to the beginner.

The condition is simply an expression. Whenever that statement is due to be executed, the condition is evaluated, and if a non-zero result obtained then execution is suspended. Similarly, a log of the breakpoint can be added to the Log Window, the text of which is also an expression, which returns a string result. For instance, you can add a log to a breakpoint which sends the contents of a variable to the Log Window. This can be useful for tracking the state of a variable as the program executes - the breakpoint does not always have to suspend execution.

#### Watches

**Watches** are another type of debugging tool, loosely related to breakpoints, but much more flexible. They are a list of expressions (or other types of data, such as variable contents, system variable states, memory states etc) which are updated after every statement executes. The Watch Window is constantly updated as the program runs, to reflect the state of these watches.

Watches may have conditions as well. The program halts when the value of the expression satisfies the condition.

- To create a watch, use the Add Watch window.
- To view the values of watches, or delete them, use the Watch List window.

Alternatively, you can watch variables - and edit their values - in the Variables window and System Variables window.

See also:

- The Variables window
- The Breakpoints window
- The Breapoint-properties window
- The Watchlist window
- The Watch-properties window
- The Log window

### **REM directives**

Spectrum BASIC uses the **REM** statement for comments, which are pieces of text that serve to explain the purpose of the code. **REM** statements have no effect in a running program.

BASin supports a number of REM directives, which are special commands that can be activated during the running of a program by including the appropriate REM statement.

**REM** directives are case-insensitive, but there must be no more than one space between **REM** and the directive.

You cannot use a **REM** directive in a direct command - only in a program line.

REM fast

Tells BASin to start running the program as fast as possible, disregarding any timing constraints. While not true to the behaviour of a real Spectrum, this feature is useful for skipping through time-consuming operations.

REM slow

Tells BASin to resume normal execution speed.

REM log *expression* 

Tells BASin to evaluate *expression* (which may be string or numeric) and write the result to the Log window.

This directive supports BASIC keywords and operators as well as simple variables. Here are some examples:

# 10 REM log i 80 REM log "Reached line 80"

# 100 REM log FN p(j) 500 REM log "Length of b\$ is " + S

**Note:** If an error occurs while evaluating an expression, the error message is written to the Log window instead.

# Variables window

The Variables window is available from the View menu.

🗷 Variables 🛛 🔀			
Name	Туре	Contents	
a	Numeric	2	
b\$	String	"test"	
c	NumArr	0, 0, 0, 0, 0, 0, 0, 0,	
Watch	Edit	Close	

This window keeps a list of any variables that have been declared at runtime or as a Direct Command with a LET statement. Once the variable has been declared, it will appear in the list. The name will show at the far left of the window, followed by its type:

- **Numeric** a variable that holds either a floating point (decimal) number or an Integer (whole) number.
- **String** a variable that contains a string of ASCII characters.
- For Var a numeric, single letter variable, that contains information about the FOR command that created it. The current value, the STEP value, and the limit (TO value) is shown.
- **NumArray** a numeric array. The dimensions of the array are also shown.

• **StrArray** - an array of strings. As for the numeric array, the dimensions are also shown with the type.

After this, the contents of the variable are shown. This display will update in real-time as the program changes the variable contents.

You can use either a click of the right mouse button on the highlighted variable, or use one of the two buttons at the bottom-left of the window to either edit the variable or watch it.

If you elect to watch the variable, then a Watch will be created and added to the Watch Window containing the variable name as it's watch-expression. You can edit this watch with the Watch Properties window.

Editing a simple String or Numeric variable

Editing variables brings up the variable editor. The contents of this window depend upon the type of variable being edited.

🗷 Edit Numeric 🛛 🛛	🗷 Edit String 🛛 🛛 🔀
New value for 'a'	New value for 'b\$'
2	"hello"
Okay Cancel	Okay Cancel

For a simple numeric or string variable, you can alter the contents by typing a new expression into the variable editor's edit field. This expression must be of correct syntax, and evaluate to the type of variable being edited. This means, for example, that any string literals you edit must be enclosed in quote (") marks, but does grant the ability to use such expressions as A\$+B\$ or "Hello"(TO 4) which will of course be evaluated when you click OK, and the result stored in the variable.

Editing a FOR control variable

🗷 Edit FOR Variable 🛛 🔀			
Parameters for f -			
Current Value:			
1			
TO Value:	STEP Value:		
10	0.5		
Repeats at: Line:	Statement		
65534	2		
	Okay Cancel		

A FOR variable will, when edited, allow you to alter the current value, the STEP value, and the limit (the TO value), as well as the line and statement number that the corresponding NEXT statement will jump to when the loop is processed. Note that if the FOR variable is declared as a direct statement, the line to jump to will be 65534, i.e. an impossible jump. Again, all the edit fields will accept any numeric expression as their contents.

Editing an array variable

E	🛛 Edit Numeric Array	×
r	Values for a(2,10)	
k	12	~
	0	
	24	
	0	
	36	
	48	
	0	
	0	
	0	~
	Okay Cance	

Numeric or String arrays will present themselves as a list in the variable editor of their contents. Clicking any of the elements will open an edit field for that element, which again can be any expression relevant to the variable type.

# **Breakpoints window**

The Breakpoints window is available from the View menu.

Breakpoints		
Line Source	Condition	Count
✓ 10:1 PRINT		0
Add New Delete Properties		Close

The breakpoints window lists every breakpoint that the user has set within the program, along with some other information which can be useful when debugging. The list has a small column to the left, which contains a checkbox. When checked, this means that the breakpoint is active. When unchecked, the breakpoint will be ignored and won't suspend program operation. This option can also be set from the context menu activated by right-clicking a breakpoint in the list. After this, the line and statement is listed, in the Line:Statement format (i.e., 10:1 would be line ten, statement 1), along with the source of the line in BASIC code. Any condition that has been set for the breakpoint is listed next to this.

Conditional breakpoints must have their conditions attached using the Breakpoint Properties Window after the breakpoint has been created. This condition is an expression which must evaluate to a numerical result. Whenever program execution reaches the line and statement specified for this breakpoint, the condition (if present) is evaluated. If the result is non-zero, then the breakpoint will suspend program operation and return you to the editor. A result of zero will allow the program to skip the breakpoint, and continue execution.

Finally, to the far right, is the Pass Count. This number will be incremented when program execution reaches the breakpoint, regardless of whether or not execution is suspended by the breakpoint. By watching this value rise, you can see if your program is executing that line of code, and how often. You can add a new breakpoint by either choosing the "Add Breakpoint..." menu item (after clicking your right mouse button in the breakpoints list), or by using the "Add New..." button. A new breakpoint will be created, and the Breakpoint Properties Windowl open allowing you to edit the new breakpoint's attributes. You can also call up the properties for any other breakpoint by either using the "Properties" button, or by using the "Properties" option on that breakpoint's context menu.

breakpoints can be deleted by highlighting a breakpoint in the main list, and either using the "Delete" option on the context (right-click) menu, or by using the "Delete" button.

The context menu for the main list has extra functionality if no breakpoint is highlighted, and will allow you to enable, disable or delete all the breakpoints in the list. These items should be used with care, as they can easily make a mess of all your debugging efforts so far. When a breakpoint is highlighted, then an extra option appears on the menu - "View Source", which will move the Editor cursor to the start of the line and statement that the breakpoint refers to.

# **Breakpoint Properties window**

The Breakpoint Properties window is available from the Breakpoints window.

Breakpoint Pro	perties	
Breakpoint Advance	ed b	
		Enabled 🔽 Break 🔽
Line 10	Number:	Statement:
		Okay Cancel

This window is used to set the behaviour of any of the breakpoints currently set within your program. There are two tabs - The "Breakpoint" tab and the "Advanced" tab.

**Note:** When creating a new breakpoint, this is known as the Add Breakpoint window.

The Breakpoint tab

In the "Breakpoint" tab, you can set the basic options for your breakpoint. The "Enabled" checkbox allows you to "turn off" a breakpoint, which will remain completely inactive until you want to use it, and the "Break" checkbox lets you choose whether or not this breakpoint should suspend program operation when it fires. The Line and Statement number that the breakpoint will trigger at is displayed in the two edit boxes below these options.

#### Advanced tab

The "Advanced" tab gives you the opportunity to add a condition to the breakpoint. If the breakpoint is enabled, then the condition will be tested if it is present. The condition can be any Sinclair BASIC expression, which includes the use of variables (as long as they are defined by your program). The expression should resolve to a numerical result, and if it does then the breakpoint

will trigger when that result is non-zero.

Finally, you can also specify a log entry which will be made when the breakpoint fires. Bear in mind that the log text specified will only be logged if the breakpoint is enabled, and any conditions have been met. This log text can, like the condition, also be an expression of any type. You can indicate that this is the case by checking the "Log Expression" checkbox below the edit field.

# Watch List window

The Watch List window is available from the View menu.

Watch List	
Expression	Result
<ul> <li>✓ a</li> <li>✓ Mem Byte [22528]</li> </ul>	2 56
✓ SysVar: CHARS	15360 (0,60)
Add New Delete Properties	Refresh Close

Watches are another tool which, like breakpoints, are evaluated at runtime. Unlike breakpoints, they are evaluated after every line and statement of your program, so they can take up quite a lot of CPU time if you have many of them present in the Watch list. Watches consist of different types - the most basic being an expression which the user enters, and is evaluated as the program runs. Other watch types include variable watches, system variable watches, and watches that monitor memory addresses. This can be very handy for keeping an eye on how your program affects the environment it runs in.

The main area of the window is the Watch List. This lists, like the Breakpoints Window, every watch declared by you. To the left of each entry is the "Enabled" checkbox - you can disable watches temporarily with this. The "Result" field to the right will inform you that the Watch is disabled if you choose to do this.

After the Enabled box, the expression or type of watch is listed. Variables are listed with a prefix "Var:", System variables are given a "SysVar:" prefix, and memory watches are listed with a "Mem" prefix followed by the data size that they are watching (Byte, Word or DWord). Finally the result of the watch, if it is enabled, is shown. In the case of an expression, the result of the evaluation (or any syntax errors in the expression) is shown. Variables will display their contents, unless they become undefined due to a CLEAR statement for example, where they will show the "Variable not found" error.

Managing watches

You can use the "Add New..." button to add a new watch (the Watch Properties Window will pop up to allow you to customise the new Watch), or right-click in an empty part of the watch list and choose the corresponding menu item. Delete watches using the "Delete" button after highlighting a watch in the list, or again use the context menu on your right mouse button. You can alter any watch by selecting it and using the "Properties" button, which will bring up the Watch Properties Window.

If you find that your watches are at odds to your program state (after you maybe POKE'd one of your variables programmatically for example), you can force a manual update of all watches by clicking the "Refresh" button, which again is available on the context menu.

### **Watch Properties window**

The Watch Properties window is available from the Watch List window.

🗷 Watch Properties 🛛 🛛 🛛			
	Enabled	K	
Watch Type Expression	Expression	~	
a			
	Break on non-zero		
0	kay Cancel		

When you create a new Watch, or edit a Watch you have already created, this window will allow you to customise the way that the Watch behaves.

**Note:** When creating a new watch, this is known as the Add Watch window.

Firstly, the "Enabled" checkbox will allow you to turn on or off the watch in question. A disabled watch will not be evaluated at runtime, and so can speed up debugging when many watches are created but not necessarily needed at the current time.

After this, you can choose what type of Watch you want from the "Watch Type" drop-down list:

Types of watches available

#### • Expression Watches

An Expression watch will evaluate the expression you specify in the edit field below. This can be any Sinclair BASIC expression, including variables and memory PEEKs. Below this, you can opt to break execution (suspend the program) if the expression evaluates to a non-zero result. This is similar to the conditional breakpoint, but different in one important respect. Watches are evaluated as each line and statement is executed, and so are constantly updated. A conditional breakpoint will only be evaluated when the specified line and statement number has been reached, so an Expression Watch can break whenever the expression results non-zero, which can be handy for tracking down where a particular variable is corrupted, say.

#### • Variable Watches

The Variable watch is similar to the Expression Watch but is restricted to only watching the variables that have been declared at runtime or as a Direct Command. These watches can not break execution. You can choose which variable by either typing in the name, or selecting it from the drop-down list.

#### • System Variable Watches

These are similar to the Variable Watches, in that you can only select to monitor a System Variable (see Chapter 25 of the Sinclair BASIC Manual for more on these) from the drop-down list. These watches cannot break program execution. They will be displayed in their correct format - two-byte words, or single byte values, or lists of multi-byte contents.

#### • Memory Address Watches

These watches will monitor the contents of memory addresses. They can display, from the address which lies in the range 0 to 65535, one byte, a Word (two bytes) or a Dword (four bytes). This will be updated between statements, as the other watches are. Again, these watches cannot break program execution.

# Log window

The Log window is available from the View menu.



This window displays messages logged by BASin, such as breakpoint log text and syntax errors when pasting multiple lines from the Add Code window.

Click **Clear** to delete all messages from the window.

Click **Save As** to save the logged messages to a text file.

# **Print Preview window**

The Print Preview window is available from the File menu, the display window's Display menu, and the ZX printer output window.

💵 Print		×
Print Pre	view	
Print Opt	Page 1/1	•
Break at C	olumn Data Source ZX Printer	~
32	Scaling (%) 407	*
Display	as-is Multiple column	s
Printer Se	ettings Print Close	

This window is opened automatically when any of the three printing options are chosen: the Print option of the File Menu in the Editor, the Print option of the Display menu on the Display Window, and the Send To Printer button in the ZX Printer Output Window. It allows you to set various options as to how the output will be printed to the PC printer.

The window is divided into two areas: the options, and the Print Preview.

The **print preview area** is a graphical representation of how the output will appear on the page. The size of the image is taken from your printer's current orientation and paper type settings, and the actual data is laid out according to the options at the bottom of the window.

The options start with the **Data Source**, which is usually set by the window that opened the dialog. You can change between the current screen, the current BASIC program, and the ZX Printer output. Each has options of their own which can be set.

When printing the current BASIC program, you can use the numeric field to the left of the options area to set at which character column the lines of BASIC will be wrapped around to the next line. The output produced by this source will be as close to the Editor's display as possible (taking into account the line wrapping), with the same syntax highlighting options as you have selected in the Options Window.

When printing the current screen, you can elect at the lower left of the options area to either grab the display "as-is", which will take the current display size and any scaling options into account, or to take the display as the spectrum sees it. Both options will take the border area into account when printing.

When printing the ZX Printer output, you cannot set either of the above options, and the only options available to you are the standard set available to all three Data Sources.

The scaling allows you to inflate or deflate the printed output. Many printers operate at much, much higher resolutions (dots per inch, or DPI) than the ZX Spectrum's printer could manage. For this reason, you can scale the output up or down to enhance readability or maximise space usage.

Below the scaling option you can also select Multiple column output, which only applies to the current BASIC program and the ZX Printer output. If there is room on your page, BASin will allow the printed output to run in columns down the page before heading into a new page.

Below these are the following buttons:

#### • Printer Settings

This will open your printer's Preferences control panel applet, allowing you to set paper source and orientation or quality settings. Any changes will be reflected in the preview area at the top of the window.

#### • Print

Will bring up the standard windows Printer dialog, where you can set the number of copies and the pages you wish to print before commencing the print job. Note that you can also change the printer you are sending the job to using this window - which can cause problems if the paper sizes and orientation of another printer differ from the ones used in the preview.

#### • Close

Will close the Print Preview Window without printing anything.

# **BASin's tools**

BASin offers a number of tools, accessible through the Tools Menu, which can make certain aspects of programming a touch easier by removing the tedium of say, entering long lists of DATA values, or designing UDGs on paper.

Some of these tools will do a certain amount of programming for you, others are aids to programming. The Token Table Window, for example, makes it easier to insert colour control characters or keyword tokens into your program, and the Renumber Tool allows you to make your program tidier and create room for more lines between lines that have become a bit tight in their numbering.

Choose a tool that you want to read the help for:

- The Token Table helps you to locate and insert special symbols.
- The **UDG Editor** can manipulate user-defined graphics (UDGs).
- The Renumber tool changes the line numbering in all or part of your program.

#### Other tools

These tools are used by one or more of the tools or debugging utilities, and as such are not available for use by the user except in conjunction with those parts of BASin. However, they are documented here:

• The Add Code window controls the insertion of groups of program lines.
# **Token Table window**

The Token Table window is available from the Tools menu.

KV T	ok	en	Ta	ıЫ	e																				×
Alph	na-N	lun	neri	cs į	/ Gr	apł	nics	) (	Con	tro		ode	s	Ke	yw	ord	s								
Alp	har	num	heri	cs a	and	Sy	mb	ols:	_						_								_		
	1		#	串	%	&	1	(	)	×	+	1	-	•	1	Ø	1	5	З	4	5	6	7		
8	9	:	5	<	=	>	?	0	A	в	С	D	Ε	F	G	н	Ι	J	К	L	М	Ν	0		

This tool opens up a window which contains a tabbed set of pages which contain the usually hard to find characters and tokens present in the Sinclair ASCII set. You can browse these pages to find the token you want, whereupon double clicking it will send that character or token to the Editor window. Clicking once on these will show a larger version, which can be useful to distinguish between them.

The status area at the bottom of the window, below the tabs, shows which character code (CHR\$ code) corresponds to the character or control code/token you are currently pointing at.

The window is laid out in three tabs:

Alpha-numerics/Graphics

This page shows three areas of characters that are available to Sinclair BASIC - the Character set, which contains all the characters from the space character through to the copyright symbol.

Alp	har	num	eri	cs a	and	Sy	mbo	ols:															
	1		#	\$	%	&	1	(	)	×	+	,	-	•	1	Ø	1	5	з	4	5	6	7
8	9	:	5	<	=	>	?	0	A	в	С	D	Е	F	G	н	I	J	к	L	М	Ν	0
Ρ	Q	R	s	т	υ	V	υ	×	Υ	z	C	1	J	Ť	_	£	а	ь	с	Р	е	f	9
h	i	j	ĸ	ι	m	п	0	Ρ	q	٢	s	t	υ	v	ω	×	У	z	£	Т	3	~	©
Pre	set	"C	hea	que	rbo	ard	G	rap	hic	s									~	Us	e C	HA	RS
	•	-																					
Use	er D	efi	ned	Gr	apł	nics	(U	DG	s)														
A	в	С	D	Е	F	G	н	I	J	к	L	м	Ν	0	Р	0	R	s	т	U			

There is a checkbox below this area, marked "Use Chars" - this, when enabled, will instead of using the default Sinclair Character set, take the graphical representations from the memory region which is pointed to by the CHARS system variable. This allows you to see the characters as they would appear on-screen.

#### Control codes

This page displays the colour and cursor movement characters for you to insert into your program.

Attribute C	ontrol T	okens	:						
Flash	Off	i	Flas	h On		Brigh	t Of	F F	
Bright	On	-	True	Vide	20	Inver	se l	/ideo	0
Colour Tok	ens:								
Black	Blue	Red	Mag	enta	Greet	n Cyan	Yel	.low	White
Black	Blue	Red	Mag	enta	Greet	n Cyan	Yel	LOU	White
Editing Con	trols:					-		0	
CAPS	E	TIC		<⊅ Le	ft	🕏 Rigi	n t	3	
& Down	÷	UP		DELE	TE	ENTER			

These will be sent as "Hex chars" - small custom characters which comprise the hexadecimal equivalent of their CHR\$ codes. For colour control characters, this usually entails sending two characters to the editor - the colour control code and it's parameter. For example, the code for setting the INK is number 16 - so that will be sent first, followed by the ink number from 0 to 7. The Status area, although showing the correct PRINT code for the colour controls, does not show the Editor codes needed. The correct characters will be sent though.

#### Keywords

This page allows you to send BASIC keywords to your program.

Keywor	ds:																
RND	INH	EY\$	PI	FN	P	OIN	т	sci	REE	EN \$	F	атті	R	AT	TF	ав	VAL\$
CODE	: V	AL	LEN	SI	N	CO	s	та	N	ASN	1	AC:	5	ATI	N	LN	EXP
INT	SQ	R SG	N F	BS	PE	EEK	I	Z	USF	۲ S	TF	₹\$	сн	R\$	Ζ	от	BIN
OR	AND	< =	> =	$\langle \rangle$	L	IN	E	тне	Z	то		STE	ΕP	DE	F	FN	CAT
FORM	IAT	MOU	E E	RAS	5E	OP	EN	#	CI	LOS	E	#	ME	RG	E	VE	RIFY
BEEP	0:	IRCLI	E II	vк	PAP	PER	F	LAS	БΗ	BR	IG	нт	II	<b>VUE</b>	RS	E	OVER
OUT	LF	RINT		LIS	т	ST	DP	R	EAC		) A	та	R	EST	OR	E	NEW
BORD	ER	CON	TINU	JE	DI	4 F	EM	F	OR	GC	) .	то	GC	) 5	UВ	I	NPUT
LOAD	) L	IST	LET	P	AU	SE	NE	ЕΧТ	F	окі	Ш	PR	IN	т	PL	от	RUN
SAVE	R	ANDO	MIZ	E :	IF	CL	s	DR	aw	CI	LE	AR	R	ETU	JRN	1	COPY

Like the Control Codes, these will all send the corresponding character to the editor as a "Hex char", with the corresponding hexadecimal value. This allows you to insert the tokens for each of the BASIC's keywords into strings in your program. The advantage of this is that you get a complete word using only one byte of memory. If you elect to send a keyword outside of a string literal, then the sequence of letters that make up the keyword will be sent instead.

# The UDG Editor window

File Edit Help

The UDG Editor window is available from the Tools menu.

One of the most laborious and time-consuming tasks when programming in BASIC is the construction of UDGs for your programs. There is space in memory for up to 21 of these, and in the past they had to be designed on squared paper, and the binary values of the character's lines calculated and entered in using either POKEs or DATA statements.

The UDG editor provides a user-friendly and intuitive graphical interface for creating these graphics.

The window comprises the following areas:

Editing grid



This area represents, in a large form, the 8×8 character grid used to create one character. You can use the mouse in here to set or un-set pixels in the character. You can select the character you want to work on by clicking in the Character Selection Area, and any changes you make will be shown there as you make them.

A preview of your current character is also displayed at the top right of the grid region.

#### Character selection area

A	в	С	D	Е	F	G	н	I	J	к	L	М	Ν	0	Р
Q	R	5	т	U											
			1												
															2 1

This area, to the right of the window, contains a "palette" of characters available. When editing UDGs, this contains 21 graphics.

You can use the left mouse button in a single "click" to send a character to the editing grid. You can also use a "drag and drop" operation on these characters:

- Use the left mouse button to select a character, and without letting go of the left mouse button, "drag" the character to another character in the selection area to make a *copy* of that character in that cell.
- Use the right mouse button to drag a character to exchange it with the

character in the destination cell.

- Use the left mouse button to drag a character to the editing grid, to assign that character to a particular placeholder on the grid.
- Use the right mouse button to copy a character's data from one character to the editing grid. This is the same as copying between characters on the palette using the left button detailed above.

The Menu strip

The main menu for the UDG/Charset editor allows you to access advanced functions. From left to right, the menu consists of:

#### The File Menu

• New

Clears all the UDGs away and lets you start afresh with a new set.

# • Exit

Closes the UDG/Charset editor.

## • Exit and discard

Closes the window and undoes any changes you have made to the UDGs.

### The Edit Menu

This menu allows you access to a simple Undo feature, tools for manipulating the grid and individual graphics, and to utilise the clipboard. This clipboard is internal to the UDG editor, and does not give access to the Windows system clipboard.

# • Undo

Un-Does the last change you made to the current character.

### • Grid

Gives tools for working with the editing grid.

### • Fill

Fills the grid with the currently selected graphic in the palette to the right of the window. This can be useful for designing graphics that must tile seamlessly.

### • Clear

Clears the entire grid, which will also clear all graphics assigned to the grid.

### • Invert

Inverts every graphic assigned to the grid, which will invert the entire grid.

### • Flip

Brings up a submenu which allows you to flip the entire grid.

# Horizontal

Flips the whole grid left to right. This means that all graphics in the grid will be flipped, and also the data from grid regions at the left of the grid will be moved to the right, and vice-versa.

## Vertical

As for horizontal flipping, but flips vertically, top to bottom.

### • Rotate

Brings up a submenu for rotating the grid. **Note:** These operations can only be performed on graphics with a square data format (such as 8x8 pixels, or 16x16 pixels).

## 90° Left

Rotates the whole grid 90 degrees anticlockwise. As with flipping, this means that not only will each graphic be moved, but characters will assume the data of other characters as they are moved into place.

# 90° Right

Rotates the whole grid 90 degrees clockwise. Operates the same way as the 90° Left option, in that the whole grid is rotated.

### • Shift

Brings up a submenu which gives options for scrolling the grid, with wraparound.

## Left

Shifts the grid left by one pixel. Graphics will overflow into one-another at their edges.

# Right

Shifts the grid right by one pixel. As with all shift operations, graphics will overflow at their edges.

• Up

Shifts the whole grid up one pixel, wrapping around at the bottom of the grid.

### Down

Shifts the grid down one pixel, wrapping round at the top.

### • Characters

Provides tools for working with the individually selected (highlighted) graphic in the palette to the right of the window. Many of these operations are similar to the ones for the editing grid, but work in subtly different ways.

### • Cut

Stores a copy of the current image on the Graphic Editor clipboard, and clears the current character from the editing grid.

### • Copy

Stores a copy of the current graphic on the Graphic Editor clipboard.

### • Paste

Retrieves any graphic stored on the Graphic Editor clipboard, and sends it to the currently selected graphic slot, replacing any graphic that is already there.

### • Insert New

Inserts a new blank graphic at the currently highlighted position.

### • **Delete**

Removes the currently selected graphic. This will reduce the number of graphics available in the palette by 1.

### • Clear

Clears the currently selected graphic to a blank sheet.

### • Invert

Swaps the set or un-set state of every pixel in the Editing Grid, thus inverting the character.

## • Flip

Brings up a submenu which allows you to flip the currently selected graphic.

# Horizontal

Flips the current character in the vertical plane.

### Vertical

Flips the current character in the horizontal plane.

### • Rotate

Brings up a submenu which allows you to rotate the character by 90 degrees.

## ■ 90° Left

Rotates the current character 90 degrees anticlockwise.

# 90° Right

Rotates the current character 90 degrees clockwise.

## • Shift

Brings up a submenu, with four options for shifting the current character around the Editing Grid.

# Left

Shifts all pixels one place to the left, with wraparound.

# Right

Shifts all pixels one place to the right, with wraparound.

• Up

Shifts all pixels up one place, with wraparound.

### Down

Shifts all pixels down one place, with wraparound.

# **Renumber window**

The Renumber window is available from the Tools menu.

Renumber		
Renumber		
The Whole Prog	gram	
	From Start:	To End:
	1	9999
Using Lines		
Starting at Line:	Increme	nt in Steps of:
10	10	
	Okay	Cancel

This tool is very handy for any BASIC, such as Sinclair BASIC, which uses line numbers. Whilst writing a program, you may find that you need to insert lines between existing lines, and that after a while the numbering is so tight that there is no more room.

This tool alleviates this problem. It will take each line number, and replace it with the numbers you choose, which can (assuming that you set a sane Step value) increase the number of available new lines.

You can, using the first option, elect to either renumber the whole program, or just a portion. If you decide to renumber just a portion (the "**Whole Program**" checkbox is un-checked) then the following options must be set:

#### • From Start

The line number that begins the block of code you wish to renumber.

#### • To End

The line number that lies at the end of the block of code you wish to renumber.

**Note:** Both options must be in numerical order - the end value cannot be smaller than the start value.

From hereon, you can set the parameters for the renumbering itself. You can use the options:

#### • Starting At...

This option specifies what line number the renumber should use as the first line.

#### • Increment in steps of...

Each new line that is renumbered will be the previous line's number, plus this amount. Setting to 10 is probably best, as it gives a nice amount of space between the lines. Setting this to 1 will renumber sequentially as 1, 2, 3, ... 10, 11, 12 etc, but will prevent you from inserting any new lines into the program.

**Note:** A note of warning - if a block renumber operation needs to overwrite any existing lines then they will be overwritten regardless of the settings in the Options Window. A log entry will be made to inform you of the fact.

Only lines that contain an absolute reference to another line number will be renumbered. Any statements, such as GO TO, GO SUB, RESTORE etc that reference a line by a calculation, i.e.

#### GO TO A\*10

Will not be renumbered, and again a log entry warning you of the fact will be made. If any log entries are made, then the log window will be shown after the renumber is complete.

# Add Code window

The Add Code window appears when a group of lines is about to be inserted into your program.



The code you paste in must be checked and inserted properly. To this end, when BASin detects such an operation, it sends the code to the **Add Code window** first.

The Add Code window displays the lines to be inserted in a simple text editor, where you can make any changes that you require before confirming the insertion.

If a line to be pasted would overwrite an existing line, what happens depends on the **Line overwrite protection** setting on the **BASin tab**. If either type of overwrite protection is enabled, BASin will prompt you to approve or reject the replacement line. Otherwise, the existing line will be replaced automatically.

If a syntax error occurs during insertion of program lines, the error will be automatically displayed in the Log window and the remaining lines will appear in the Add Code window for correcting.

# **Options window**

The Options window is available from the Tools menu.

Options should not be changed by the beginner, as you can change the way that BASin works.

mulation	Display	Sound	Files			
BASin	Editor fonts	Error Reporting				

This window allows you to configure the behaviour and appearance of BASin.

The options are split across two tabs:

- BASin tab.
- Error reporting tab.

# **BASin tab**

The **BASin** tab can be found in the Options window.

Options			×
Emulation	Display	Sound	Files
BASin	Editor fonts	Error	Reporting

Programming aids

#### **Restore previous session**

When the user exits BASin, the current state of the editor and the emulator is saved in the BASin folder under the filename "Session.bin". It is safe to delete this file, as BASin does not require it to run. When this option is enabled, and the file exists in BASin's folder, BASin will load that state, allowing you to continue programming where you left off.

#### Show syntax helper on startup

If this box is checked, the syntax helper appears in the editor when you start BASin. To show or hide it temporarily, use the option on the View menu.

#### Auto line number (.)

If this box is checked, you can start program lines with a . full stop instead of a line number. BASin will automatically choose a line number for you: the next available multiple of 10 (at the end of a program) or a number halfway between the last entered line and the one that follows it (in the middle of a program).

#### **Predictive auto-typing**

If this box is checked, BASin will suggest likely keywords while you are typing. Keyword suggestions are displayed in grey ahead of the edit cursor. You can accept a suggestion by pressing ➡, or ignore it by continuing to type.

**Note:** On slower computers, BASin may perform poorly if this option is enabled.

#### Line overwrite protection

This editor feature helps to protect you from accidentally overwriting program lines with new ones. By default, BASin will alert you by flashing a green cursor at the start of the line, and making the error-beep sound. Pressing return a second time signals to BASin that you have understood that you are about to replace a line, and performs the replace operation. If you elect not to replace, you can edit as normal, changing the line number to one more appropriate. There are three methods available for this option:

- **None** No prevention or warning will be made, and lines will be replaced regardless.
- **Protect all** All operations that will replace older lines, including edited lines (as opposed to newly created lines) will warn you before replacing.
- **Only for new lines** Edited lines will not warn, only lines that have been typed from scratch.

**Note:** the Add Code Window treats all lines as "new" and will warn on all that may cause a line-replace, if this option is set to either of the latter two types.

#### **Brackets completion**

Determines whether BASin automatically balances unmatched opening brackets with corresponding closing brackets.

- **None** Does not automatically insert brackets.
- Auto typed As you type, BASin will close brackets for you. If you type a "(" character, BASin will add a closing ")" ahead of the cursor, which will stay

ahead until you step past it using the right cursor key.

• **Auto completion** - Inserts any missing closing brackets when you enter a program line.

#### **Graphics modifier**

Whilst editing programs, graphics mode allows you to enter colour codes and user defined graphics (UDGs). This option determines which key is used to enable graphics mode.

- Alt Gr (XP/NT Only) This option, as it says, is available to users of NT based operating systems only. It allows you to use the right-alt (Alt Gr) to enter graphical characters, and acts as a *modifier* key you hold it down as long as you need to enter graphics, then release it when you're done.
- Scroll Lock Users of Windows 9x based systems can use this (or the following) option as a *toggle* key press once to enter graphics mode, and press again to leave graphics mode.
- **Num Lock** The **Num Lock** key is used to toggle graphics mode in the same manner as the **Scroll Lock** key.

# **Error reporting tab**

The **Error reporting** tab can be found in the **Options window**.

Options			
Emulation	Display	Sound	Files
BASin	Editor fonts	Error	Reporting

Error notification

#### List of error messages

Basin can report errors exactly as the original Spectrum would have done - by clearing the two lower lines of the Display Window and writing the error there. This tends to corrupt the display, which can be annoying if you were working in that area using PRINT # or INPUT #, for example. The list of checkboxes here represents all the error messages that Sinclair BASIC can produce.

Error messages in the list that are checked will be displayed in the Error window when they occur. Those that are unchecked will overwrite the bottom two lines of the screen display (as on a real Spectrum).

#### Move editor cursor to error

If this box is checked, the editor will automatically move the cursor to the line and statement where the error occurred.

# Sinclair Spectrum BASIC Manual

*Sinclair ZX Spectrum: BASIC Programming* By Steven Vickers. Edited by Robin Bradbeer.

#### Contents

8	
Chapter 1	Introduction A guide to the ZX Spectrum keyboard and a description of the display.
Chapter 2	Basic programming concepts Programs, line numbers, editing programs using ↓ and ↑, RUN, LIST, GO TO, CONTINUE, INPUT, NEW, REM, PRINT, STOP in INPUT data, ESC (BREAK)
Chapter 3	Decisions IF, STOP, =, <, >, <=, >=, <>
Chapter 4	Looping FOR, NEXT, TO, STEP. Introducing FOR - NEXT loops
Chapter 5	Subroutines GO SUB, RETURN
Chapter 6	READ, DATA, RESTORE
Chapter 7	Expressions Mathematical expressions using +, -, *, /, scientific notation and variable names.
Chapter 8	Strings Handling strings and slicing.
Chapter 9	Functions User-definable functions and others readily available on the ZX Spectrum using DEF, LEN, STR\$, VAL, SGN, ABS, INT, SQR, FN

Chapter	Mathematical functions
10	Including simple trigonometry: ↑, PI, EXP, LN, SIN, COS, TAN, ASN, ACS, ATN
Chapter	Random numbers
11	Using RANDOMIZE and RND.
Chapter	Arrays
12	Strings and numeric arrays - DIM.
Chapter	Conditions
13	Logical expressions: AND, OR, NOT.
Chapter 14	The character set A look at the ZX character set including graphics and how to construct your own graphic characters: CODE, CHR\$, POKE, PEEK, USR, BIN
Chapter	More about <b>PRINT</b> and <b>INPUT</b>
15	Some more complicated uses of these commands using separators: , ; ', TAB, AT, LINE and CLS.
Chapter	Colours
16	INK, PAPER, FLASH, BRIGHT, INVERSE, OVER, BORDER
Chapter	Graphics
17	PLOT, DRAW, CIRCLE, POINT
Chapter	Motion
18	Animated graphics using PAUSE, INKEY\$ and PEEK
Chapter	BEEP
19	The sound capabilities of the ZX Spectrum using BEEP.
Chapter	File Storage
20	How to store your programs on cassette tape: SAVE, LOAD, VERIFY, MERGE.
Chapter	The ZX Printer
21	LLIST, LPRINT, COPY

Chapter	Other equipment				
22	Connecting the ZX Spectrum to other machines and devices.				
Chapter	IN and OUT				
23	Input/output ports and their uses: IN, OUT				
Chapter	The memory				
24	A look at the internal workings of the ZX Spectrum: CLEAR				
Chapter 25	The system variables				
Chapter	Using machine code				
26	Introducing <b>USR</b> with a numeric argument				
Appendix A	The character set				
Appendix B	Reports				
Appendix	A description of the ZX Spectrum for reference				
C	The BASIC				
Appendix D	Example programs				
Appendix E	Binary and hexadecimal				

Original edition published in 1982 by Sinclair Research Limited.

Subsequently converted to ASCII text and published in 1995 by Chris Owen on behalf of Amstrad plc.

HTML version by Pete Robinson.

This version reformatted and modified for BASin by Paul E Collins and published in 2004.

This manual is freely distributable but must not be distributed without this notice.

© Copyright 1995 Amstrad plc - all rights reserved.

### **Chapter 1: Introduction**

Before you start programming, you should be aware of the difference between commands and instructions. Commands are obeyed straight away, and instructions begin with a line number and are stored away for later. You may also be aware of some simple commands: PRINT, LET, and INPUT (which can be used on all machines that use BASIC), and BORDER, PAPER and BEEP (which are used on the Spectrum).

This BASIC manual details various programming concepts and commands, telling you exactly what you can and cannot do. You will also find some exercises at the end of each chapter. Don't ignore these; many of them illustrate points that are hinted at in the text. Look through them, and do any that interest you, or that seem to cover ground that you don't understand properly.

Whatever else you do, keep using the computer. If you have the question "What does it do if I tell it such and such?" then the answer is easy: type it in and see. Whenever the manual tells you to type something in, always ask yourself, "what could I type instead?", and try out your replies. The more of your own programs you write, the better you will understand the computer.

At the end of this programming manual are some appendices. These include sections on the way the memory is organised, how the computer manipulates numbers, and a series of example programs.

The keyboard

BASin uses the standard PC keyboard and a version of the ZX Spectrum 128 editor. The various modes and single-key entry system from the original 48K Spectrum do not apply. See The keyboard for information on entering special characters that do not appear on a PC keyboard.

While you type, your position is indicated by the cursor, a flashing block that shows where the next character from the keyboard will be inserted. The cursor can be moved about with the four directional arrow keys: it will automatically jump to the end of a line rather than wander into unused space. If any key is held down for more than about 2 or 3 seconds, it will start auto-repeating.

When **Enter** is pressed, the current line is executed, entered into the program, or used as **INPUT** data as appropriate, unless it contains a syntax error. In this case the cursor changes colour (from blue to red) and moves to the approximate position of the error, and the computer sounds a beep.

As program lines are entered, a listing is built up in the editor. You can use move the cursor onto any line in the listing with the keys  $\clubsuit$  and  $\clubsuit$ , and edit the existing line by typing your changes and pressing **Enter**.

When a command is executed or a program run, output is shown in a separate display window. Error reports may appear in the bottom part of the display window or in a message box. These reports are described in Appendix B.

The **Esc** key (and, in certain circumstances, **Space**) acts as a break, stopping the computer with report D or L. This is recognised:

- 1. at the end of a statement while a program is running, or
- 2. while the computer is loading, saving, or printing.

The display window

This has 24 lines, each 32 characters long, and is divided into two parts. The top part is at most 22 lines and displays either a listing or program output. When printing in the top part has reached the bottom, it all scrolls up one line; if this would involve losing a line that you have not had a chance to see yet, then the computer stops with the message SCroll?. Pressing any of the keys n, **Space** or **Esc** will make the program stop with report

### D BREAK - CONT repeats

...any other key will let the scrolling continue. The bottom part is used for inputting commands, program lines, and **INPUT** data, and also for displaying

reports. The bottom part starts off as two lines (the upper one blank), but it expands to accommodate whatever is typed in. When it reaches the current print position in the top half, further expansions will make the top half scroll up.

Chapter 2 ➡

## **Chapter 2: Basic programming concepts**

Summary

- Programs
- Line numbers
- Editing programs using **+** and **+**
- RUN, LIST
- GO TO, CONTINUE, LET, INPUT, NEW, REM, PRINT
- **STOP** in **INPUT** data
- **Esc** (BREAK on a real Spectrum)

Type in these two lines of a computer program to print out the sum of two numbers:

# 20 PRINT a 10 LET a=10

Because these lines began with numbers, they were not obeyed immediately but stored away, as program lines. You will also have noticed here that the line numbers govern the order of the lines within the program: this is most important when the program is run, but it is also reflected in the order of the lines in the listing that you can see on the screen now.

So far you have only entered one number, so type

### 15 LET b=15

and in it goes. It would have been impossible to insert this line between the first two if they had been numbered 1 and 2 instead of 10 and 20 (line numbers must be whole numbers between 1 and 9999), so that is why, when first typing in a program, it is good practice to leave gaps between the line numbers. If you do get stuck, though, BASin provides a line renumbering option on the Tools menu.

Now you need to change line 20 to

### 20 PRINT a+b

You could type out the replacement in full, but it is easier to move onto the existing line and edit it in place. The quickest way to move to a line is the Go to Line Number command on the Search menu.

Move to line 20, and keep pressing the  $\rightarrow$  key until the cursor moves to the end of the line. Then type

#### +b (without **Enter**)

The line should now read

#### 20 PRINT a+b

Press **Enter** and it will replace the old line 20.

BASin has a safety feature that was not present on the original ZX Spectrum. When you enter a new line with the same number as an existing line, the cursor turns green and jumps to the line number to warn you. Pressing **Enter** a second time will replace the old line with the new one. This behaviour can be turned off in BASin's options.

Run the edited program using **RUN** and **Enter** and the sum will be displayed.

Run the program again and then type

#### PRINT a, b

The variables are still there, even though the program has finished.

If you enter a line by mistake, say

### 12 LET b=8

it will go up into the program and you will realise your mistake. To delete this unnecessary line, type

12 (with Enter of course)

Lastly, type

### LIST 15

You will now see in the display window

# 15 LET b=15 20 PRINT a+b

Line 10 is not present in the listing, but it is still in your program - which you can prove by scrolling through the program in the editor. The only effects of **LIST 15** are to produce a listing that starts at line 15, and to put the program cursor at line 15. If you have a very long program, then **LIST** will probably be a more useful way of moving the program cursor than  $\clubsuit$  and  $\clubsuit$ .

This illustrates another use of line numbers: they act as names for the program lines so that you can refer to them, rather like the way in which variables have names.

**LIST** on its own makes the listing start at the beginning of a program.

Another useful command is **NEW**. This erases any old programs and variables in the computer. Now carefully type in this program, which changes Fahrenheit temperatures to Centigrade.

```
10 REM temperature conversion
20 PRINT "deg F", "deg C"
30 PRINT
40 INPUT "Enter deg F", F
50 PRINT F,(F-32)*5/9
60 GO TO 40
```

Although **GO TO** has a space in it, it is really all one keyword. The space is optional.

Now run it. You will see the headings printed on the screen by line 20, but what happened to line 10? Apparently the computer has completely ignored it. Well, it has. **REM** in line 10 stands for remark, or reminder, and is there solely to remind you of what the program does. A **REM** command consists of **REM** followed by anything you like, and the computer will ignore it right up to the end of the line.

By now, the computer has got to the **INPUT** command on line 40 and is waiting for you to type in a value for the variable F - you can tell this because there is a flashing cursor at the bottom of the display window. Enter a number; remember **Enter**. Now the computer has displayed the result and is waiting for another number. This is because of line 60, **GO TO 40**, which means exactly what it says. Instead of running out of program and stopping, the computer jumps back to line 40 and starts again. So, enter another temperature. After a few more of these you might be wondering if the machine will ever get bored with this; it won't. Next time it asks for another number, type **STOP**. The computer comes back with a report **H STOP in INPUT**, **40:1**, which tells you why it stopped, and where (in the first command of line 40).

If you want to continue the program type

#### CONTINUE

and the computer will ask you for another number.

When **CONTINUE** is used the computer remembers the line number in the last report that it sent you, as long as it was not **O OK**, and jumps back to that line: in our case, this involves jumping to line 40, the **INPUT** command.

Replace line 60 by **GO TO 31** - it will make no perceptible difference to the running of the program. If the line number in a **GO TO** command refers to a non-existent line, then the jump is to the next line after the given number. The same goes for **RUN**; in fact **RUN** on its own actually means **RUN 0**.

Now type in numbers until the display starts getting full. When it is full, the computer will move the whole of the top half of the display up one line to make room, losing the heading off the top. This is called scrolling.

When you are tired of this, stop the program using **STOP** (or **Esc**) and return to the listing in the editor.

Look at the **PRINT** statement on line 50. The punctuation in this - the comma (, ) - is very important, and you should remember that it follows much more definite rules than the punctuation in English.

Commas are used to make the printing start either at the left hand margin, or in the middle of the display, depending on which comes next. Thus in line 50, the comma causes the centigrade temperature to be printed in the middle of the line. With a semicolon, on the other hand, the next number or string is printed immediately after the preceding one. You can see this in line 50, if the comma is replaced by a semicolon.

Another punctuation mark you can use like this in **PRINT** commands is the apostrophe ('). This makes whatever is printed next appear at the beginning of the next line on the display but this happens anyway at the end of each **PRINT** command, so you will not need the apostrophe very much. This is why the **PRINT** command in line 50 always starts its printing on a new line, and it is also why the **PRINT** command in line 30 produces a blank line.

If you want to inhibit this, so that after one **PRINT** command the next one carries on on the same line, you can put a comma or semicolon at the end of the

first. To see how this works, replace line 50 in turn by each of

```
50 PRINT F,
50 PRINT F;
```

and

50 PRINT F

and run each version - for good measure you could also try

50 PRINT F'

The one with the comma spreads everything out in two columns, that with the semicolon crams everything together, that without either allows a line for each number and so does that with the apostrophe - the apostrophe gives a new line of its own, but inhibits the automatic one.

Remember the difference between commas and semicolons in **PRINT** commands; also, do not confuse them with the colons (:) that are used to separate commands in a single line.

Now type in these extra lines:

100	REM th	nis pol	ite	program	rememb
110	INPUT	n\$			
120	PRINT	"Hello	";r	ו\$;"!"	
130	G0 T0	110			

This is a separate program from the last one, but you can keep them both in the computer at the same time. To run the new one, type

RUN 100

Because this program inputs a string instead of a number, it prints out two string quotes - this is a reminder to you, and it usually saves you some typing as well. Try it once with any alias you care to make up for yourself.

Next time round, you will get two string quotes again. but you don't have to use them if you don't want to. Try this, for example. Rub them out (with → and **Backspace** twice), and type

n\$

Since there are no string quotes, the computer knows that it has to do some calculation: the calculation in this case is to find the value of the string variable called n, which is whatever name you happen to have typed in last time round. Of course, the INPUT statement acts like LET n, so the value of n is unchanged.

The next time round, for comparison, type

n\$

again, this time without rubbing out the string quotes. Now, just to confuse you, the variable n has the value "n\$".

If you want to use **STOP** for string input, you must first move the cursor back to the beginning of the line, using **←**.

Now look back at that RUN 100 we had earlier on. That just jumps to line 100, so couldn't we have said GO TO 100 instead? In this case, it so happens that the answer is yes; but there is a difference. RUN 100 first of all clears all the variables and the display, and after that works just like GO TO 100. GO TO 100 doesn't clear anything. There may well be occasions where you want to run a program without clearing any variables; here GO TO would be necessary and RUN could be disastrous, so it is better not to get into the habit of automatically typing RUN to run a program.

Another difference is that you can type **RUN** without a line number, and it starts

off at the first line in the program. **GO TO** must always have a line number.

Both these programs stopped because you typed **STOP** in the input line; sometimes - by mistake - you write a program that you can't stop and won't stop itself. Type

# 200 GO TO 200 RUN 200

This looks all set to go on for ever unless you pull the plug out; but there is a less drastic remedy. Press **Esc** (equivalent to the Break key on a real Spectrum). The program will stop, saying

### L BREAK into program

At the end of every statement, the program looks to see if this key is pressed; and if it is, then it stops. The **Esc** key can also be used when you are in the middle of printing, or using various other bits of machinery that you can attach to the computer - just in case the computer is waiting for them to do something but they're not doing it.

In these cases there is a different report,

#### D BREAK - CONT repeats

**CONTINUE** in this case (and in fact in most other cases too) repeats the statement where the program was stopped; but after the report L BREAK into program, CONTINUE carries straight on with the next statement after allowing for any jumps to be made.

Run the name program again and when it asks you for input type

**n\$** (after removing the quotes)

**n\$** is an undefined variable and you get an error report
# 2 Variable not found

If you now type

```
LET n$="something definite"
```

(which has its own report of **0 OK**, **0:1**) and

## CONTINUE

you will find that you can use **n\$** as input data without any trouble.

In this case **CONTINUE** does a jump to the **INPUT** command in line 110. It disregards the report from the **LET** statement because that said 'OK', and jumps to the command referred to in the previous report, the first command in line 110. This is intended to be useful. If a program stops over some error then you can do all sorts of things to fix it, and **CONTINUE** will still work afterwards.

As we said before, the report L BREAK into program is special, because after it **CONTINUE** does not repeat the command where the program stopped.

Type

### LIST (and Enter of course)

and when it asks **SCrOll?** (because it has filled up the display) press n for 'No'. The computer will give the report **D BREAK** - **CONT repeats** as though you had typed **Esc**. You might at some stage find out what happens if you press y instead of n; n, **Space** and **Esc** count as No, while everything else counts as Yes.

Now type

23 REM

and the edit cursor jumps to a blank line just after the newly inserted line 23; type

## 28 REM

and the edit cursor jumps to the new line 28. (In both cases, by typing a new line, you have caused the editor to move to that line in the listing.)

Experiment with moving the current line about by typing

line number **REM** 

Using the program full of **REM**s above, type

### LIST

and then n when it asks you **scroll?**. Now type

## CONTINUE

**CONTINUE** is a bit quirky here, because the bottom part of the display goes blank; but you can restore normality with **Esc**. The reason is that **LIST** was the first command in the line, so **CONTINUE** repeats this command. Unfortunately, the first command in the line is now **CONTINUE** itself so the computer just sits there doing **CONTINUE** over and over again until you stop it.

You can vary this by replacing **LIST** with

#### . LIST

2

(bear in mind that BASin splits lines at the ":" character) for which **CONTINUE** gives **0 OK** (because **CONTINUE** jumps to the second command in the line, which is taken to be its end) or

#### : LIST

for which **CONTINUE** gives N **Statement lost** (because **CONTINUE** jumps to the third command in the line, which no longer exists).

You have now seen the statements **PRINT**, **LET**, **INPUT**, **RUN**, **LIST**, **GO TO**, **CONTINUE**, **NEW** and **REM**, and they can all be used either as direct commands or in program lines - this is true of almost all commands in ZX Spectrum BASIC. **RUN**, **LIST**, **CONTINUE** and **NEW** are not usually of much use in a program, but they can be used.

Exercises

- 1. Put a **LIST** statement in a program, so that when you run it, it lists itself.
- 2. Write a program to input prices and print out the tax due (at 15 per cent). Put in **PRINT** statements so that the computer announces what it is going to do, and asks for the input price with extravagant politeness. Modify the program so that you can also input the tax rate (to allow for zero ratings or future changes).
- 3. Write a program to print a running total of numbers you input. (Suggestion: have two variables called total set to 0 to begin with and item. Input item, add it to total, print them both, and go round again.)
- 4. What would **CONTINUE** and **NEW** do in a program? Can you think of any uses at all for this?

← Chapter 1

```
<"> <="> "> "> =">
```

## **Chapter 3: Decisions**

Summary

IF, STOP, =, <, >, <=, >=, <>

All the programs we have seen so far have been pretty predictable - they went straight through the instructions, and then went back to the beginning again. This is not very useful. In practice the computer would be expected to make decisions and act accordingly. The instruction used has the form . . .

**IF** something is true, or not true **THEN** do something else.

For example, use **NEW** to clear the previous program from the computer and type in and run this program. (This is clearly meant for two people to play!)

```
10 REM Guess the number
20 INPUT a:
CLS
30 INPUT "Guess the number", b
40 IF b=a THEN PRINT "That is correct":
STOP
50 IF ba THEN
PRINT "That is too big, try again"
70 GO TO 30
```

You can see that an **IF** statement takes the form

IF condition THEN . . .

where the '. . .' stands for a sequence of commands, separated by colons in the usual way. The condition is something that is going to be worked out as either true or false: if it comes out as true then the statements in the rest of the line after

**THEN** are executed, but otherwise they are skipped over, and the program executes the next instruction.

The simplest conditions compare two numbers or two strings: they can test whether two numbers are equal or whether one is bigger than the other; and they can test whether two strings are equal, or (roughly) one comes before the other in alphabetical order. They use the relations =, <, >, <=, >= and <>.

= means 'equals'. Although it is the same symbol as the = in a LET command, it is used in quite a different sense.

< means 'is less than' so that

1 < 2 -2 < -1 -3 < 1

are all true, but

1 < 0 0 < **-**2

are false.

Line 40 compares a and b. If they are equal then the program is halted by the **STOP** command. The report **9 STOP statement**, **30:3** shows that the third statement, or command, in line 30 caused the program to halt, i.e. **STOP**.

Line 50 determines whether b is less than a, and line 60 whether b is greater than a. If one of these conditions is true then the appropriate comment is printed, and the program works its way to line 70 which tells the computer to go back to line 30 and start all over again.

The **CLS**, clear screen, command in line 20 was to stop the other person seeing what you put in.

So > means 'is greater than', and is just like < but the other way round. You can remember which is which, because the thin end points to the number that is

supposed to be smaller.

<= means 'is less than or equal to', so that it is like < except that it is true even if the two numbers are equal: thus 2<=2 is true, but 2<2 is false.

>= means 'is greater than or equal to' and is similarly like >.

<> means 'is not equal to', the opposite in meaning to =.

Mathematicians usually write <=, >= and <> as  $\leq$ ,  $\geq$  and  $\neq$ . They also write things like '2<3<4' to mean '2<3 and 3<4', but this is not possible in BASIC.

Note: in some other versions of BASIC, the **IF** statement can have the form

**IF** condition **THEN** line number

This means the same as

IF condition THEN GO TO line number

Exercises

- 1. Try this program:
  - 10 PRINT "x": STOP: PRINT "y"

When you run it, it will display X and stop with the report

9 STOP statement, 10:2

Now type

### CONTINUE

You might expect this to jump back to the **STOP** command - **CONTINUE** 

usually repeats the statement referred to in the report. However, here this would not be very useful, because the computer would just stop again without displaying Y. Therefore, things are arranged so that after report 9 **CONTINUE** jumps to the command after the **STOP** command - so in our example, after **CONTINUE**, the computer prints Y and reaches the end of the program.

← Chapter 2

Chapter 4 ➡

## **Chapter 4: Looping**

Summary

FOR, NEXT, TO, STEP

Suppose you want to input five numbers and add them together. One way (don't type this in unless you are feeling dutiful) is to write

```
10 LET total=0
20 INPUT a
30 LET total=total+a
40 INPUT a
50 LET total=total+a
60 INPUT a
70 LET total=total+a
80 INPUT a
90 LET total=total+a
100 INPUT a
110 LET total=total+a
120 PRINT total
```

```
This method is not good programming practice. It may be just about controllable for five numbers, but you can imagine how tedious a program like this to add ten numbers would be, and to add a hundred would be just impossible.
```

Much better is to set up a variable to count up to 5 and then stop the program, like this (which you should type in):

```
10 LET total=0
20 LET count=1
30 INPUT a
40 REM count=number of times that
50 LET total=total+a
60 LET count=count+1
70 IF count<=5 THEN
   GO TO 30
80 PRINT total</pre>
```

Notice how easy it would be to change line 70 so that this program adds ten numbers, or even a hundred.

This sort of counting is so useful that there are two special commands to make it easier: the **FOR** command and the **NEXT** command. They are always used together. Using these, the program you have just typed in does exactly the same as

```
10 LET total=0
```

```
20 FOR c=1 TO 5
```

```
30 INPUT a
```

```
40 REM c=number of times that a ha
```

```
50 LET total=total+a
```

```
60 NEXT c
```

80 PRINT total

(To get this program from the previous one, you just have to edit lines 20, 40, 60,

and 70.)

Note that we have changed **COUNT** to **C**. The counting variable - or control variable - of a **FOR** . . . **NEXT** loop must have a single letter for its name.

The effect of this program is that **C** runs through the values 1 (the initial value), 2, 3, 4 and 5 (the limit), and for each one, lines 30, 40 and 50 are executed. Then, when **C** has finished its five values, line 80 is executed.

An extra subtlety to this is that the control variable does not have to go up by 1 each time: you can change this 1 to anything you like by using a STEP part in the FOR command. The most general form for a FOR command is

FOR control variable = initial value TO limit STEP step

where the control variable is a single letter, and the initial value, limit and step are all things that the computer can calculate as numbers - like the actual numbers themselves, or sums, or the names of numeric variables. So, if you replace line 20 in the program by

### 20 FOR c=1 TO 5 STEP 3/2

then **C** will run through the values 1, 2.5 and 4. Notice that you don't have to restrict yourself to whole numbers, and also that the control value does not have to hit the limit exactly - it carries on looping as long as it is less than or equal to the limit.

Try this program, to print out the numbers from 1 to 10 in reverse order.

## 10 FOR n=10 TO 1 STEP -1 20 PRINT n 30 NEXT n

We said before that the program carries on looping as long as the control variable is less than or equal to the limit. If you work out what this would mean

in this case, you will see that it gives nonsense. The normal rule has to be modified: when the step is negative, the program carries on looping as long as the control variable is greater than or equal to the limit.

You must be careful if you are running two FOR . . . NEXT loops together, one inside the other. Try this program, which prints out the numbers for a complete set of six spot dominoes.

```
10 FOR m=0 TO 6
20 FOR n=0 TO m
30 PRINT m;":";n;" ";
40 NEXT n
50 PRINT
60 NEXT m
```

You can see that the n-loop is entirely inside the m-loop - they are properly nested. What must be avoided is having two FOR . . . NEXT loops that overlap without either being entirely inside the other, like this:

```
5 REM this program is wrong
10 FOR m=0 TO 6
20 FOR n=0 TO m
30 PRINT m;":";n;" ";
40 NEXT m
50 PRINT
60 NEXT n
```

Two FOR . . . NEXT loops must either be one inside the other, or be completely

separate.

Another thing to avoid is jumping into the middle of a FOR ... NEXT loop from the outside. The control variable is only set up properly when its FOR statement is executed, and if you miss this out the NEXT statement will confuse the computer. You will probably get an error report saying NEXT without FOR or Variable not found.

There is nothing whatever to stop you using **FOR** and **NEXT** in a direct command. For example, try:

```
FOR m=0 TO 10: PRINT m: NEXT m
```

You can sometimes use this as a (somewhat artificial) way of getting round the restriction that you cannot GO TO anywhere inside a command - because a command has no line number. For instance,

FOR m=0 TO 1 STEP 0: INPUT a: PRINT a: NEXT m

The step of zero here makes the command repeat itself forever.

This sort of thing is not really recommended, because if an error crops up then you have lost the command and will have to type it in again - and **CONTINUE** will not work.

Exercises

1. A control variable has not just a name and a value, like an ordinary variable, but also a limit, a step, and a reference to the statement after the corresponding FOR statement. Persuade yourself that when the FOR statement is executed all this information is available (using the initial value as the first value the variable takes), and also that this information is enough for the

**NEXT** statement to know by how much to increase the value, whether to jump back, and if so where to jump back to.

2. Run the third program above and then type

## PRINT C

Why is the answer 6, and not 5?

(Answer: the NEXT command in line 60 is executed five times, and each time 1 is added to C. The last time, C becomes 6; and then the NEXT command decides not to loop back, but to carry on, C being past its limit.)

What happens if you put **STEP 2** in line 20?

3. Change the third program so that instead of automatically adding five numbers, it asks you to input how many numbers you want adding. When you run this program, what happens if you input 0, meaning that you want no numbers adding? Why might you expect this to cause problems for the computer, even though it is clear what you mean? (The computer has to make a search for the command

**NEXT C**, which is not usually necessary.) In fact this has all been taken care of.

- 4. In line 10 of the fourth program above, change 10 to 100 and run the program. It will print the numbers from 100 to 79 on the screen, and then say scroll? at the bottom. This is to give you a chance to see the numbers that are about to be scrolled off the top. If you press N, Space or Esc, the program will stop with the report D BREAK CONT repeats. If you press any other key, then it will print another 22 lines and ask you again.
- 5. Delete line 30 from the fourth program. When you run the new curtailed program, it will print the first number and stop with the message **O OK**. If you type

## NEXT n

the program will go once round the loop, printing out the next number.

← Chapter 3

Chapter 5 ➡

## **Chapter 5: Subroutines**

Summary

#### GO SUB, RETURN

Sometimes different parts of the program will have rather similar jobs to do, and you will find yourself typing the same lines in two or more times; however this is not necessary. You can type the lines in once, in a form known as a subroutine, and then use, or call, them anywhere else in the program without having to type them in again.

To do this, you use the statements **GO SUB** (**GO** to **SUB**routine) and **RETURN**. This takes the form

#### GO SUB n

where n is the line number of the first line in the subroutine. It is just like GO TO n except that the computer remembers where the GO SUB statement was so that it can come back again after doing the subroutine. It does this by putting the line number and the statement number within the line (together these constitute the return address) on top of a pile of them (the GO SUB stack).

You can view the **GO SUB** stack in BASin.

#### RETURN

takes the top *return address* off the **GO SUB** stack, and goes to the statement after it.

As an example, let's look at the number guessing program again. Retype it as follows:

```
10 REM "A rearranged guessing game"
20 INPUT a: CLS
```

```
30 INPUT "Guess the number ",b
40 IF a=b THEN
    PRINT "Correct": STOP
50 IF a<b THEN
    GO SUB 100
60 IF a>b THEN
    GO SUB 100
70 GO TO 30
100 PRINT "Try again"
110 RETURN
```

The **GO TO** statement in line 70 is very important because otherwise the program

Here is another rather silly program illustrating the use of **GO SUB**.

100 LET x=10 110 GOSUB 500 120 PRINT s 130 LET x=x+4 140 GO SUB 500 150 PRINT s 160 LET x=x+2 170 GO SUB 500 180 PRINT s 190 STOP 500 LET s=0 510 FOR y=1 TO x 520 LET s=s+y 530 NEXT y 540 RETURN

When this program is run, see if you can work out what is happening. The subroutine starts at line 500.

A subroutine can happily call another, or even itself (a subroutine that calls itself is recursive), so don't be afraid of having several layers.

← Chapter 4

Chapter 6 ➡

## **Chapter 6**

Summary

### READ, DATA, RESTORE

In some previous programs we saw that information, or data, can be entered directly into the computer using the **INPUT** statement. Sometimes this can be very tedious, especially if a lot of the data is repeated every time the program is run. You can save a lot of time by using the **READ**, **DATA** and **RESTORE** commands. For example:

```
10 READ a,b,c
20 PRINT a,b,c
30 DATA 10,20,30
40 STOP
```

A **READ** statement consists of **READ** followed by a list of the names of variables, separated by commas. It works rather like an **INPUT** statement, except that instead of getting you to type in the values to give to the variables, the computer looks up the values in the DATA statement.

Each DATA statement is a list of expressions - numeric or string expressions separated by commas. You can put them anywhere you like in a program, because the computer ignores them except when it is doing a READ. You must imagine the expressions from all the DATA statements in the program as being put together to form one long list of expressions, the DATA list. The first time the computer goes to READ a value, it takes the first expression from the DATA list; the next time, it takes the second; and thus as it meets successive READ statements, it works its way through the DATA list. (If it tries to go past the end of the DATA list, then it gives an error.)

Note that it's a waste of time putting DATA statements in a direct command, because READ will not find them. DATA statements have to go in the program. Let's see how these fit together in the program you've just typed in. Line 10 tells the computer to read three pieces of data and give them the variables a, b and c. Line 20 then says **PRINT** these variables. The DATA statement in line 30 gives the values of a, b and c. Line 40 stops the program. To see the order in which things work change line 20 to:

20 PRINT b,c,a

The information in DATA can be part of a FOR . . . NEXT loop. Type in

```
10 FOR n=1 TO 6
20 READ D
30 DATA 2,4,6,8,10,12
40 PRINT D
50 NEXT n
60 STOP
```

When this program is **RUN** you can see the **READ** statement moving through the **DATA** list. **DATA** statements can also contain string variables. For example:

```
10 READ d$
20 PRINT "The date is",d$
30 DATA "June 1st, 1982"
40 STOP
```

This is the simple way of fetching expressions from the DATA list: start at the beginning and work through until you reach the end. However, you can make the

computer jump about in the DATA list, using the RESTORE statement. This has RESTORE, followed by a line number, and makes subsequent READ statements start getting their data from the first DATA statement at or after the given line number. (You can miss out the line number, in which case it is as though you had typed the line number of the first line in the program.)

Try this program:

- 10 READ a, b
- 20 PRINT a,b
- 30 RESTORE 10
- 40 READ x,y,z
- 50 PRINT x,y,z
- 60 DATA 1,2,3
- 70 STOP

In this program the data required by line 10 made a = 1 and b = 2. The **RESTORE 10** instruction reset the variables, and allowed x, y and z to be **READ** starting from the first number in the **DATA** statement. Rerun this program without line 30 and see what happens.

← Chapter 5

Chapter 7 →

## **Chapter 7: Expressions**

Summary

Operations: +, -, \*, / Expressions, scientific notation, variable names

You have already seen some of the ways in which BASin can calculate with numbers. It can perform the four arithmetic operations +, -, \* and / (remember that \* is used for multiplication, and / is used for division), and it can find the value of a variable, given its name.

The example:

### LET tax=sum\*15/100

gives just a hint of the very important fact that these calculations can be combined. Such a combination, like SUM\*15/100, is called an expression: so an expression is just a short-hand way of telling the computer to do several calculations, one after the other. In our example, the expression SUM\*15/100 means 'look up the value of the variable called "sum", multiply it by 15, and divide by 100'.

Here is a summary of the order in which BASin evaluates mathematical expressions (sometimes called *operator precedence*).

Multiplications and divisions are done first. They have higher priority than addition and subtraction. Relative to each other, multiplication and division have the same priority, which means that the multiplications and divisions are done in order from left to right. When they are dealt with, the additions and subtractions come next - these again have the same priority as each other, so we do them in order from left to right.

Although all you really need to know is whether one operation has a higher or lower priority than another, the computer does this by having a number between 1 and 16 to represent the priority of each operation: \* and / have priority 8, and

+ and - have priority 6.

This order of calculation is absolutely rigid, but you can circumvent it by using brackets: anything in brackets is evaluated first and then treated as a single number.

Expressions are useful because, whenever the computer is expecting a number from you, you can give it an expression instead and it will work out the answer. The exceptions to this rule are so few that they will be stated explicitly in every case.

You can add together as many strings (or string variables) as you like in a single expression, and if you want, you can even use brackets.

We really ought to tell you what you can and cannot use as the names of variables. As we have already said, the name of a string variable has to be a single letter followed by \$; and the name of the control variable of a FOR . . . NEXT loop must be a single letter; but the names of ordinary numeric variables are much freer. They can use any letters or digits as long as the first one is a letter. You can put spaces in as well to make it easier to read, but they won't count as part of the name. Also, it doesn't make any difference to the name whether you type it in capitals or lower case letters.

Here are some examples of the names of variables that are allowed:

x t42 this name is so long that I shall never be able now we are six nOWWeaReSiX

(These last two names are considered the same, and refer to the same variable.)

These are not allowed to be the names of variables:

**2001** [it begins with a digit]

3 bears [begins with a digit]

M\*A\*S\*H [\* is not a letter or a digit]

Fotherington-Thomas [- is not a letter or a digit]

Numerical expressions can be represented by a number and exponent. Try the following to prove the point:

PRINT 2.34e0

PRINT 2.34e1

PRINT 2.34e2

and so on up to

PRINT 2.34e15

You will see that after a while the computer also starts using scientific notation. This is because no more than fourteen characters can be used to write a number. Similarly, try

PRINT 2.34e-1

PRINT 2.34e-2

and so on.

**PRINT** gives only eight significant digits of a number. Try

### PRINT 4294967295, 4294967295-429e7

This proves that the computer can hold the digits of 4294967295, even though it is not prepared to display them all at once.

BASin uses floating point arithmetic, which means that it keeps separate the digits of a number (its mantissa) and the position of the point (the exponent. This is not always exact, even for whole numbers. Type

#### PRINT 1e10+1-1e10, 1e10-1e10+1

Numbers are held to about nine and a half digits' accuracy, so 1e10 is too big to be held exactly right. The inaccuracy (actually about 2) is more than 1, so the numbers 1e10 and 1e10+1 appear to the computer to be equal. For an even more peculiar example, type

#### PRINT 5e9+1-5e9

Here the inaccuracy in 5e9 is only about 1, and the 1 to be added on in fact gets rounded up to 2. The numbers 5e9+1 and 5e9+2 appear to the computer to be equal.

The largest integer (whole number) that can be held completely accurately is 1 less than 32 2's multiplied together (or 4,294,967,295).

The string "" with no characters at all is called the empty or null string. Remember that spaces are significant and an empty string is not the same as one containing nothing but spaces.

Try:

### PRINT "Have you finished "Finnegans Wake" yet?"

When you press **Enter**, you will get the flashing red cursor that shows there is a mistake somewhere in the line. When the computer finds the double quotes at the beginning of "Finnegans Wake", it imagines that these mark the end of the string "Have you finished", and it then can't work out what

'Finnegans Wake' means.

There is a special device to get over this: whenever you want to write a string quote symbol in the middle of a string, you must write it twice, like this:

## PRINT "Have you finished ""Finnegans Wake"" yet

As you can see from what is printed on the screen, each double quote is only really there once; you just have to type it twice to get it recognized.

Chapter 6

Chapter 8 ➡

## **Chapter 8: Strings**

Summary

Slicing, using **TO**. Note that this notation is not standard BASIC.

Given a string, a substring of it consists of some consecutive characters from it, taken in sequence. Thus "string" is a substring of "bigger string", but "b string" and "big reg" are not.

There is a notation called slicing for describing substrings, and this can be applied to arbitrary string expressions. The general form is

```
string expression (start TO finish)
```

so that, for instance,

"abcdef"(2 TO 5)="bcde"

If you omit the start, then 1 is assumed; if you omit the finish then the length of the string is assumed. Thus

```
"abcdef"( TO 5)="abcdef"(1 TO 5)="abcde"
```

```
"abcdef"(2 TO )="abcdef"(2 TO 6)="bcdef"
```

```
"abcdef"( TO )="abcdef"(1 TO 6)="abcdef"
```

(You can also write this last one as "abcdef"(), for what it's worth.)

A slightly different form misses out the **TO** and just has one number.

"abcdef"(3)="abcdef"(3 TO 3)="c"

Although normally both start and finish must refer to existing parts of the string,

this rule is overridden by another one: if the start is more than the finish, then the result is the empty string. So

"abcdef"(5 **TO** 7)

gives error **3** Subscript wrong because the string only contains 6 characters and 7 is too many, but

```
"abcdef"(8 TO 7)=""
```

and

"abcdef"(1 TO 0)=""

The start and finish must not be negative, or you get error B Integer out of range. This next program is a simple one illustrating some of these rules.

```
10 LET a$="abcdef"
20 FOR n=1 TO 6
30 PRINT a$(n TO 6)
40 NEXT n
50 STOP
```

Type **NEW** when this program has been run and enter the next program:

```
10 LET a$="ABLE WAS I"
20 FOR n=1 TO 10
30 PRINT a$(n TO 10),a$((10-n) TO
40 NEXT n
50 STOP
```

For string variables, we can not only extract substrings, but also assign to them. For instance, type

#### LET a\$="Hi there, I'm BASin"

and then

#### LET a\$(5 TO 8)="\*\*\*\*\*"

and

#### PRINT a\$

Notice how since the substring a\$(5 TO 8) is only 4 characters long, only the first four stars have been used. This is a characteristic of assigning to substrings: the substring has to be exactly the same length afterwards as it was before. To make sure this happens, the string that is being assigned to it is cut off on the right if it is too long, or filled out with spaces if it is too short - this is called Procrustean assignment after the inn-keeper Procrustes who used to make sure that his guests fitted the bed by either stretching them out on a rack or cutting their feet off.

If you now try

```
LET a$()="Hello there"
```

and

```
PRINT a$;"."
```

you will see that the same thing has happened again (this time with spaces put in) because a () counts as a substring.

### LET a\$="Hello there"

will do it properly.

Complicated string expressions will need brackets round them before they can be sliced. For example,

"abc"+"def"(1 TO 2)="abcde"

("abc"+"def")(1 TO 2)="ab"

Exercise

1. Try writing a program to print out the day of the week using string slicing. Hint: let the string be SunMonTuesWedThursFriSat.

← Chapter 7

Chapter 9 ➡

## **Chapter 9: Functions**

Summary

#### DEF FN, LEN, STR\$, VAL, SGN, ABS, INT, SQR, FN

Consider the sausage machine. You put a lump of meat in at one end, turn a handle, and out comes a sausage at the other end. A lump of pork gives a pork sausage, a lump of fish gives a fish sausage, and a lump of beef a beef sausage.

Functions are practically indistinguishable from sausage machines but there is a difference: they work on numbers and strings instead of meat. You supply one value (called the argument), mince it up by doing some calculations on it, and eventually get another value, the result.

Meat in  $\rightarrow$  Sausage Machine  $\rightarrow$  Sausage out Argument in  $\rightarrow$  Function  $\rightarrow$  Result out

Different arguments give different results, and if the argument is completely inappropriate the function will stop and give an error report.

Just as you can have different machines to make different products - one for sausages, another for dish cloths, and a third for fish-fingers and so on, different functions will do different calculations. Each will have its own name to distinguish it from the others.

You use a function in expressions by typing its name followed by the argument, and when the expression is evaluated the result of the function will be worked out.

As an example, there is a function called LEN, which works out the length of a string. Its argument is the string whose length you want to find, and its result is the length, so that if you type

#### PRINT LEN "Sinclair"

the computer will write the answer 8, the number of letters in 'Sinclair'.

If you mix functions and operations in a single expression, then the functions will be worked out before the operations. Again, however, you can circumvent this rule by using brackets. For instance, here are two expressions which differ only in the brackets, and yet the calculations are performed in an entirely different order in each case (although, as it happens, the end results are the same).

LEN "Fred"+ LEN	"Bloggs" L	-E
4+LEN "Bloggs"	L	-E
4+6	L	-E
10	1	LO

Here are some more functions.

**STR\$** converts numbers into strings: its argument is a number, and its result is the string that would appear on the screen if the number were displayed by a **PRINT** statement. Note how its name ends in a **\$** sign to show that its result is a string. For example, you could say

#### LET a\$=STR\$ 1e2

which would have exactly the same effect as typing

LET a\$="100"

Or you could say

PRINT LEN STR\$ 100.000

and get the answer 3, because STR\$100.000 = "100".

VAL is like STR\$ in reverse: it converts strings into numbers. For instance,

#### VAL "3.5"=3.5

In a sense, VAL is the reverse of STR\$, because if you take any number, apply STR\$ to it, and then apply VAL to it, you get back to the number you first thought of. However, if you take a string, apply VAL to it, and then apply STR\$ to it, you do not always get back to your original string.

VAL is an extremely powerful function, because the string which is its argument is not restricted to looking like a plain number - it can be any numeric expression. Thus, for instance,

VAL "2\*3"=6

or even,

VAL ("2"+"\*3") = 6

There are two processes at work here. In the first, the argument of VAL is evaluated as a string: the string expression "2"+"\*3" is evaluated to give the string "2\*3". Then, the string has its double quotes stripped off, and what is left is evaluated as a number: so 2\*3 is evaluated to give the number 6.

This can get pretty confusing if you don't keep your wits about you; for instance,

### PRINT VAL "VAL""VAL"""2"""""

(Remember that inside a string a string quote must be written twice. If you go down into further depths of strings, then you find that string quotes need to be quadrupled or even octupled.)

There is another function, rather similar to VAL, although probably less useful, called VAL\$. Its argument is still a string, but its result is also a string. To see how this works, recall how VAL goes in two steps: first its argument is evaluated as a string, then the string quotes stripped off this, and whatever is left is evaluated as a number. With VAL\$, the first step is the same, but after the string quotes have been stripped off in the second step, whatever is left is evaluated as

another string. Thus

## VAL\$ """Fruit punch""" = "Fruit punch"

(Notice how the string quotes proliferate again.) Do

LET a\$="99"

and print out all of the following:

VAL a\$ VAL "a\$" VAL """a\$""" VAL\$ a\$ VAL\$ "a\$" VAL\$ """a\$"""

Some of these will work, and some of them won't; try to explain all the answers. (Keep a cool head.)

**SGN** is the sign function (sometimes called signum). It is the first function you have seen that has nothing to do with strings, because both its argument and its result are numbers. The result is +1 if the argument is positive, 0 if the argument is zero, and -1 if the argument is negative.

**ABS** is another function whose argument and result are both numbers. It converts the argument into a positive number (which is the result) by forgetting the sign, so that for instance

ABS - 3.2 = ABS 3.2 = 3.2

**INT** stands for 'integer part' - an integer is a whole number, possibly negative. This function converts a fractional number into an integer by throwing away the

fractional part, so that for instance,

### INT 3.9 = 3

Be careful when you are applying it to negative numbers, because it always rounds down: thus, for instance,

#### INT - 3.9 = -4

**SQR** calculates the square root of a number - the result that, when multiplied by itself, gives the argument. For instance,

SQR 4 = 2 because 2\*2=4

SQR 0.25 = 0.5 because 0.5\*0.5=0.25

SQR 2 = 1.4142136 (approximately) because 1.4142136\*1.414

If you multiply any number (even a negative one) by itself, the answer is always positive. This means that negative numbers do not have square roots, so if you apply SQR to a negative argument you get an error report A Invalid argument.

You can also define functions of your own. Possible names for these are FN followed by a letter (if the result is a number) or FN followed by a letter followed by \$ (if the result is a string). These are much stricter about brackets: the argument must be enclosed in brackets.

You define a function by putting a **DEF FN** statement somewhere in the program. For instance, here is the definition of a function **FN S** whose result is the square of the argument:

10 DEF FN s(x)=x\*x: REM the square of x

The X in brackets is a name by which you wish to refer to the argument of the

function. You can use any single letter you like for this (or, if the argument is a string, a single letter followed by **\$**).

After the = sign comes the actual definition of the function. This can be any expression, and it can also refer to the argument using the name you've given it (in this case, X) as though it were an ordinary variable.

When you have entered this line, you can invoke the function just like one of the computer's own functions, by typing its name, FN S, followed by the argument. Remember that when you have defined a function yourself, the argument must be enclosed in brackets. Try it out a few times:

```
PRINT FN s(2)
```

```
PRINT FN s(3+4)
```

```
PRINT 1+1NT FN s (LEN "chicken"/2+3)
```

Once you have put the corresponding **DEF FN** statement into the program, you can use your own functions in expressions just as freely as you can use the computer's.

Note: in some dialects of BASIC you must even enclose the argument of one of the computer's functions in brackets. This is not the case in ZX Spectrum BASIC.

**INT** always rounds down. To round to the nearest integer, add .5 first - you could write your own function to do this.

20 DEF FN r(x)=INT (x+0.5): REM gives x rounded

You will then get, for instance,

FN 
$$r(2.9) = 3$$
 FN  $r(2.4)$   
FN  $r(-2.9) = -3$  FN  $r(-2.4)$ 

Compare these with the answers you get when you use INT instead of FN r. Type in and run the following:

```
10 LET x=0:
   LET y=0:
   LET a=10
20 DEF FN p(x,y)=a+x
30 DEF FN q()=a+x*y
40 PRINT FN p(2,3),FN q()
```

There are a lot of subtle points in this program.

First, a function is not restricted to just one argument: it can have more, or even none at all - but you must still always keep the brackets.

Second, it doesn't matter whereabouts in the program you put the DEF FN statements. After the computer has executed line 10, it simply skips over lines 20 and 30 to get to line 40. They do, however, have to be somewhere in the program. They can't be in a command.

Third, X and Y are both the names of variables in the program as a whole, and the names of arguments for the function FN p. FN p temporarily forgets about the variables called X and Y, but since it has no argument called a, it still remembers the variable a. Thus when FN p(2,3) is being evaluated, a has the value 10 because it is the variable, X has the value 2 because it is the first argument, and Y has the value 3 because it is the second argument. The result is then, 10+2\*3=16. When FN q() is being evaluated, on the other hand. there are no arguments, so a, X and Y all still refer to the variables and have values 10, 0 and 0 respectively. The answer in this case is 10+0\*0=10.

Now change line 20 to

20 DEF FN p(x,y)=FN q()

This time, FN p(2,3) will have the value 10 because FN q will still go
back to the variables x and y rather than using the arguments of FN p.

Some BASICs (not the ZX Spectrum BASIC) have functions called LEFT\$, RIGHT\$, MID\$ and TL\$.

LEFT\$ (a\$, n) gives the substring of a\$ consisting of the first n characters. RIGHT\$ (a\$, n) gives the substring of a\$ consisting of the characters from nth on.

MID\$ (a\$, n1, n2) gives the substring of a\$ consisting of n2 characters starting at the n1th.

TL\$ (a\$) gives the substring of a\$ consisting of all its characters except the first.

You can write some user-defined functions to do the same: e.g.

Check that these work with strings of length 0 or 1.

Note that our FN 1\$ has two arguments, one a number and the other a string. A function can have up to 26 numeric arguments (why 26?) and at the same time up to 26 string arguments.

Exercise

1. Use the function

FN s(x)= $x^*x$ 

to test **SQR**: you should find that

## FN s(SQR x)=x

if you substitute any positive number for X, and

## whether x is positive or negativ

← Chapter 8

Chapter 10 →

## **Chapter 10: Mathematical functions**

Summary

#### ↑, PI, EXP, LN, SIN, COS, TAN, ASN, ACS, ATN

This chapter deals with the mathematics that BASin can handle. Quite possibly you will never have to use any of this at all, so if you find it too heavy going, don't be afraid of skipping it. It covers the operation  $\uparrow$  (raising to a power), the functions EXP and LN, and the trigonometrical functions SIN, COS, TAN and their inverses ASN, ACS, and ATN.

 $\uparrow$  and EXP

You can raise one number to the power of another - that means 'multiply the first number by itself the second number of times'. This is normally shown by writing the second number just above and to the right of the first number; but obviously this would be difficult on a computer so we use the symbol  $\uparrow$  instead. For example, the powers of 2 are

2↑1=2	
2↑2=2*2=4	(2 squared)
2↑3=2*2*2=8	(2 cubed)
2↑4=2*2*2*2=16	(2 to the power four)

Thus at its most elementary level, 'a  $\uparrow$  b' means 'a multiplied by itself b times', but obviously this only makes sense if b is a positive whole number. To find a definition that works for other values of b, we consider the rule

 $a^{b+c} = a^{b*a^{c}}$ 

(Notice that we give  $\uparrow$  a higher priority than \* and / so that when there are several operations in one expression, the  $\uparrow$ s are evaluated before the \*s and /s.) You should not need much convincing that this works when b and c are both positive whole numbers; but if we decide that we want it to work even when they are not, then we find ourselves compelled to accept that

a^0 = 1 a^(-b) = 1/a^b

 $a^{(1/b)} =$ the bth root of a...

...which is to say, the number that you have to multiply by itself b times to get a and finally...

 $a^{b*c} = (a^{b})^{c}$ 

If you have never seen any of this before then don't try to remember it straight away; just remember that

 $a^{-1} = 1/a$ and

 $a^{(1/2)} = SQR a$ 

and maybe when you are familiar with these the rest will begin to make sense.

Experiment with all this by trying this program:

```
10 INPUT a,b,c
20 PRINT a^(b+c),a^b*a^c
30 GO TO 10
```

Of course, if the rule we gave earlier is true, then each time round the two numbers that the computer prints out will be equal. (Note - because of the way the computer works out  $\uparrow$ , the number on the left - a in this case - must never be

negative.)

A rather typical example of what this function can be used for is that of compound interest. Suppose you keep some of your money in a building society and they give 15% interest per year. Then after one year you will have not just the 100% that you had anyway, but also the 15% interest that the building society have given you, making altogether 115% of what you had originally. To put it another way, you have multiplied your sum of money by 1.15, and this is true however much you had there in the first place. After another year, the same happened that then will have again, SO vou will have 1.15\*1.15=1.15<sup>2</sup>=1.3225 times your original sum of money. In general, after y years, you will have  $1.15 \uparrow y$  times what you started out with.

If you try this command

FOR y=0 TO 198: PRINT y, 10\*1.15^y:NEXT y

you will see that even starting off from just £10, it all mounts up quite quickly, and what is more, it gets faster and faster as time goes on. (Although even so, you might still find that it doesn't keep up with inflation.)

This sort of behaviour, where after a fixed interval of time some quantity multiplies itself by a fixed proportion, is called exponential growth, and it is calculated by raising a fixed number to the power of the time.

Suppose you did this:

#### 10 DEF FN $a(x)=a^x$

Here, a is more or less fixed, by **LET** statements: its value will correspond to the interest rate, which changes only every so often.

There is a certain value for a that makes the function FN a look especially pretty to the trained eye of a mathematician: and this value is called e. BASin has a function called EXP defined by

EXP x=e^x

Unfortunately, e itself is not an especially pretty number: it is an infinite non-recurring decimal. You can see its first few decimal places by doing

#### PRINT EXP 1

because EXP  $1 = e \uparrow 1 = e$ . Of course, this is just an approximation. You can never write down e exactly.

#### LN

The inverse of an exponential function is a logarithmic function: the logarithm (to base a) of a number x is the power to which you have to raise a to get the number x, and it is written  $\log_a x$ . Thus by definition  $a \uparrow \log_a x = x$ ; and it is also true that  $\log (a \uparrow x) = x$ .

You may well already know how to use base 10 logarithms for doing multiplications; these are called common logarithms. BASin has a function LN which calculates logarithms to the base e; these are called natural logarithms. To calculate logarithms to any other base, you must divide the natural logarithm by the natural logarithm of the base:

 $\log_a x = LN \times / LN a$ 

PI

Given any circle, you can find its perimeter (the distance round its edge; often called its circumference) by multiplying its diameter (width) by a number called  $\pi$ .  $\pi$  is a Greek p, and it is used because it stands for perimeter. Its name is pi.)

Like e,  $\pi$  is an infinite non-recurring decimal; it starts off as 3.141592653589...

The word **PI** on the Spectrum is taken as standing for this number - try

# PRINT PI

SIN, COS and TAN; ASN, ACS and ATN

The trigonometrical functions measure what happens when a point moves round a circle. Here is a circle of radius 1 (1 what? It doesn't matter, as long as we keep to the same unit all the way through. There is nothing to stop you inventing a new unit of your own for every circle that you happen to be interested in) and a point moving round it. The point started at the 3 o'clock position, and then moved round in an anti-clockwise direction.



We have also drawn in two lines called axes through the centre of the circle. The one through 9 o'clock and 3 o'clock is called the x-axis, and the one through 6 o'clock and 12 o'clock is called the y-axis.

To specify where the point is, you say how far it has moved round the circle from its 3 o'clock starting position: let us call this distance a. We know that the circumference of the circle is  $2\pi$  (because its radius is 1 and its diameter is thus 2): so when it has moved a quarter of the way round the circle, a =  $\pi/2$ ; when it has moved halfway round, a =  $\pi$ ; and when it has moved the whole way round, a =  $2\pi$ .

Given the curved distance round the edge, a, two other distances you might like to know are how far the point is to the right of the y-axis, and how far it is above the x-axis. These are called, respectively, the cosine and sine of a. The functions **COS** and **SIN** on the computer will calculate these.



Note that if the point goes to the left of the y-axis, then the cosine becomes negative; and if the point goes below the x-axis, the sine becomes negative.

Another property is that once a has got up to  $2\pi$ , the point is back where it started and the sine and cosine start taking the same values all over again:

SIN (a+2\*PI) = SIN a

COS (a+2\*PI) = COS a

The tangent of a is defined to be the sine divided by the cosine; the corresponding function on the computer is called TAN.

Sometimes we need to work these functions out in reverse, finding the value of a that has given sine, cosine or tangent. The functions to do this are called arcsine (ASN on the computer), arccosine (ACS) and arctangent (ATN).

In the diagram of the point moving round the circle, look at the radius joining the centre to the point. You should be able to see that the distance we have called a, the distance that the point has moved round the edge of the circle, is a way of measuring the angle through which the radius has moved away from the x-axis. When a =  $\pi/2$ , the angle is 90 degrees; when a =  $\pi$  the angle is 180 degrees; and

so round to when a =  $2\pi$ , and the angle is 360 degrees. You might just as well forget about degrees, and measure the angle in terms of a alone: we say then that we are measuring the angle in radians. Thus  $\pi/2$  radians = 90 degrees and so on.

You must always remember that SIN, COS and so on use radians and not degrees. To convert degrees to radians, divide by 180 and multiply by  $\pi$ ; to convert back from radians to degrees, you divide by  $\pi$  and multiply by 180.

Chapter 9

Chapter 11 ➡

## **Chapter 11: Random numbers**

Summary

#### RANDOMIZE, RND

This chapter deals with the function **RND** and the keyword **RANDOMIZE**. They are both used in connection with random numbers, so you must be careful not to get them mixed up. On the original Spectrum keyboard, **RANDOMIZE** was abbreviated to **RAND**.

In some ways **RND** is like a function: it does calculations and produces a result. It is unusual in that it does not need an argument.

Each time you use it, its result is a new random number between 0 and 1. (Sometimes it can take the value 0, but never 1.)

Try

## 10 PRINT RND 20 GO TO 10

to see how the answer varies. Can you detect any pattern? You shouldn't be able to; 'random' means that there is no pattern.

Actually, **RND** is not truly random, because it follows a fixed sequence of 65536 numbers. However, these are so thoroughly jumbled up that there are at least no obvious patterns and we say that **RND** is pseudo-random.

**RND** gives a random number between 0 and 1, but you can easily get random numbers in other ranges. For instance, **5**\*RND is between 0 and 5, and **1.3+0.7**\*RND is between 1.3 and 2. To get whole numbers, use **INT** (remembering that **INT** always rounds down) as in **1+INT** (RND\*6), which

we shall use in a program to simulate dice. RND\*6 is in the range 0 to 6, but since it never actually reaches 6, INT (RND\*6) is 0, 1, 2, 3, 4 or 5.

Here is the program:

```
10 REM dice throwing program
20 CLS
30 FOR n=1 TO 2
40 PRINT 1+INT (RND*6);" ";
50 NEXT n
60 INPUT a$:
G0 TO 20
```

Press **Enter** each time you want to throw the dice.

The **RANDOMIZE** statement is used to make **RND** start off at a definite place in its sequence of numbers, as you can see with this program:

```
10 RANDOMIZE 1
20 FOR n=1 TO 5:
    PRINT RND ,:
    NEXT n
30 PRINT:
    GO TO 10
```

After each execution of **RANDOMIZE 1**, the **RND** sequence starts off again with 0.0022735596. You can use other numbers between 1 and 65535 in the **RANDOMIZE** statement to start the **RND** sequence off at different places.

If you had a program with RND in it and it also had some mistakes that you had not found, then it would help to use RANDOMIZE like this so that the program behaved the same way each time you ran it.

**RANDOMIZE** on its own (and **RANDOMIZE 0** has the same effect) is different, because it really does randomize **RND** - you can see this in the next program.

10 RANDOMIZE 20 PRINT RND: GO TO 10

The sequence you get here is not very random, because **RANDOMIZE** uses the time since the computer was switched on. Since this has gone up by the same amount each time **RANDOMIZE** is executed, the next **RND** does more or less the same. You would get better randomness by replacing **GO TO 10** by **GO TO 20**.

Note: Most dialects of BASIC use **RND** and **RANDOMIZE** to produce random numbers, but not all use them in the same way.

Here is a program to toss coins and count the numbers of heads and tails.

```
10 LET heads=0:
   LET tails=0
20 LET coin=INT (RND*2)
30 IF coin=3 THEN
   LET heads=heads+1
40 IF coin=1 THEN
   LET tails=tails+1
```

```
50 PRINT heads;",";tails,
60 IF tails<>6 THEN
PRINT heads/tails;
70 PRINT:
G0 T0 20
```

The ratio of heads to tails should become approximately 1 if you go on long enough, because in the long run you expect approximately equal numbers of heads and tails.

Exercises

1. Test this rule:

Suppose you choose a number between 1 and 872 and type

**RANDOMIZE** your number

Then the next value of **RND** will be

(75\*(your number+1)-1)/65536

2. (For mathematicians only.)

Let p be a (large) prime, and let a be a primitive root modulo p.

Then if  $b_i$  is the residue of  $a_i$  modulo  $p \ (1 \le b_i \le p\text{-}1$  ), the sequence

<u>b<sub>i</sub> - 1</u> p - 1

is a cyclical sequence of p-1 distinct numbers in the range 0 to 1 (excluding 1). By choosing a suitably, these can be made to look fairly random.

65537 is a Fermat prime,  $2^{16}+1$ . Because the multiplicative group of non-zero residues modulo 65537 has a power of 2 as its order, a residue is a primitive root if and only if it is not a quadratic residue. Use Gauss' law of quadratic reciprocity to show that 75 is a primitive root modulo 65537.

BASin uses p=65537 and a=75, and stores some  $b_i$ -1 in memory. RND entails replacing  $b_i$ -1 in memory by  $b_{i+1}$ -1, and yielding the result ( $b_{i+1}$ -1) (p-1). RANDOMIZE n (with  $1 \le n \le 65535$ ) makes  $b_i$  equal to n+1.

**RND** is approximately uniformly distributed over the range 0 to 1.

← Chapter 10

Chapter 12 →

## **Chapter 12: Arrays**

Summary

Arrays (the way the Spectrum handles string arrays is slightly non-standard). **DIM**...

Suppose you have a list of numbers, for instance the marks of ten people in a class. To store them in the computer you could set up a single variable for each person. but you would find them very awkward. You might decide to call the variables Bloggs 1, Bloggs 2, and so on up to Bloggs 10, but the program to set up these ten numbers would be rather long and boring to type in.

How much nicer it would be if you could type this:

```
5 REM this program will not work
10 FOR n=1 TO 10
20 READ Bloggs n
30 NEXT n
40 DATA 10,2,5,19,16,3,11,1,0,6
```

Well you can't.

However, there is a mechanism by which you can apply this idea, and it uses arrays. An array is a set of variables, its elements, all with the same name, and distinguished only by a number (the subscript) written in brackets after the name. In our example the name could be b (like control variables of FOR . . . NEXT loops, the name of an array must be a single letter), and the ten variables would then be b(1), b(2), and so on up to b(10).

The elements of an array are called subscripted variables, as opposed to the simple variables that you are already familiar with.

Before you can use an array, you must reserve some space for it inside the computer, and you do this using a **DIM** (for dimension) statement.

#### DIM b(10)

sets up an array called **b** with dimension 10 (i.e. there are 10 subscripted variables b(1),...,b(10)) and initializes the 10 values to 0. It also deletes any array called b that existed previously. (But not a simple variable. An array and a simple numerical variable with the same name can coexist, and there shouldn't be any confusion between them because the array variable always has a subscript).

The subscript can be an arbitrary numerical expression, so now you can write

```
10 FOR n=1 TO 10
20 READ b(n)
30 NEXT n
40 DATA 10,2,5,19,16,3,11,1,0,6
```

You can also set up arrays with more than one dimension. In a two dimensional array you need two numbers to specify one of the elements - rather like the line and column numbers to specify a character position on the television screen - so it has the form of a table. Alternatively, if you imagine the line and column numbers (two dimensions) as referring to a printed page, you could have an extra dimension for the page numbers. Of course, we are talking about numeric arrays; so the elements would not be printed characters as in a book, but numbers. Think of the elements of a three-dimensional array V as being specified by V (page number, line number, column number).

For example, to set up a two-dimensional array **C** with dimensions 3 and 6, you use a **DIM** statement

DIM c(3,6)

This then gives you **3\*6=18** subscripted variables

 $c(1,1), c(1,2), \dots, c(1,6)$  $c(2,1), c(2,2), \dots, c(2,6)$  $c(3,1), c(3,2), \dots, c(3,6)$ 

The same principle works for any number of dimensions.

Although you can have a number and an array with the same name, you cannot have two arrays with the same name, even if they have different numbers of dimensions.

There are also string arrays. The strings in an array differ from simple strings in that they are of fixed length and assignment to them is always Procrustean - chopped off or padded with spaces. Another way of thinking of them is as arrays (with one extra dimension) of single characters. The name of a string array is a single letter followed by \$, and a string array and a simple string variable cannot have the same name (unlike the case for numbers).

Suppose then, that you want an array **a\$** of five strings. You must decide how long these strings are to be - let us suppose that 10 characters each is long enough. You then say

**DIM** a\$(5, 10) - (type this in)

This sets up a 5\*10 array of characters, but you can also think of each row as being a string:

a\$(1)	=	a\$(1,1)	a\$(1,2)	•	•	-	a\$(1
a\$(2)	=	a\$(2,1)	a\$(2,2)	•		•	a\$(2
:			:				:
a\$(5)	=	a\$(5,1)	a\$(5,2)			•	a\$(5

If you give the same number of subscripts (two in this case) as there were dimensions in the **DIM** statement, then you get a single character; but if you miss the last one out, then you get a fixed length string. So, for instance,

a (2,7) is the 7th character in the string a (2); using the slicing notation, we could also write this as a (2)(7). Now type

#### LET a\$(2)="1234567890"

and

#### PRINT a\$(2), a\$(2,7)

For the last subscript (the one you can miss out), you can also have a slicer, so that for instance

a(2,4 TO 8) = a(2)(4 TO 8) = "45678"

Remember:

In a string array, all the strings have the same, fixed length. The **DIM** statement has an extra number (the last one) to specify this length. When you write down a subscripted variable for a string array, you can put in an extra number, or a slicer, to correspond with the extra number in the **DIM** statement. You can have string arrays with no dimensions. Type

#### DIM a\$(10)

and you will find that **a\$** behaves just like a string variable, except that it always has length 10, and assignment to it is always Procrustean.

Exercises

Use READ and DATA statements to set up an array m\$ of twelve strings in which m\$(n) is the name of the nth month. (Hint: the DIM statement will be DIM m\$(12,9). Test it by printing out all the m\$(n) (use a loop).

Type

# PRINT "now is the month of ";m\$(

What can you do about all those spaces?

← Chapter 11

Chapter 13 ➡

<"> <="> "> "> =">

## **Chapter 13: Conditions**

Summary

AND, OR, NOT

We saw in Chapter 3 how an **IF** statement takes the form

IF condition THEN . . .

The conditions there were the relations (=, <, >, <=, >= and <>), which compare two numbers or two strings. You can also combine several of these, using the logical operations, AND, OR and NOT.

One relation AND another relation is true whenever both relations are true, so you could have a line like:

# IF a\$="yes" AND x>0 THEN PRINT x

in which X only gets printed if a\$="yes" and X>0. The BASIC here is so close to English that it hardly seems worth spelling out the details. As in English, you can join lots of relations together with AND, and then the whole lot is true if all the individual relations are.

One relation **OR** another is true whenever at least one of the two relations is true. (Remember that it is still true if both the relations are true; this is something that English doesn't always imply.)

The **NOT** relationship turns things upside down. The **NOT** relation is true whenever the relation is false, and false whenever it is true!

Logical expressions can be made with relations and AND, OR and NOT, just as numerical expressions can be made with numbers and +, - and so on; you can

even put them in brackets if necessary. They have priorities in the same way as the usual operations +, -, \*, / and  $\uparrow$  do: **OR** has the lowest priority, then **AND**, then **NOT**, then the relations, and the usual operations.

**NOT** is really a function, with an argument and a result, but its priority is much lower than that of other functions. Therefore its argument does not need brackets unless it contains AND or OR (or both). NOT a=b means the same as NOT (a=b) (and the same as a<>b, of course).

<> is the negation of = in the sense that it is true if, and only if, = is false. In other words,

```
a<>b is the same as NOT a=b
and also
NOT a<>b is the same as a=b
```

Persuade yourself that >= and <= are the negations of < and > respectively: thus you can always get rid of **NOT** from in front of a relation by changing the relation.

Also,

```
NOT (a first logical expression AND a second)
is the same as
NOT (the first) OR NOT (the second)
```

and

```
NOT (a first logical expression OR a second)is the same asNOT (the first) AND NOT (the second).
```

Using this you can work NOTs through brackets until eventually they are all applied to relations, and then you can get rid of them. Logically speaking, NOT is unnecessary, although you might still find that using it makes a program clearer.

The following section is quite complicated, and can be skipped by the fainthearted!

Try

#### PRINT 1=2, 1<>2

which you might expect to give a syntax error. In fact, as far as the computer is concerned, there is no such thing as a logical value: instead it uses ordinary numbers, subject to a few rules.

- =, <, >, <=, >= and <> all give numeric results: 1 for true, and 0 for false. Thus the **PRINT** command above printed 0 for '1=2', which is false, and 1 for '1<>2', which is true.
- 2. In

**IF** condition **THEN** . . .

the condition can be actually any numeric expression. If its value is 0, then it counts as false, and any other value (including the value of 1 that a true relation gives) counts as true. Thus the **IF** statement means exactly the same as

IF condition <> 0 THEN...

3. AND, OR and NOT are also number-valued operations.

**x AND y** has the value **x** if **y** is true (non-zero), or 0 (false) if **y** is false (zero).

**X OR y** has the value 1 (true), if y is true (non-zero), or x, if y is false (zero).

**NOT** X has the value 0 (false), if x is true (non-zero), or 1 (true), if x is false (zero).

4. (Notice that 'true' means 'non-zero' when we're checking a given value, but it means '1' when we're producing a new one.)

Read through the chapter again in the light of this revelation, making sure that it all works.

In the expressions x AND y, x OR y and NOT x, x and y will usually take the values 0 and 1 for false and true. Work out the ten different combinations

(four for AND, four for OR and two for NOT) and check that they do what the chapter leads you to expect them to do.

Try this program:

```
10 INPUT a
20 INPUT b
30 PRINT (a AND a>=b)+(b AND a<b)
40 GO TO 10
```

Each time it prints the larger of the two numbers a and b. Convince yourself that you can think of x AND y as meaning x if y (else the result is 0) and of x OR y as meaning x unless y (in which case the result is 1).

An expression using AND or OR like this is called a conditional expression. An example using OR could be

#### LET total price=price less tax\*(1.15 OR v\$="zer

Notice how AND tends to go with addition (because its default value is 0), and OR tends to go with multiplication (because its default value is 1).

You can also make string valued conditional expressions, but only using AND.

x\$ AND y has the value x\$ if y is non-zero, or "" if y is zero, so it means x\$ if y (else the empty string).

Try this program, which inputs two strings and puts them in alphabetical order.

```
10 INPUT "type in two strings"'a$,b$
20 IF a$>b$ THEN
   LET c$=a$:
   LET a$=b$:
   LET b$=c$
30 PRINT a$;" ";("<" AND a$<b$)+("=" AND a$=b$)</pre>
```

#### 40 GO TO 10

Exercise

BASIC can sometimes work along different lines from English. Consider, for instance, the English clause 'If a doesn't equal b or c'. How would you write this in BASIC? The answer is not IF a<>b OR c nor is it IF a<>b OR a<>c.

← Chapter 12

Chapter 14 ➡

## **Chapter 14: The character set**

Summary

#### CODE, CHR\$, POKE, PEEK, USR, BIN

The letters, digits, punctuation marks and so on that can appear in strings are called characters, and they make up the alphabet, or character set, that BASin uses. Most of these characters are single symbols, but there are some more, called tokens, that represent whole words, such as **PRINT**, **STOP**, **<>** and so on.

There are 256 characters, and each one has a code between 0 and 255. There is a complete list of them in Appendix A. To convert between codes and characters, there are two functions, CODE and CHR\$.

**CODE** is applied to a string, and gives the code of the first character in the string (or 0 if the string is empty).

CHR\$ is applied to a number, and gives the single character string whose code is that number.

This program prints out the entire character set:

10 FOR a=32 TO 255: PRINT CHR\$ a;: NEXT a

At the top you can see a space, 15 symbols and punctuation marks, the ten digits, seven more symbols, the capital letters, six more symbols, the lower case letters and five more symbols. These are all (except £ and ©) taken from a widely-used set of characters known as ASCII (standing for American Standard Codes for Information Interchange); ASCII also assigns numeric codes to these characters, and these are the codes that BASin uses.

The rest of the characters are not part of ASCII, and are peculiar to the ZX

Spectrum. First amongst them are a space and 15 patterns of black and white blobs. These are called the graphics symbols and can be used for drawing pictures. You can enter these from the keyboard, using what is called graphics mode. If you press the graphics mode key (**Num Lock**, **Scroll Lock**, or **Alt Gr** - depending on your settings) then the keys for the digits 1 to 8 will start giving the graphics symbols: on their own they give the symbols drawn on the keys; and with either shift pressed they give the same symbol but inverted, i.e. black becomes white, and vice versa.

Regardless of shifts, digit 9 takes you out of graphics mode.

<u>Symbol</u>	<u>Code</u>	<u>How obtained</u>	<u>Symbol</u>	Code	<u>How obtained</u>
	128	<b>G</b> 8		143	G shift 8
	129	<b>G</b> 1		142	G shift 1
	130	<b>C</b> 2		141	G shift 2
	131	<b>G</b> 3		140	G shift 3
	132	<b>G</b> 4		139	G shift 4
	133	<b>G</b> 5		138	🕒 shift 5
	134	<b>G</b> 6		137	G shift 6
	135	<b>G</b> 7		136	G shift 7

Here are the sixteen graphics symbols:

After the graphics symbols, you will see what appears to be another copy of the alphabet from A to U. These are characters that you can redefine yourself, although when the machine is first switched on they are set as letters - they are called user-defined graphics. You can type these in from the keyboard by going into graphics mode, and then using the letter keys from A to U.

BASin provides a Graphic Editor that allows you to redefine characters visually. If, however, you want to define a new character in code, follow this recipe - it defines a character to show pi.

1. Work out what the character looks like. Each character has an 8×8 square of dots, each of which can show either the paper colour or the ink colour (see the introductory booklet). You'd draw a diagram something like this, with black squares for the ink colour:



We've left a 1 square margin round the edge because the other letters all have one (except for lower case letters with tails, where the tail goes right down to the bottom).

- 2. Work out which user-defined graphic is to show  $\pi$  let's say the one corresponding to P, so that if you press P in graphics mode you get  $\pi$ .
- 3. Store the new pattern. Each user-defined graphic has its pattern stored as eight numbers, one for each row. You can write each of these numbers as BIN followed by eight 0's or 1's 0 for paper, 1 for ink so

that the eight numbers for our  $\pi$  character are

- BIN 0000000
- BIN 0000000
- BIN 0000010
- BIN 00111100
- BIN 01010100
- BIN 00010100
- BIN 00010100

### BIN 0000000

(If you know about binary numbers, then it should help you to know that **BIN** is used to write a number in binary instead of the usual decimal.)

These eight numbers are stored in memory, in eight places, each of which has an address. The address of the first byte, or group of eight digits, is USR "P" (P because that is what we chose in (ii)), that of the second is USR "P"+1, and so on up to the eighth, which has address USR "P"+7.

USR here is a function to convert a string argument into the address of the first

byte in memory for the corresponding user-defined graphic. The string argument must be a single character which can be either the user-defined graphic itself or the corresponding letter (in upper or lower case). There is another use for USR, when its argument is a number, which will be dealt with in Chapter 26.

Even if you don't understand this, the following program will do it for you:

```
10 FOR n=0 TO 7
20 INPUT row:
        POKE USR "P"+n,row
30 NEXT n
```

It will stop for **INPUT** data eight times to allow you to type in the eight **BIN** numbers above - type them in the right order, starting with the top row.

The **POKE** statement stores a number directly in a memory location, bypassing the mechanisms normally used by the BASIC. The opposite of **POKE** is **PEEK**, and this allows us to look at the contents of a memory location although it does not actually alter the contents of that location. They will be dealt with properly in Chapter 24.

After the user-defined graphics come the tokens.

You will have noticed that we have not printed out the first 32 characters, with codes 0 to 31. These are control characters. They don't produce anything printable, but have some less tangible effect on the display, or they are used for controlling something other than the display and the computer prints ? to show that it doesn't understand them. They are described more fully in Appendix A.

Three that the display uses are those with codes 6, 8 and 13; on the whole, CHR\$ 8 is the only one you are likely to find useful.

CHR\$ 6 prints spaces in exactly the same way as a comma does in a PRINT statement; for instance

PRINT 1; CHR\$ 6;2

does the same as

PRINT 1,2

Obviously this is not a very clear way of using it. A more subtle way is to say

```
LET a$="1"+CHR$ 6+"2"
PRINT a$
```

CHR\$ 8 is 'backspace': it moves the print position back one place - try

## PRINT "1234"; CHR\$ 8;"5"

which prints up 1235.

CHR\$ 13 is 'newline': it moves the print position on to the beginning of the next line.

The display also uses those with codes 16 to 23; these are explained in Chapters 15 and 16. All the control characters are listed in Appendix A.

Using the codes for the characters we can extend the concept of 'alphabetical ordering' to cover strings containing any characters, not just letters. If instead of thinking in terms of the usual alphabet of 26 letters we use the extended alphabet of 256 characters, in the same order as their codes, then the principle is exactly the same. For instance, these strings are in their Spectrum alphabetical order. (Notice the rather odd feature that lower case letters come after all the capitals: so "a" comes after "Z"; also, spaces matter.)

## CHR\$ 3+"ZOOLOGICAL GARDENS" CHR\$ 8+"AARDVARK HUNTING" " AAAARGH!"

```
"(Parenthetical remark)"
"100"
"129.95 inc. VAT"
"AASVOGEL"
"Aardvark"
"PRINT"
"Zoo"
"[interpolation]"
"aardvark"
"aasvogel"
"zoo"
"zoology"
```

Here is the rule for finding out which order two strings come in. First, compare the first characters. If they are different, then one of them has its code less than the other, and the string it came from is the earlier (lesser) of the two strings. If they are the same, then go on to compare the next characters. If in this process one of the strings runs out before the other, then that string is the earlier, otherwise they must be equal.

The relations =, <, >, <=, >= and <> are used for strings as well as for numbers: < means 'comes before' and > means 'comes after', so that

## "AA man"<"AARDVARK" "AARDVARK">"AA man"

are both true.

<= and >= work the same way as they do for numbers, so that

```
"The same string"<="The same string"
```

is true, but

```
"The same string"<"The same string"
```

is false.

Experiment on all this using the program here, which inputs two strings and puts them in order.

```
10 INPUT "Type in two strings:", a$, b$
20 IF a$>b$ THEN
   LET c$=a$:
   LET a$=b$:
   LET b$=c$
30 PRINT a$;" ";
40 IF a$<b$ THEN
   PRINT "<";:
   GO TO 60
50 PRINT "="
60 PRINT " ";b$
70 GO TO 10</pre>
```

Note how we have to introduce c\$ in line 20 when we swap over a\$ and b\$.

LET a\$=b\$: LET b\$=a\$

would not have the desired effect.

This program sets up user-defined graphics to show chess pieces: P for pawn, R for rook, N for knight, B for bishop, K for king, and Q for queen.

```
5 LET b=BIN 01111100:
    LET C=BIN 00111000:
    LET d=BIN 00010000
 10 FOR n=1 TO 6:
    READ p$:
    REM 6 pieces
 20 FOR f=0 TO 7:
    REM read piece into 8 bytes
 30 READ a:
    POKE USR p$+f,a
 40 NEXT f
 50 NEXT n
100 REM bishop
110 DATA "b",0,d, BIN 00101000,BIN
120 DATA BIN 01101100, c, b, 0
130 REM king
140 DATA "k",0,d,c,d
150 DATA c, BIN 01000100,c,0
160 REM rook
170 DATA "r",0, BIN 01010100,b,c
180 DATA c,b,b,0
190 REM queen
200 DATA "q",0, BIN 01010100, BIN
```

```
210 DATA BIN 01101100,b,b,0
220 REM pawn
230 DATA "p",0,0,d,c
240 DATA c,d,b,0
250 REM knight
260 DATA "n",0,d,c, BIN 01111000
270 DATA BIN 00011000,c,b,0
```

Note that 0 can be used instead of **BIN 0000000**.

When you have run the program, look at the pieces by going into graphics mode.

Exercises

- 1. Imagine the space for one symbol divided up into four quarters like a Battenburg cake. Then if each quarter can be either black or white, there are 2×2×2×2=16 possibilities. Find them all in the character set.
- 2. Run this program:
  - 10 INPUT a
  - 20 PRINT CHR\$ a;
  - 30 GO TO 10

If you experiment with it, you'll find that CHR\$ a is rounded to the nearest whole number; and if a is not in the range 0 to 255 then the program stops with error report:

## B integer out of range

3. Which of these two is the lesser?

"EVIL"

"evil"

4. Work out how to modify the program to set up userdefined graphics so that it uses **READ** and **DATA** statements instead of the **INPUT** statement.

← Chapter 13

Chapter 15 ➡
## **Chapter 15: More about PRINT and INPUT**

Summary

**CLS PRINT** items: nothing at all

Expressions (numeric or string type): TAB numeric expression, AT numeric expression, numeric expression
PRINT separators: , ; ', INPUT items: variables (numeric or string type)
LINE string variable, Any PRINT item not beginning with a letter. (Tokens are not considered as beginning with a letter.)
Scrolling, SCREEN\$

You have already seen **PRINT** used quite a lot, so you will have a rough idea of how it is used. Expressions whose values are printed are called **PRINT** items, and they are separated by commas or semicolons, which are called **PRINT** separators. A **PRINT** item can also be nothing at all, which is a way of explaining what happens when you use two commas in a row.

There are two more kinds of **PRINT** items, which are used to tell the computer not what, but where to print. For example **PRINT AT 11**, **16**; "\*" prints a star in the middle of the screen.

AT line, column

moves the **PRINT** position (the place where the next item is to be printed) to the line and column specified. Lines are numbered from 0 (at the top) to 21, and columns from 0 (on the left) to 31.

**SCREEN\$** is the reverse function to **PRINT AT**, and will tell you (within limits) what character is at a particular position on the screen. It uses line and column numbers in the same way as **PRINT AT**, but enclosed in brackets: for instance

### PRINT SCREEN\$ (11,16)

will retrieve the star you printed in the paragraph above.

Characters taken from tokens print normally, as single characters, and spaces return as spaces. Lines drawn by PLOT, DRAW or CIRCLE, user-defined characters and graphics characters return as a null (empty) string, however. The same applies if OVER has been used to create a composite character.

TAB column

prints enough spaces to move the **PRINT** position to the column specified. It stays on the same line, or, if this would involve backspacing, moves on to the next one. Note that the computer reduces the column number 'modulo 32' (it divides by 32 and takes the remainder); so **TAB** 33 means the same as **TAB** 1.

As an example,

```
PRINT TAB 30;1;TAB 12;"Contents"; AT 3,1;"CHAPT
```

is how you might print out the heading of a contents page on page 1 of a book.

Try running this:

```
10 FOR n=8 TO 23
20 PRINT TAB 8*n;n;
30 NEXT n
```

This shows what is meant by the **TAB** numbers being reduced modulo 32. For a more elegant example, change the 8 in line 20 to a 6.

Some small points:

1. These new items are best terminated with semicolons, as we have done above. You can use commas (or nothing, at the end of the statement), but this means

that after having carefully set up the **PRINT** position you immediately move it on again: not usually terribly useful.

- You cannot print on the bottom two lines (22 and 23) on the screen because they are reserved for commands, INPUT data, reports and so on. References to 'the bottom line' usually mean line 21.
- 3. You can use **AT** to put the **PRINT** position even where there is already something printed; the old stuff will be obliterated when you print more.

Another statement connected with **PRINT** is **CLS**. This clears the whole screen, something that is also done by **CLEAR** and **RUN**.

When the printing reaches the bottom of the screen, it starts to scroll upwards rather like a typewriter. You can see this if you do

```
CLS:
FOR n=1 TO 30:
PRINT n:
NEXT n
```

and then do

### PRINT 99

a few times.

If the computer is printing out reams and reams of stuff, then it takes great care to make sure that nothing is scrolled off the top of the screen until you have had a chance to look at it properly. You can see this happening if you type CLS: FOR n=1 TO 100: PRINT n: NEXT n

When it has printed a screen full, it will stop, writing SCrOll? at the bottom of the screen. You can now inspect the first 22 numbers at your leisure. When you have finished with them, press Y (for 'yes') and the computer will give you another screen full of numbers. Actually, any key will make the computer carry on except N (for 'no'), **Space** or **Esc**. These will make the computer stop running the program with a report D BREAK - CONT repeats.

The **INPUT** statement can do much more than we have told you so far. You have already seen **INPUT** statements like

```
INPUT "How old are you?", age
```

in which the computer prints the caption **How old are you?** at the bottom of the screen, and then you have to type in your age.

In fact, an **INPUT** statement is made up of items and separators in exactly the same way as a **PRINT** statement is, so **HOW old are you?** and **age** are both **INPUT** items. **INPUT** items are generally the same as **PRINT** items, but there are some very important differences.

First, an obvious extra **INPUT** item is the variable whose value you are to type in age in our example above. The rule is that if an **INPUT** item begins with a letter, it must be a variable whose value is to be input.

Second, this would seem to mean that you can't print out the values of variables as part of a caption; however, you can get round this by putting brackets round the variable. Any expression that starts with a letter must be enclosed in brackets if it is to be printed as part of a caption.

Any kind of **PRINT** item that is not affected by these rules is also an **INPUT** item. Here is an example to illustrate what's going on:

LET my age = INT (RND \* 100): INPUT ("I am ";my age;".");"How old are you?",

**my age** is contained in brackets, so its value gets printed out. **your age** is not contained in brackets, so you have to type its value in.

Everything that an **INPUT** statement writes goes to the bottom part of the screen, which acts somewhat independently of the top half. In particular, its lines are numbered relative to the top line of the bottom half, even if this has moved up the actual television screen (which it does if you type lots and lots of **INPUT** data).

To see how **AT** works in **INPUT** statements, try running this:

```
10 INPUT "This is line 1.",a$; AT 0,0;"This is
AT 2,0;"This is line 2.",a$;
AT 1,0;"This is still line 1.",a$
```

(just press **Enter** each time it stops.) When **This is line 2.** is printed, the lower part of the screen moves up to make room for it; but the numbering moves up as well, so that the lines of text keep their same numbers.

Now try this:

As the lower part of the screen goes up and up, the upper part is undisturbed until the lower part threatens to write on the same line as the **PRINT** position. Then the upper part starts scrolling up to avoid this.

Another refinement to the **INPUT** statement that we haven't seen yet is called **LINE** input and is a different way of inputting string variables. If you write **LINE** before the name of a string variable to be input, as in

### INPUT LINE a\$

then the computer will not give you the string quotes that it normally does for a string variable, although it will pretend to itself that they are there. So if you type in

### cat

as the **INPUT** data, **a\$** will be given the value **Cat**. Because the string quotes do not appear on the string, you cannot delete them and type in a different sort of string expression for the **INPUT** data. Remember that you cannot use **LINE** for numeric variables.

The control characters CHR\$ 22 and CHR\$ 23 have effects rather like AT and TAB. They are rather odd as control characters, because whenever one is sent to the television to be printed, it must be followed by two more characters that do not have their usual effect: they are treated as numbers (their codes) to specify the line and column (for AT) or the tab position (for TAB). You will almost always find it easier to use AT and TAB in the usual way rather than the control characters, but they might be useful in some circumstances. The AT control character is CHR\$ 22. The first character after it specifies the line number and the second the column number, sothat

### PRINT CHR\$ 22+CHR\$ 1+CHR\$ c;

has exactly the same effect as

### PRINT AT 1,C;

This is so even if CHR\$ 1 or CHR\$ C would normally have a different meaning (for instance if C=13); the CHR\$ 22 before them overrides that.

The TAB control character is CHR\$ 23 and the two characters after it are used to give a number between 0 and 65535 specifying the number you would have in a TAB item:

PRINT CHR\$ 23+CHR\$ a+CHR\$ b;

has the same effect as

```
PRINT TAB a+256*b;
```

You can use **POKE** to stop the computer asking you **SCrOll?** by doing

POKE 23692,255

every so often. After this it will scroll up 255 times before stopping with scroll? As an example, try

```
10 FOR n=0 TO 10000
20 PRINT n:
POKE 23692,255
30 NEXT n
```

and watch everything whizz off the screen!

Exercises

1. Try this program on some children, to test their multiplication tables.

10 LET m\$=" " 20 LET a=INT (RND\*12)+1:

```
LET b=INT (RND*12)+1
```

```
30 INPUT (m$);" What is ";(a);"
```

100 IF c=a-b THEN

```
LET m$="Right.": GO TO 20
```

110 LET m\$="Wrong. Try again.": GO TO 30

If they are perceptive, they might manage to work out that they do not have to do the calculation themselves. For instance, if the computer asks them to type the answer to 2\*3, all they have to type in is 2\*3.

One way of getting round this is to make them input strings instead of numbers. Replace **C** in line 30 by **C\$**, and in line 100 by **VAL C\$**, and insert a line

40 IF c\$ <> STR\$ VAL c\$ THEN LET m\$="Type it properly, as GO TO 30

That will fool them. After a few more days, however, one of them may discover that they can get round this by rubbing out the string quotes and typing in STR\$ (2\*3). To stop up this loophole, you can replace c\$ in line 30 by LINE c\$.

← Chapter 14

Chapter 16 →

# **Chapter 16: Colours**

Summary

INK, PAPER, FLASH, BRIGHT, INVERSE, OVER, BORDER

Run this program:

10	FOR m=4 TO 1:
	BRIGHT m
20	FOR n=1 TO 14
30	FOR c=4 T0 7
40	PAPER c:
	PRINT " ;:
	REM 4 coloured spaces
50	NEXT c:
	NEXT n:
	NEXT m
60	FOR m=0 TO 1:
	BRIGHT m:
	PAPER 7
70	FOR c=0 TO 3
80	INK c:
	PRINT c;" ";
90	NEXT c:

```
PAPER 0
100 FOR c=4 TO 7
110 INK c:
PRINT c;" ";
120 NEXT c:
NEXT m
130 PAPER 7:
INK 0:
BRIGHT 0
```

This shows the eight colours (including white and black) and the two levels of brightness that BASin can produce. Here is a list of them for reference.

0	- black
1	- blue
2	- red
3	- purple, or magenta
4	- green
5	- pale blue, technically called cyan
6	- yellow
7	- white

On a black and white television, these numbers are in order of brightness.

To use these colours properly, you need to understand a bit about how the picture is arranged.

The picture is divided up into 768 (24 lines of 32) positions where characters can be printed, and each character is printed as an 8×8 square of dots like that below for a. This should remind you of the user-defined graphics in Chapter 14, where we had 0s for the white dots and 1s for the black dots.



The character position also has associated with it two colours: the ink, or foreground colour, which is the colour for the black dots in our square, and the paper, or background colour, which is used for the white dots. To start off with, every position has black ink and white paper so writing appears as black on white.

The character position also has a brightness (normal or extra bright) and something to say whether it flashes or not - flashing is done by swapping the ink and paper colours. This can all be coded into numbers, so a character position then has

- 1. an 8×8 square of 0s and 1 s to define the shape of the character, with 0 for paper and 1 for ink,
- 2. ink and paper colours, each coded into a number between 0 and 7,
- 3. a brightness 0 for normal, 1 for extra bright and
- 4. a flash number 0 for steady, 1 for flashing.

Note that since the ink and paper colours cover a whole character position, you cannot possibly have more than two colours in a given block of 64 dots. The same goes for the brightness and flash number: they refer to the whole character position, not individual dots. The colours, brightness and flash number at a given position are called attributes.

When you print something on the screen, you change the dot pattern at that position; it is less obvious, but still true, that you also change the attributes at that position. To start off with you do not notice this because everything is printed with black ink on white paper (and normal brightness and no flashing), but you can vary this with the INK, PAPER, BRIGHT and FLASH statements. Try

### PAPER 5

and then print a few things: they will all appear on cyan paper, because as they are printed the paper colours at the positions they occupy are set to cyan (which has code 5).

The others work the same way, so after

```
PAPER number between 0 and 7
INK number between 0 and 7
BRIGHT 0 or 1 (Think of 0 as off and 1 as on)
or
FLASH 0 or 1
```

any printing will set the corresponding attribute at all the character positions it uses. Try some of these out. You should now be able to see how the program at the beginning worked (remember that a space is a character that has **INK** and **PAPER** the same colour).

There are some more numbers you can use in these statements that have less direct effects.

8 can be used in all four statements, and means 'transparent' in the sense that the old attribute shows through. Suppose, for instance, that you do

### PAPER 8

No character position will ever have its paper colour set to 8 because there is no such colour; what happens is that when a position is printed on, its paper colour is left the same as it was before. **INK 8, BRIGHT 8** and **FLASH 8** work the same way for the other attributes.

9 can be used only with PAPER and INK, and means 'contrast'. The colour (ink or paper) that you use it with is made to contrast with the other by being made white if the other is a dark colour (black, blue, red or magenta), and black if the other is a light colour (green, cyan, yellow or white).

Try this by doing

```
INK 9:
FOR c=0 TO 7:
PAPER c:
PRINT c:
NEXT c
```

A more impressive display of its power is to run the program at the beginning to make coloured stripes, and then doing

```
INK 9:
PAPER 8:
PRINT AT 0,8;:
FOR n=1 TO 1080:
PRINT n;:
NEXT n
```

The ink colour here is always made to contrast with the old paper colour at each position.

Colour television relies on the rather curious fact that the human eye can only really see three colours - the primary colours, blue, red and green. The other colours are mixtures of these. For instance, magenta is made by mixing blue with red - which is why its code, 3, is the sum of the codes for blue and red.

To see how all eight colours fit together, imagine three rectangular spotlights, coloured blue, red and green, shining at not quite the same place on a piece of white paper in the dark. Where they overlap you will see mixtures of colours, as shown by this program (note that ink spaces are obtained by using either **Shift** with 8 when in graphics mode):

```
10 BORDER 0:
     PAPER 0:
     INK 7:
     CLS
  20 FOR a=1 TO 6
  30 PRINT TAB 6; INK 1; "nnnnnnn
     REM 18 ink squares
  40 NEXT a
  50 LET dataline=20
  60 GO SUB 1000
  70 LET dataline=210
  80 GO SUB 1000
  90 STOP
 200 DATA 2,3,7,5,4
 210 DATA 2,2,6,4,4
1000 FOR a=1 TO 6
1010 RESTORE dataline
1020 FOR b=1 TO 5
```

```
1030 READ c:

PRINT INK c;"nnnnnn";:

REM 6 ink squares

1040 NEXT b:

PRINT:

NEXT a

1050 RETURN
```

There is a function called **ATTR** that finds out what the attributes are at a given position on the screen. It is a fairly complicated function, so it has been relegated to the end of this chapter.

There are two more statements, **INVERSE** and **OVER**, which control not the attributes, but the dot pattern that is printed on the screen. They use the numbers 0 for off and 1 for on in the same way as **FLASH** and **BRIGHT** do, but those are the only possibilities. If you do **INVERSE 1**, then the dot patterns printed will be the inverse of their usual form: paper dots will be replaced by ink dots and vice versa. Thus a would be printed as



If (as at switch-on) we have black ink and white paper, then this **a** will appear as white on black - but we still have black ink and white paper at that character position. It is the dots that have changed.

The statement

### OVER 1

sets into action a particular sort of overprinting. Normally when something is written into a character position it completely obliterates what was there before; but now the new character will simply be added in on top of the old one (but see Exercise 1). This can be particularly useful for writing composite characters, like letters with accents on them, as in this program to print out German letters - an 'o' with an umlaut above it. (Do NEW first.)

```
10 OVER 1
20 FOR n=1 TO 32
30 PRINT "o"; CHR$ 8;"""";
40 NEXT n
```

(notice the control character CHR\$ 8 which backs up one space.)

There is another way of using **INK**, **PAPER** and so on which you will probably find more useful than having them as statements. You can put them as items in a **PRINT** statement (followed by ;), and they then do exactly the same as they would have done if they had been used as statements on their own, except that their effect is only temporary: it lasts as far as the end of the **PRINT** statement that contains them. Thus if you type

PRINT PAPER 6;"x";:
PRINT "y"

then only the X will be on yellow.

**INK** and the rest when used as statements do not affect the colours of the lower part of the screen, where commands and **INPUT** data are typed in. The lower part of the screen uses the colour of the border for its paper colour and code 9 for contrast for its **INK** colour, has flashing off, and everything at normal brightness. You can change the border colour to any of the eight normal colours

(not 8 or 9) using the statement

#### BORDER colour

When you type in **INPUT** data, it follows this rule of using contrasting ink on border coloured paper; but you can change the colour of the captions written by the computer by using **INK** and **PAPER** (and so on) items in the **INPUT** statement, just as you would in a **PRINT** statement. Their effect lasts either to the end of the statement, or until some **INPUT** data is typed in, whichever comes first. Try

### INPUT FLASH 1; INK 1;"What is your number?";n

There is one more way of changing the colours by using control characters - rather like the control characters for AT and TAB in Chapter 15.

- CHR\$ 16 corresponds to INK
- CHR\$ 17 corresponds to PAPER
- CHR\$ 18 corresponds to FLASH
- CHR\$ 19 corresponds to BRIGHT
- CHR\$ 20 corresponds to INVERSE
- CHR\$ 21 corresponds to OVER

These are each followed by one character that shows a colour by its code: so (for instance)

#### PRINT CHR\$ 16+CHR\$ 9; . . .

has the same effect as

#### PRINT INK 9; . . .

On the whole, you would not bother to use these control characters because you might just as well use the colour items. However, one very useful thing you can do with them is put them in programs: this results in different parts being listed in different colours, to set them apart from each other or even just to look pretty.

You must put them in after the line number, or they will just get lost.

To get these into the program, you have to enter them from the keyboard. Note that control characters tend to be less useful in BASin than on a real Spectrum, because BASin already uses colour for syntax highlighting.

The **ATTR** function has the form

```
ATTR (line, column)
```

Its two arguments are the line and column numbers that you would use in an AT item, and its result is a number that shows the colours and so on at the corresponding character position on the television screen. You can use this as freely in expressions as you can any other function.

The number that is the result is the sum of four other numbers as follows:

128 if the character position is flashing, 0 if it is steady64 if the character position is bright, 0 if it is normal8 \* the code for the paper colourthe code for the ink colour

For instance, if the character position is flashing and normal with yellow paper and blue ink then the four numbers that we have to add together are 128, 0, 8\*6=48 and 1, making 177 altogether. Test this with

PRINT AT 0,0; FLASH 1; PAPER 6; INK 1;" "; ATTR

Exercises

1. Try

# PRINT "B"; CHR\$ 8; OVER 1;"/";

Where the / has cut through the B, it has left a white

dot. This is the way overprinting works: two papers or two inks give a paper, one of each gives an ink. This has the interesting property that if you overprint with the same thing twice you get back what you started off with. If you now type

# PRINT CHR\$ 8; OVER 1;"/"

why do you recover an unblemished B?

2. Type

```
BORDER 0:
PAPER 0:
INK 0:
CLS
```

and notice that the '0 OK' report is printed in a contrasting colour to make it readable. Isn't it just as well that **INK** and **PAPER** don't affect the lower part of the screen?

3. Run this program:

# 10 POKE 22527+RND\*704, RND\*127 20 GO TO 10

Never mind how this works; it is changing the colours of squares on the television screen and the RNDs should ensure that this happens randomly. The diagonal stripes that you eventually see are a manifestation of the hidden pattern in RND - the pattern that makes it pseudorandom instead of truly random.

4. Type or **LOAD** in the chess piece characters in Chapter 14, and then type in this program which draws a diagram of a chess position using them.

```
5 REM draw blank board
10 LET bb=1:
  LET bw=2:
  REM red and blue for board
15 PAPER bw:
  INK bb:
  CLS
20 PLOT 79,128:
  REM border
```

```
30 DRAW 65,0:
    DRAW 0,-65
 40 DRAW -65,0:
    DRAW 0,65
 50 PAPER bb
 60 REM board
 70 FOR n=0 TO 3:
    FOR m=0 TO 3
 80 PRINT AT 6+2*n, 11+2*m;"
                              11
 90 PRINT AT 7+2*n, 10+2*m;"
                              11
100 NEXT m:
    NEXT n
110 PAPER 8
120 LET pw=6:
    LET pb=5:
    REM colours of white and bla
200 DIM b$ (8,8):
    REM positions of pieces
205 REM set up initial positions
210 LET b$(1)="rnbqkbnr"
220 LET b$(2)="ppppppp"
230 LET b(7) = "PPPPPPPP"
240 LET b$(8)="RNBQKBNR"
```

```
300 REM display board
310 FOR n=1 TO 8:
    FOR m=1 TO 8
320 LET bc=CODE b(n,m):
    INK pw
325 IF bc=CODE " " THEN
    GO TO 350:
    REM space
330 IF bc>CODE "Z" THEN
    INK pb:
    LET bc=bc-32:
    REM lower case for black
340 LET bc=bc+79:
    REM convert to graphics
350 PRINT AT 5+n, 9+m; CHR$ bc
360 NEXT m:
    NEXT n
400 PAPER 7:
    INK O
```

← Chapter 15

Chapter 17 →

## **Chapter 17: Graphics**

Summary

### PLOT, DRAW, CIRCLE, POINT, pixels

In this chapter we shall see how to draw pictures with BASin. The part of the screen you can use has 22 lines and 32 columns, making  $22 \times 32 = 704$  character positions. As you may remember from Chapter 16, each of these character positions is made of an 8 by 8 square of dots, and these are called pixels (picture elements).

A pixel is specified by two numbers, its coordinates. The first, its x coordinate, says how far it is across from the extreme left-hand column. (Remember, x is a cross.) The second, its y coordinate, says how far it is up from the bottom (wise up). These coordinates are usually written as a pair in brackets, so (0,0), (255,0), (0,175) and (255,175) are the bottom left-, bottom right-, top left- and top right-hand corners.

The statement

```
PLOT x coordinate, y coordinate
```

inks in the pixel with these coordinates, so this measles program

### 10 PLOT INT (RND\*256), INT(RND\*176): INPUT a\$: GO TO 10

plots a random point each time you press **Enter**.

Here is a rather more interesting program. It plots a graph of the function **SIN** (a sine wave) for values between 0 and  $2\pi$ .

# 10 FOR n=0 TO 255

# 20 PLOT n,88+80\*SlN (n/128\*PI) 30 NEXT n

This next program plots a graph of SQR (part of a parabola) between 0 and 4:

# 10 FOR n=0 TO 255 20 PLOT n,80\*SQR (n/64) 30 NEXT n

Notice that pixel coordinates are rather different from the line and column in an AT item. You may find the diagram in Chapter 15 useful when working out pixel coordinates and line and column numbers.

To help you with your pictures, the computer will draw straight lines, circles and parts of circles for you, using the **DRAW** and **CIRCLE** statements.

The statement **DRAW** to draw a straight line takes the form

#### DRAW x, y

The starting place of the line is the pixel where the last PLOT, DRAW or CIRCLE statement left off (this is called the PLOT position; RUN, CLEAR, CLS and NEW reset it to the bottom left hand corner, at (0,0)), and the finishing place is x pixels to the right of that and y pixels up. The DRAW statement on its own determines the length and direction of the line, but not its starting point.

Experiment with a few PLOT and DRAW commands; for instance

PLOT 0,100: DRAW 80,-35 PLOT 90,150: DRAW 80,-35 Notice that the numbers in a **DRAW** or **PLOT** statement can be negative.

You can also plot and draw in colour, although you have to bear in mind that colours always cover the whole of a character position and cannot be specified for individual pixels. When a pixel is plotted, it is set to show the full ink colour, and the whole of the character position containing it is given the current ink colour. This program demonstrates this:

```
10 BORDER 0:
   PAPER 0:
   INK 7:
   CLS:
   REM black out screen
20 LET x1=0:
   LET y1=0:
   REM start of line
30 LET c=1:
   REM for ink colour, starting bl
   LET x2=INT (RND*256):
40
   LET y2=INT (RND*176):
   REM random finish of line
   DRAW INK c;x2-x1,y2-y1
50
   LET x1=x2:
60
   LET y1=y2:
   REM next line starts where last
70 LET c=c+1:
```

```
IF c=8 THEN
LET c=1:
REM new colour
80 GO TO 40
```

The lines seem to get broader as the program goes on, and this is because a line changes the colours of all the inked in pixels of all the character positions that it passes through. Note that you can embed PAPER, INK, FLASH, BRIGHT, INVERSE and OVER items in a PLOT or DRAW statement just as you could with PRINT and INPUT. They go between the key word and the coordinates, and are terminated by either semicolons or commas.

An extra frill with **DRAW** is that you can use it to draw parts of circles instead of straight lines, by using an extra number to specify an angle to be turned through: the form is

DRAW x, y, a

x and y are used to specify the finishing point of the line just as before and a is the number of radians that it must turn through as it goes - if a is positive it turns to the left, while if a is a negative it turns to the right. Another way of seeing a is as showing the fraction of a complete circle that will be drawn: a complete circle is  $2\pi$  radians, so if a =  $\pi$  it will draw a semicircle, if a =  $0.5 \times \pi$  a quarter of a circle, and so on.

For instance suppose a =  $\pi$ . Then whatever values x and y take, a semicircle will be drawn. Run

10 PLOT 100,100: DRAW 50,50, PI

which will draw this:

```
finish at (150,150)
start at (100,100)
```

The drawing starts off in a south-easterly direction, but by the time it stops it is going north-west: in between it has turned round through 180 degrees, or  $\pi$  radians (the value of a).

Run the program several times, with **PI** replaced by various other expressions e.g. -**PI**, **PI**/2, **3**\***PI**/2, **PI**/4, 1, 0.

The last statement in this chapter is the **CIRCLE** statement, which draws an entire circle. You specify the coordinates of the centre and the radius of the circle using

CIRCLE x coordinate, y coordinate, radius

Just as with **PLOT** and **DRAW**, you can put the various sorts of colour items in at the beginning of a **CIRCLE** statement.

The **POINT** function tells you whether a pixel is ink or paper colour. It has two arguments, the coordinates of the pixel (and they must be enclosed in brackets); and its result is 0 if the pixel is paper colour, 1 if it is ink colour. Try

```
CLS:
PRINT POINT (0,0):
PLOT 0,0:
PRINT POINT (0,0)
```

Type

```
PAPER 7:
INK 0
```

and let us investigate how **INVERSE** and **OVER** work inside a **PLOT** statement. These two affect just the relevant pixel, and not the rest of the

character positions. They are normally off (0) in a **PLOT** statement, so you only need to mention them to turn them on (1).

Here is a list of the possibilities for reference:

**PLOT** - this is the usual form. It plots an ink dot, i.e. sets the pixel to show the ink colour.

**PLOT INVERSE 1;** - this plots a dot of ink eradicator, i.e. it sets the pixel to show the paper colour.

**PLOT OVER 1;** - this changes the pixel over from whatever it was before: so if it was ink colour it becomes paper colour, and vice versa.

**PLOT INVERSE 1; OVER 1; -** this leaves the pixel exactly as it was before; but note that it also changes the **PLOT** position, so you might use it simply to do that.

As another example of using the **OVER** statement fill the screen up with writing using black on white, and then type

### PLOT 0,0: DRAW OVER 1;255,175

This will draw a fairly decent line, even though it has gaps in it wherever it hits some writing. Now do exactly the same command again. The line will vanish without leaving any traces whatsoever. This is the great advantage of **OVER 1**. If you had drawn the line using

```
PLOT 0,0:
DRAW 255,175
```

and erased it using

```
PLOT 0,0:
DRAW INVERSE 1;255,175
```

then you would also have erased some of the writing.

Now try

PLOT 0,0: DRAW OVER 1;250,175

and try to undraw it by

```
DRAW OVER 1;-250,-175
```

This doesn't quite work, because the pixels the line uses on the way back are not quite the same as the ones that it used on the way down. You must undraw a line in exactly the same direction as you drew it.

One way to get unusual colours is to speckle two normal ones together in a single square, using a user-defined graphic. Run this program:

# 1000 FOR n=0 TO 6 STEP 2 1010 POKE USR "a"+n, BIN 01010101: POKE USR "a"+n+1, BIN 1010101 1020 NEXT n

which gives the user-defined graphic corresponding to a chessboard pattern. If you print this character (graphics mode, then A) in red ink on yellow paper, you will find it gives a reasonably acceptable orange.

Exercises

1. Play about with PAPER, INK, FLASH and BRIGHT items in a PLOT statement. These are the parts that affect the whole of the character position containing the pixel. Normally it is as though the PLOT statement had started off

and only the ink colour of a character position is altered when something is plotted there, but you can change this if you want.

Be especially careful when using colours with **INVERSE 1**, because this sets the pixel to show the paper colour, but changes the ink colour and this might not be what you expect.

2. Try to draw circles using **SIN** and **COS** (if you have read Chapter 10, try to work out how). Run this:

10 FOR n=0 TO 2\*PI STEP PI /180

- 20 PLOT 100+80\*COS n,87+80\*SIN n
- 30 NEXT n
- 40 CIRCLE 150,87,80

You can see that the **CIRCLE** statement is much quicker, even if less accurate.

3. Try

CIRCLE 100,87,80:

# DRAW 50,50

You can see from this that the **CIRCLE** statement leaves the **PLOT** position at a rather indeterminate place - it is always somewhere about half way up the right hand side of the circle. You will usually need to follow the **CIRCLE** statement with a **PLOT** statement before you do any more drawing.

4. Here is a program to draw the graph of almost any function. It first asks you for a number n; it will plot the values from -n to +n. It then asks you for the function itself, input as a string. The string should be an expression using x as the argument of the function.

```
10 PLOT 0,87:
    DRAW 255,0
20 PLOT 127,0:
    DRAW 0,175
30 INPUT s, e$
35 LET t=0
40 FOR f=0 TO 255
50 LET x=(f-128)*s/128:
    LET y=VAL e$
60 IF ABS y>87 THEN
```

```
LET t=0:

GO TO 100

70 IF NOT t THEN

PLOT f,y+88:

LET t=1:

GO TO 100

80 DRAW 1,y-old y

100 LET old y=INT (y+.5)

110 NEXT f
```

Run it and, as an example, type in 10 for the number n and  $10 \text{*TAN} \times \text{for the function}$ . It will plot a graph of tan x as x ranges from -10 to +10.

← Chapter 16

Chapter 18 →

## **Chapter 18: Motion**

Summary

### PAUSE, INKEY\$, PEEK

Quite often you will want to make the program take a specified length of time, and for this you will find the **PAUSE** statement useful.

#### PAUSE n

stops computing and displays the picture for n frames of the television (at 50 frames per second in Europe or 60 in America). n can be up to 65535, which gives you just under 22 minutes; if n=0 then it means 'PAUSE for ever'.

A pause can always be cut short by pressing a key (note that **Esc** will cause a break as well). You have to press the key down after the pause has started.

This program works the second hand of a clock:

10	REM First we draw the clock fa
20	FOR n=1 TO 12
30	PRINT AT 10-10*COS (n/6*PI),16
40	NEXT n
50	REM Now we start the clock
60	FOR t=0 TO 200000:
	REM t is the time in seconds
70	LET a=t/30*PI:
	REM a is the angle of the seco
80	LET sx=80*SIN a:

```
LET sy=80*COS a
200 PLOT 128,88:
DRAW OVER 1;sx,sy:
REM draw second hand
210 PAUSE 42
220 PLOT 128,88:
DRAW OVER 1;sx,sy:
REM erase second hand
400 NEXT t
```

This clock will run down after about 55.5 hours because of line 60, but you can easily make it run longer. Note how the timing is controlled by line 210. You might expect **PAUSE 50** to make it tick one a second, but the computing takes a bit of time as well and has to be allowed for. This is best done by trial and error, timing the computer clock against a real one, and adjusting line 210 until they agree. (You can't do this very accurately; an adjustment of one frame in one second is 2% or half an hour in a day.)

There is a much more accurate way of measuring time. This uses the contents of certain memory locations. The data stored is retrieved by using PEEK. Chapter 25 explains what we're looking at in detail. The expression used is

### (65536\*PEEK 23674+256\*PEEK 23673+PEEK 23672)/50

This gives the number of seconds since the computer was turned on (up to about 3 days and 21 hours, when it goes back to 0).

Here is a revised clock program to make use of this:

## 10 REM First we draw the clock fa

```
20 FOR n=1 TO 12
```

- 30 PRINT AT 10-10\*COS (n/6\*PI),16

- 40 NEXT n
- 50 DEF FN t()=INT (65536\*PEEK 236 REM number of seconds since st
- 100 REM Now we start the clock

```
110 LET t1=FN t()
```

120 LET a=t1/30\*PI:

REM a is the angle of the seco

- 130 LET sx=72\*SIN a: LET sy=72\*COS a
- 140 PLOT 131,91: DRAW OVER 1;sx,sy: REM draw hand

```
200 \text{ LET } t=FN t()
```

210 IF t<=t1 THEN GO TO 200:

REM wait until time for next h

```
220 PLOT 131,91:
```

```
DRAW OVER 1;sx,sy:
```

REM rub out old hand

```
230 LET t1=t:
```

```
GO TO 120
```
The internal clock that this method uses should be accurate to about .01% as long as the computer is just running its program, or 10 seconds per day; but it stops temporarily whenever you do BEEP, or a LOAD or SAVE operation, or use the printer or any of the other extra pieces of equipment you can use with the computer. All these will make it lose time.

The numbers PEEK 23674, PEEK 23673 and PEEK 23672 are held inside the computer and used for counting in 50ths of a second. Each is between 0 and 255, and they gradually increase through all the numbers from 0 to 255; after 255 they drop straight back to 0.

The one that increases most often is PEEK 23672. Every 1/50 second it increases by 1. When it is at 255, the next increase takes it to 0, and at the same time it nudges PEEK 23673 up by 1. When (every 256/50 seconds) PEEK 23673 is nudged from 255 to 0, it in turn nudges PEEK 23674 up by 1. This should be enough to explain why the expression above works.

Now, consider carefully: suppose our three numbers are 0 (for PEEK 23674), 255 (for PEEK 23673) and 255 (for PEEK 23672). This means that it is about 21 minutes after switch-on - our expression ought to yield

#### (65536\*0+256\*255+255)/50 = 1310.7

But there is a hidden danger. The next time there is a 1/50 second count, the three numbers will change to 1, 0 and 0. Every so often, this will happen when you are half way through evaluating the expression: the computer would evaluate PEEK 23674 as 0, but then change the other two to 0 before it can peek them. The answer would then be

(65536\*0+256\*0+0)/50 = 0

which is hopelessly wrong.

A simple rule to avoid this problem is evaluate the expression twice in succession and take the larger answer. If you look carefully at the program above you can see that it does this implicitly.

Here is a trick to apply the rule. Define functions

10	DEF	FN $m(x,y)=(x+y+ABS (x-y))/2$
	REM	the larger of x and y
20	DEF	FN u()=(65536*PEEK 23674+25
	REM	time, may be wrong
30	DEF	FN t()=FN m(FN u(), FN u())
	REM	time, right

You can change the three counter numbers so that they give the real time instead of the time since the computer was switched on. For instance, to set the time at 10.00am, you work out that this is 10\*60\*60\*50=1800000 fiftieths of a second, and that

1800000=65536\*27+256\*119+64

To set the three numbers to 27, 119 and 64, you do

#### POKE 23674,27: POKE 23673,119: POKE 23672,64

In countries with mains frequencies of 60 Hertz these programs must replace '50' by '60' where appropriate.

The function **INKEY\$** (which has no argument) reads the keyboard. If you are pressing exactly one key (or a **Shift** key and just one other key) then the result is the character that that key normally gives; otherwise the result is the empty string.

Try this program, which works like a typewriter.

# 10 IF INKEY\$ <>"" THEN GO TO 10

```
20 IF INKEY$ ="" THEN
GO TO 20
30 PRINT INKEY$;
40 GO TO 10
```

Here line 10 waits for you to lift your finger off the keyboard and line 20 waits for you to press a new key.

Remember that unlike **INPUT**, **INKEY\$** doesn't wait for you. So you don't type **Enter**, but on the other hand if you don't type anything at all then you've missed your chance.

Exercises

- 1. What happens if you miss out line 10 in the typewriter program?
- 2. Another way of using **INKEY\$** is in conjunction with **PAUSE**, as in this alternative typewriter program.

10 PAUSE 0

- 20 PRINT INKEY\$;
- 30 GO TO 10

To make this work, why is it essential that a pause should not finish if it finds you already pressing a key when it starts?

- 3. Adapt the second hand program so that it also shows minute and hour hands, drawing them every minute. If you're feeling ambitious, arrange so that every quarter of an hour it puts on some kind of show you could produce the Big Ben chimes with BEEP. (See next chapter.)
- 4. (For sadists.) Try this:

10	IF INH	KEY\$ ="" THEN	
	GO TO	10	
20	PRINT	AT 11,14;"OUCH!	"
30	IF IN	KEY\$ <>"" THEN	
	GO TO	30	
40	PRINT	AT 11,14;"	11
50	GO TO	10	

← Chapter 17

Chapter 19 →

## **Chapter 19: SOUND**

Summary

#### **BEEP, PLAY**

Depending upon which model you are using, the Spectrum had two methods of generating sound. The original 48k model comes equipped with a loudspeaker, which is driven by the **BEEP** command. It is capable of outputting a single channel of sound (i.e, no more than one note at a time).

The 128k Spectrums have, as well as the loudspeaker, a sound chip called the AY-3-8912. This chip is capable of generating much more complex sounds, and can play up to three of them at once. The chip is driven using the PLAY command. Note that using the PLAY command will indicate to BASin that you are working on a 128k program. For information about this, and limitations the 128k BASIC has, see here.

The BEEP command

The loudspeaker is sounded by using the **BEEP** statement,

BEEP duration, pitch

where, as usual, 'duration' and 'pitch' represent any numerical expressions. The duration is given in seconds, and the pitch is given in semitones above middle C using negative numbers for notes below middle C.

Here is a diagram to show the pitch values of all the notes in one octave on the piano:



To get higher or lower notes, you have to add or subtract 12 for each octave that you go up or down.

The simplest way to compose a tune in BASin is to use the **BEEP** Composer, but here are some tips for creating a tune in code.

If you have a piano in front of you when you are programming a tune, this diagram will probably be all that you need to work out the pitch values. If, however, you are transcribing straight from some written music, then we suggest that you draw a diagram of the stave with the pitch value written against each line and space, taking the key into account.

For example, type:

```
10 PRINT "Frere Gustav"
20 BEEP 1,0:
   BEEP 1,2:
   BEEP .5,3:
   BEEP .5,2:
   BEEP 1,0
30 BEEP 1,0:
   BEEP 1,2:
   BEEP .5,3:
   BEEP .5,2:
   BEEP 1,0
40 BEEP 1,3:
   BEEP 1,5:
   BEEP 2,7
50 BEEP 1,3:
   BEEP 1,5:
   BEEP 2,7
60 BEEP .75,7:
   BEEP .25,8:
   BEEP .5,7:
   BEEP .5,5:
```



When you run this, you should get the funeral march from Mahler's first symphony, the bit where the goblins bury the US Cavalry man.

Suppose for example that your tune is written in the key of C minor, like the Mahler above. The beginning looks like this:



and you can write in the pitch values of the notes like this:



We have put in two ledger lines, just for good measure. Note how the E flat in

the key signature affects not only the E in the top space, flattening it from 16 to 15, but also the E on the bottom line, flattening it from 4 to 3. It should now be quite easy to find the pitch value of any note on the stave.

If you want to change the key of the piece, the best thing is to set up a variable key and insert key+ before each pitch value: thus the second line becomes

20 BEEP 1,key+0: BEEP 1,key+2: BEEP .5,key+3: BEEP .5,key+2: BEEP 1,key+0

Before you run a program you must give key the appropriate value - 0 for C minor, 2 for D minor, 12 for C minor an octave up, and so on. You can get the computer in tune with another instrument by adjusting key, using fractional values.

You also have to work out the durations of all the notes. Since this is a fairly slow piece, we have allowed one second for a crotchet and based the rest on that, half a second for a quaver and so on.

More flexible is to set up a variable crotchet to store the length of a crotchet and specify the durations in terms of this. Then line 20 would become

20	BEEP	crotchet,key+0:
	BEEP	<pre>crotchet,key+2:</pre>
	BEEP	<pre>crotchet/2,key+3:</pre>
	BEEP	<pre>crotchet/2,key+2:</pre>
	BEEP	crotchet,key+0

(You will probably want to give crotchet and key shorter names.)

By giving crotchet appropriate values, you can easily vary the speed of the piece.

Remember that because there is only one loudspeaker in the computer you can only play one note at a time, so you are restricted to unharmonized tunes. If you want any more you must sing it yourself.

Try programming tunes in for yourself - start off with fairly simple ones like 'Three Blind Mice'. If you have neither piano nor written music, get hold of a very simple instrument like a tin whistle or a recorder, and work the tunes out on that. You could make a chart showing the pitch value for each note that you can play on this instrument.

Type:

```
FOR n=0 TO 1000:
BEEP .5,n:
NEXT n
```

This will play notes as high as it can, and then stop with error report **B Integer out of range**. You can print out **n** to find out how high it did actually get.

Try the same thing, but going down into the low notes. The very lowest notes will just sound like clicks; in fact the higher notes are also made of clicks in the same way, but faster, so that the ear cannot distinguish them.

Only the middle range of notes are really any good for music; the low notes sound too much like clicks, and the high notes are thin and tend to warble a bit.

Type in this program line:

```
10 BEEP .5,0:
BEEP .5,2:
BEEP .5,4:
BEEP .5,5:
BEEP .5,7:
BEEP .5,9:
BEEP .5,11:
BEEP .5,12:
STOP
```

This plays the scale of C major, which uses all the white notes on the piano from middle C to the next C up. The way this scale is tuned is exactly the same as on a piano, and is called even-tempered tuning because the pitch interval of a semitone is the same all the way up the scale. A violinist, however, would play the scale very slightly differently, adjusting all the notes to make them sound more pleasing to the ear. He can do this just by moving his fingers very slightly up or down the string in a way that a pianist can't.

The natural scale, which is what the violinist plays, comes out like this:

20	BEEP	.5,0:
	BEEP	.5,2.039:
	BEEP	.5,3.86:
	BEEP	.5,4.98:
	BEEP	.5,7.02:
	BEEP	.5,8.84:
	BEEP	.5,10.88:
	BEEP	.5,12:
	STOP	

You may or may not be able to detect any difference between these two; some people can. The first noticeable difference is that the third note is slightly flatter in the naturally tempered scale. If you are a real perfectionist, you might like to program your tunes to use this natural scale instead of the even-tempered one. The disadvantage is that although it works perfectly in the key of C, in other keys it works less well - they all have their own natural scales - and in some keys it works very badly indeed. The even-tempered scale is only slightly off, and works equally well in all keys.

This is less of a problem on the computer, of course, because you can use the trick of adding on a variable key.

Some music - notably Indian music - uses intervals of pitch smaller than a semitone. You can program these into the **BEEP** statement without any trouble; for instance the quartertone above middle C has a pitch value of .5.

You can make the keyboard beep instead of clicking by

#### POKE 23609,255

The second number in this determines the length of the beep (try various values between 0 and 255). When it is 0, the beep is so short that it sounds like a soft click.

If you are interested in doing more with sound from the Spectrum, like hearing the sound that **BEEP** makes on something other the internal speaker, you will find that the signal is present on both the 'MIC' and the 'EAR' sockets. It will be at a higher level on the 'EAR' socket, but otherwise they are the same. You may use this to connect an earphone or a pair of headphones to your Spectrum. This will not cut out the internal loudspeaker. If you are really keen to make a lot of noise you could connect it up to an amplifier - the 'MIC' socket will probably give about the right level - or you could record the sound onto tape and get the Spectrum to play along with itself.

You will not damage the Spectrum even if you short-circuit the 'MIC' or 'EAR' sockets, so experiment to find which gives the best output for what you want to do.

#### The PLAY command

The **PLAY** command allows you to make more complex pieces of music. However, it is a 128k only command, and will not run on a standard 48k Spectrum. See the chapter on **BASin** and the 128k for more details about this.

In the examples that follow, it is important that you type in the string expressions *exactly* as shown in upper case and lower case letters, i.e. the example "ga" should *not* be typed in as "Ga", "gA" or "GA".

Type in this command (don't worry about what it means just yet)...

#### PLAY "ga"

Two notes were played - the second slightly higher than the first. The difference between the notes is called a *tone*. Now try...

#### PLAY "g\$a"

Again there were two notes played - the first one was the same as the previous example, but there was less of a jump to the second. If you didn't hear the difference, then try the first example followed by the second again. The second example has half the difference between notes, and this is called a *semitone*.

When you're happy with semitones, try this...

#### PLAY "gD"

This sort of difference is called a *fifth*, and occurs quite often in music of all types. With that example ringing in your ears, type...

#### PLAY "gG"

Although (hopefully) you noticed that there was a much bigger difference that time than for the fifth, the two notes somehow sounded much more similar. This is called an *octave*, and is the point at which music starts to 'repeat itself'. Don't worry about that unduly, just remember what an octave sounds like.

**PLAY** is much more flexible than **BEEP** - it can play up to three voices in harmony with all manner of effects, and gives a much higher quality of sound. It's also much easier to use. For example, to play **A** above middle **C** for half a second, type in...

#### PLAY "a"

... and to play the **C** major scale (which needed a program to itself before), use...

#### PLAY "cdefgabC"

Notice that the last C in the example above is in upper case. This tells the **PLAY** command to play it in an octave higher than the lower case c. A *scale*, by the way, is the term used for a set of notes between two Cs. Why major? There are two main classes of scale, major and minor, and this is just musical shorthand for describing the two different sets. Just for interest, the C minor scale sounds like

this...

#### PLAY "cd\$efg\$a\$bC"

Preceding a note by \$ drops it by a semitone (*flattens* it), and preceding a note by # raises it by a semitone (*sharpens* it). The **PLAY** command spans 9 octaves, and can be told which to use by having the upper case O followed by a number, in the list of notes that it is given. Type in this little program...

10 LET o\$="05" 20 LET n\$="DECcg" 30 LET a\$=o\$+n\$ 40 PLAY a\$

There are a few new things in this program. Firstly, **PLAY** is just as happy with a string variable as with a string constant. In other words, providing that **a\$** has been set up beforehand, **PLAY a\$** works just as well as **PLAY "O5DECcg"**. In fact, using variables in **PLAY** statements has distinct advantages, and we shall be doing this from now on.

Notice also that the string **a\$** has been 'built up' by combining two smaller strings **o\$** and **n\$**. While this doesn't make much difference at this sort of level, **PLAY** can cope with strings many thousands of notes long, and the only sensible way of creating and editing those strings from BASIC is to combine lots of smaller strings in this way.

Now run the above program. Edit line 10 so that "05" becomes "07", and run it again, or if you want to make a big spaceship, make it "02". If you don't specify an octave number for a particular string, then BASin assumes that you want octave 5. Here is a diagram of the notes and octave numbers which correspond to the standard even-tempered music scale.



There is a lot of overlap, so for example, "O3D" is the same as "O4d". This makes it easier to write tunes without having to change octave all the time. Some of the notes in the lowest octaves (0 and 1) aren't very accurate for technical reasons, and so the computer just makes a brave attempt at getting as close as possible.

**PLAY** can also handle many different lengths of note. Edit the program above so that line 10 is now...

#### 10 LET o\$="2"

... and run it. Then alter the setting of **0**\$ between "**1**" and "**9**". The note length can be changed anywhere in a string by including a number between 1 and 9, and this is effective for all subsequent notes until a new number is encountered. Each of these nine note lengths has a specific musical name, and looks different when written down in musical notation. The following table shows which is which...

Number	Note Name	Musical Symbol

1	Semi-quaver	Ą
2	Dotted semi-quaver	₽.
3	Quaver	4
4	Dotted quaver	<b>"</b> ħ.
5	Crotchet	Ļ
6	Dotted crotchet	<b>.</b>
7	Minim	J
8	Dotted minim	J.
9	Semi-breve	o

**PLAY** can also cope with *triplets*, which are three notes played in the time for two. Unlike simple note lengths, the triplet number only applies for the three notes immediately following, and then the previous note length number resumes. The triplet numbers are as follows...

Number	Note Name	Musical Symbol
10	Triplet semi-quaver	****

11	Triplet quaver	3 <b>a a</b>
12	Triplet crotchet	

**PLAY** is quite happy about being told to 'shut up'! A timed period during which no notes play is called a *rest*, and the "&" is used to signify this. The length of the rest it produces is the same as the current note length. To demonstrate, edit lines **10** and **20** to...

10 LET o\$="04" 20 LET n\$="DEC&cg;"

Two notes played together without a break are called *tied notes*, which are signified in a PLAY command by an \_ underline, so a crotchet c and a minim c tied together would be "5\_7c". (The second value is then used as the note length for all subsequent notes, as before.)

There are 2 occasions when the ambiguity creeps in. Say that a piece of music needs octave 6 and a note length of 2, then...

#### 10 LET 0\$="062"

...seems a good bet - but no! The computer will find the O and try to read the number following it. When it finds 62, it will stop with the report n Out of range. In cases like this, there is a 'dummy note' called N that just serves to split things up, so line 10 should be...

#### 10 LET 0\$="06N2"

The volume can be set between 0 (minimum) and 15 (maximum) using "V" followed by a number. In practice, only 10 to 16 are likely to be useful, as 1-9 are too soft unless the computer is being used with an amplifier. As previously

mentioned, **BEEP** is louder than a single channel of **PLAY**, but if all three channels play a note at volume 15, then it should be at the same level as a not produced by **BEEP**.

Playing more than one channel at the same time is very simple; you just separate lists of notes by commas. Try this new program...

```
10 LET a$="04cCcCgGgG"
20 LET b$="06CaCe$bd$bD"
30 PLAY a$, b$
```

In general, there is no difference between the three channels, and any string of notes can be put onto any channel. The overall speed of the music, the *tempo*, must be in the string assigned to channel A (the first string after PLAY), otherwise it will be ignored. To set tempo in beats (crotchets) per minute, use "T" followed by a number between 60 and 240. The standard value is 120, or two crotchets per second. Modify the program above to...

```
5 LET t$="T120"
```

```
10 LET a$=t$+"04cCcCgGgG"
```

```
20 LET b$="06CaCe$bd$bD"
```

```
30 PLAY a$, b$
```

...and run it several times, changing line 5 for different tempos.

A common feature in music is the repetition of a group of notes. Any part of a string can be repeated by enclosing it in brackets, so if you change line 10 to...

#### 10 LET a\$=t\$+"04(cC)(gG)"

**PLAY** treats it just the same as the old line 10. If you include a closing bracket, (with no matching opening bracket) then the string up to that point is repeated indefinitely. This is useful for rhythm effects and basslines. To demonstrate, try this (you'll have to use the **ESC** key to stop the sound)...

PLAY "04N2cdefgfed)"

...and...

#### PLAY "04N2cd(efgf)ed)"

If you set up an infinitely repeating bassline, and then play a melody with it, then it would be nice if the bassline stops when the melody does. There is a device to do this - of PLAY comes across "H" (for Halt) in any of the strings it is playing, then it stops all sound immediately. Run the following program (again, you'll have to use **ESC** to stop it)...

- 10 LET a\$="cegbdfac"
- 20 LET b\$="04cC)"
- 30 PLAY a\$, b\$

Now modify line 10 to...

#### 10 LET a\$="cegbdfaCH"

...and run it again.

So far we've only used notes which start and stop at one level of volume. BASin can alter athe volume of a note while it is playing, so it can start loud and die away like a piano, or rise and fall like a dog growling. To turn these effects on, use "W" (for Waveform) followed by a number between 0 and 7, together with "U" for each channel you want to use the effect on. Any channel with a volume setting ("V") will not respond to "U". This table shows graphically how the volume changes for each setting...

0		Single decay, then off.
1	∕	Single attack, then off.
2	7	Single decay, then hold.

3		Single attack, then hold.
4	777777	Repeated decay.
5	11111	Repeated attack.
6	$\sim \sim \sim$	Repeated attack-decay.
7	$\sim\sim\sim\sim$	Repeated decay-attack.

This program plays the same note with each effect in turn, so you can compare them against the diagram above.

#### 10 LET a\$="UX1000W0C&W1C;&W2C;&W3C;&W4C;&W5C;&W 20 PLAY a\$

The U turns on effects, and the W selects which waveform to use. There's also an "X1000". The X sets how long the effect will last for (from 0 to 65535). If you don't include an X then BASin will choose the longest value. Waveforms that settle down (0 to 3 in the table above) after the initial part, work best with X settings of about 1000, whereas repetetive effects (4-7) are more effective with short values like 300. Try varying the X setting in the program above to get some idea of how each works.

The PLAY command isn't limited to pure musical notes. There are also three "white noise" generators (white noise is a sound which is like an un-tuned FM radio or TV), and any of the three channels can play notes, white noise, or a mixture of both. To select a mix of noise and note, you may use "M" followed by a number between 1 and 63. You can work out which number to use from the following table...

	Tone Channels			Noise Channels		
	А	В	С	А	В	С
Number	1	2	4	8	16	32

Write down the numbers corresponding to the effects you want, and then add them together. If you wanted A to be noise, B to be tone, and C to be both tone and noise, then add 8, 2, 4, and 32 together to get 46 (the order of the channels is the order of the strings which follow the PLAY command). The best effects can be obtained with the A channel - don't be afraid to experiment.

By now, you'll be writing symphonies. However, it can be difficult to work out just which part of the music a particular section of string is responsible for. To alleviate this problem, your music string may include 'comments' enclosed between ! exclamation marks; for example...

#### 10 LET z\$=z\$+"CDcE3Ge4\_6f! end of the 75th bar

The **PLAY** command will simply 'hop over' any comments in the string.

#### MIDI programming

The original 128k models of the Spectrum had the ability to control an external MIDI device. BASin emulates this by allowing your programs to drive your MIDI soundcard. For information about setting up this feature, see the relevant options page.

If you have an electronic musical instrument with MIDI, then you can control it using PLAY. Up to 8 channels of music can be sent to synthesisers, drum machines or sequencers. The PLAY command is constructed exactly as described so far in this section, except that each string should include a "Y" followed by a number between 1 and 16. The number after the Y controls which channel the music data is assigned to. Up to eight strings can be used; the first three strings will still be played through the normal Spectrum output, so you might want to turn down the volume. You can also send MIDI programming codes via the PLAY command, using "Z" followed by the code number. Key velocities (loudness) are calculated and sent at 8 times the V setting (so "V6"

will send 48 as a key velocity).

So, to send a little tune (in four part harmony) to a four-voice synthesiser (after consulting your synth's handbook to find out how to allocate MIDI channels to different voices), you would use the PLAY command with four strings, each starting with a Y followed by a number. This example program illustrates the PLAY command in some of its full glory...

```
10 LET a$="Y1T10002(((1CCg$b))(($E$E$b$D))((GGD
```

20 LET b\$="Y205N&&&&C;\$bfG)"

```
30 LET c$="Y304((3C&)C&1CCDD(3$E&)$E&1$E$EEE(3F)
```

```
40 LET d$="Y4N9&&&&&(9EGF7b5CD))"
```

50 PLAY a\$, b\$, c\$, d\$

Summary table

Finally, here is a brief list of the parameters that can be used in the string of a **PLAY** command, together with any values they may take...

String	Function
ag, AG	Specifies the pitch of the note within the current octave range.
\$	Specifies that the note which follows must be flattened.
#	Specifies that the note which follows must be sharpened.
0	Specifies the octave number to be used (followed by 0 to 8).
112	Specifies the length of notes to be used.
&	Specifies that a rest is to be played.
_	Specifies that a tied note is to be played.
Ν	Separates two numbers.
V	Specifies the volume to be used (followed by 0 to 15).
W	Specifies the volume effect to be used (followed by 0 to 7).
U	Specifies that volume effects are to be used in the string.

Х	Specifies the duration of the volume effect (followed by 0 to 65535).
Т	Specifies the tempo of the music (followed by 60 to 240).
()	Specifies that the enclosed phrase is to be repeated.
!!	Specifies that the enclosed text is to be skipped over.
Н	Specifies that the PLAY command must stop.
Μ	Specifies the channel(s) to be used (followed by 0 to 63).
Υ	Specifies that a MIDI channel is to be used (followed by 1 to 16).
Z	Specifies MIDI programming code (followed by code number).

Exercises

- 1. Rewrite the Mahler program so that it uses **FOR** loops to repeat the bars.
- 2. Program the computer so that it plays not only the funeral march, but also the rest of Mahler's first symphony.

← Chapter 18

Chapter 20 →

## **Chapter 20: File storage**

Summary

#### LOAD, SAVE, VERIFY, MERGE

Loading and saving work differently in BASin than on a real ZX Spectrum. See Loading and saving files.

It is also recommended that you read the sections on LOAD, SAVE, VERIFY and MERGE. These sections will acquaint you with how those commands work.

When you have read them, then come back here for more information as to the things you can do with these commands.

Working with MERGE

We have seen that LOAD deletes the old program and variables in the computer before loading in the new ones from disk; there is another command, MERGE, that does not. MERGE only deletes an old program line or variable if it has to because there is a new one with the same line number or name. Type in the 'dice' program in Chapter 11 and save it on disk, as "dice". Now enter and run the following:

- 1 PRINT 1
- 2 PRINT 2
- 10 PRINT 10
- 20 LET x=20

and then proceed as for the verification, but replacing **VERIFY** "dice" with

#### MERGE "dice"

If you list the program you can see that lines 1 and 2 have survived, but lines 10 and 20 have been replaced by those from the dice program. X has also survived

(try **PRINT** X).

You have now seen simple forms of the four statements used with the file system:

**SAVE** stores the program and variables in a file.

**VERIFY** checks the program and variables in a file against those already in the computer.

**LOAD** clears the computer of all its program and variables, and replaces them with new ones read in from a file.

**MERGE** is like **LOAD** except that it does not clear out an old program line or variable unless it has to because its line number or name is the same as that as that of a new one from the file.

In each of these, the keyword is followed by a string: for SAVE this provides a name for the program on disk, while for the other three it tells the computer which program to load. There are a couple of twists to all this.

You can provide the empty string instead of a filename; then the computer prompts you to browse for a file on disk. (On a real ZX Spectrum, the empty string indicates the next file on the cassette tape - regardless of its filename - and cannot be used with SAVE.)

A variant on **SAVE** takes the form

SAVE string LINE number

A program saved using this is recorded in such a way that when it is read back by LOAD (but not MERGE) it automatically jumps to the line with the given number, thus running itself.

So far, the only kinds of information we have stored on disk have been programs together with their variables. There are two other kinds as well, called arrays and bytes.

Arrays are dealt with slightly differently:

You can save arrays on disk using DATA in a SAVE statement by

```
SAVE string DATA arrayname()
```

String is the name that the information will have on disk and works in exactly the same way as when you save a program or plain bytes.

The array name specifies the array you want to save, so it is just a letter or a letter followed by **\$**. Remember the brackets afterwards; you might think they are logically unnecessary but you still have to put them in to make it easier for the computer.

Be clear about the separate roles of string and array name. If you say (for instance)

```
SAVE "Bloggs" DATA b()
```

then SAVE takes the array **b** from the computer and stores it on disk under the name "Bloggs". When you type

#### VERIFY "Bloggs" DATA b()

the computer will look for a number array stored on disk under the name "Bloggs" and check it against the array **b** in the computer.

#### LOAD "Bloggs" DATA b()

finds the array on disk, and then - if there is room for it in the computer - deletes any array already existing called **b** and loads in the new array from disk, calling it **b**.

You cannot use **MERGE** with saved arrays.

You can save character (string) arrays in exactly the same way. When you load in a character array, the computer will delete not only any previous character array with the same name, but also any string with the same name.

Byte storage is used for pieces of information without any reference to what the information is used for - it could be television pictures, or used-defined graphics, or something you have made up yourself. It is shown using the word CODE, as in

#### SAVE "picture" CODE 16384,6912

The unit of storage in memory is the byte (a number between 0 and 255), and each byte has an address (which is a number between 0 and 65535). The first number after **CODE** is the address of the first byte to be stored on disk, and the second is the number of bytes to be stored. In our case, 16384 is the address of the first byte in the display file (which contains the television picture) and 6912 is the number of bytes in it, so we are saving a copy of the television screen - try it. The name "picture" works just like the names for programs.

To load it back, use

#### LOAD "picture" CODE

You can put numbers after **CODE** in the form

#### LOAD name CODE start, length

Here length is just a safety measure; when the computer has found the bytes on disk with the right name, it will still refuse to load them in if there are more than length of them - since there is obviously more data than you expected it could otherwise overwrite something you had not intended to be overwritten. It gives the error report R File loading error. You can miss out length, and then the computer will read in the bytes however many there are.

Start shows the address where the first byte is to be loaded back to - this can be different from the address it was saved from, although if they are the same you can miss out the start in the LOAD statement.

CODE 16384, 6912 is so useful for saving and loading the picture that you

can replace it with **SCREEN\$** - for instance:

#### SAVE "picture" SCREEN\$ LOAD "picture" SCREEN\$

BASin supports a number of different file formats. You can indicate the format you wish to use by providing a filename extension.

#### The RAM Disk

BASin provides emulation of the Spectrum 128k and +2's Silicon Disk (or RAM Disk). Be aware that using the Silicon Disk has it's own requirements related to the use of other 128k commands. See the section on 128k commands in BASin for more information.

Anything you can do with SAVE, LOAD or MERGE for files saved from BASin, you can do with the *silicon disk* that's built into the 128k Spectrum models. This acts like the usual cassette, or hard disk in BASin (with a couple of extra commands), with the exception that it's about 64kb in size, very fast, and loses its contents when BASin is reset or quit (However, it *does* survive the NEW command). You use the commands in exactly the same way you would in normal operation - simply add an exclamation mark ! between the command and its associated string. So where you would type...

#### SAVE "squares"

...to save normally, you may instead use...

#### SAVE !"squares"

...to save to the silicon disk.

There are two extra commands for use with the silicon disk. The first one is...

#### CAT !

...which gives you a list of all the programs or data files that are stored on the

disc.

The second one is...

#### ERASE !"filename"

...to get rid of an unwanted program or data.

Perhaps the most obvious use of the silicon disc is to store chunks of BASIC program which can be merged (using MERGE !) into a smaller program, in sequence. This makes it possible to write about 90kb of BASIC program, and hold it in the emulated memory (to do this, the program structure has to be well defined).

One of the more interesting uses of the silicon disk is in *animation*, where a series of pictures can be defined by a 'slow' BASIC program and stored in the silicon disk, then called back to the screen at high speed. The following program gives a taste of this; doubtless you can do better...

```
10 INK 5:
    PAPER 0:
    BORDER 0:
    CLS
20 FOR f=1 TO 10
30 CIRCLE f*20,150,f
40 SAVE !"ball"+STR$(f) CODE 1638
50 CLS
60 NEXT f
70 FOR f=1 TO 10
80 LOAD !"ball"+STR$(f) CODE
```

```
90 NEXT f
100 BEEP 0.01,0.01
110 FOR f=9 TO 2 STEP -1
120 LOAD !"ball"+STR$(f) CODE
130 NEXT f
140 BEEP 0.01,0.01
150 GO TO 70
160 REM use GO TO 160 to clear pic
170 FOR f=10 TO 1 STEP -1
180 ERASE !"ball"+STR$(f)
190 NEXT f
```

Note that in line 40 of this program, the two numbers following **CODE** are the address in memory of the start of the screen, and the length of the top third of it. By only saving and loading the top third, the overall speed is maintained. Lines 160 to 190 are there if you break (using ESC) out of the program, modify the circle drawing bit, and try to save a new set of pictures. So before doing that, type **GO TO 160** to clear out the silicon disc. (Always try to delete files backwards so the last file to be saved will be the first to be deleted. This saves the computer a lot of juggling about, and is much faster).

Summary

Below is a complete summary of the four statements used in this chapter. Name stands for any string expression, and refers to the name under which the information is stored on disk. It should consist of ASCII printing characters. Unlike the original ZX Spectrum, BASin permits filenames to be longer than 10 characters.

There are four sorts of information that can be stored on disk: program and variables (together), number arrays, character arrays and straight bytes.

When you provide the empty string instead of a filename, BASin prompts you for the file to be opened or saved.

SAVE

Saves information on disk under the given name.

Program and variables:
 SAVE (!) name LINE line number

saves the program and variables in such a way that **LOAD** automatically follows with

**GO TO** line number

2. Bytes:

SAVE (!) name CODE start, length

saves length bytes starting at address start.

SAVE (!) name SCREEN\$

is equivalent to

SAVE (!) name CODE 16384, 6912

and saves the television picture.

3. Arrays:

```
SAVE (!)name DATA letter()
or
SAVE (!)name DATA letter$()
```

saves the array whose name is letter or letter\$ (this need bear no relation to name).

4. Verify:

Checks the information against on tape against the information already in memory. Failure to verify gives error R File loading error.

1. Program and variables:

## VERIFY name

2. Bytes:

## VERIFY name CODE start, length

If the bytes name on disk are more than length in number, then gives error R. Otherwise, checks

them against those in memory starting at address start.

## VERIFY name CODE start

checks the bytes name on tape against those in memory starting at the address from which the first disk byte was saved.

## VERIFY name SCREEN\$

is equivalent to

## VERIFY name CODE 16384, 6912

(On an original ZX Spectrum, this will almost certainly fail to verify, because the filename is printed on screen as the file loads from cassette.)

3. Arrays:

VERIFY name DATA letter() or VERIFY name DATA letter\$()

checks the array name on disk against the array

letter or letter\$ in memory.

#### LOAD

Loads new information from disk, deleting old information from memory.

1. Program and variables:

LOAD (!)name

deletes the old program and variables and loads in program and variables name from disk; if the program was saved using SAVE name LINE it performs an automatic jump.

Error **4 Out of memory** occurs if there is no room for the new program and variables. In this case the old program and variables are not deleted.

2. Bytes:

## LOAD (!) name CODE start, length

If the bytes name from disk are more than length in

number then gives error R. Otherwise, loads them into memory starting at address start and overwriting whatever was there previously.

# LOAD (!) name CODE start

loads the bytes name from disk into memory, starting at address start and overwriting whatever was there previously.

# LOAD (!) name CODE

loads the bytes name from disk into memory starting at the address from which the first disk byte was saved and overwriting the bytes that were there in memory before.

3. Arrays:

```
LOAD (!) name DATA letter()
or
LOAD (!) name DATA letter$()
```

deletes any array called letter or letter\$ (as appropriate) and forms a new one from the array stored on disk.

# Error 4 Out of memory occurs if no room for new arrays. Old arrays are not deleted.

MERGE

Loads new information from disk without deleting old information from memory.

1. Program and variables:

MERGE (!)name

merges the program name in with the one already in memory, overwriting any program lines or variables in the old program whose line numbers or names conflict with ones in the new program.

Error 4 Out of memory occurs unless there is enough room in memory for all of the old program and variables and all of the new program and variables being loaded from disk.

1. Bytes:

Not possible

2. Arrays:

Not possible

Exercises

- 1. Make a disk on which the first program, when loaded, prints a menu (a list of some other programs on the disk), asks you to choose a program, and then loads it.
- 2. Get the chess piece graphics from Chapter 14, and then type NEW: they will survive this. However, they will not survive having the computer turned off; if you want to keep then, you must save them on disk, using SAVE with CODE. The easiest way is to save all twenty-one user defined graphics by

## SAVE "chess" CODE USR "a",21\*8

followed by

## VERIFY "chess" CODE

This is the system of bytes saving that was used for saving the picture. The address of the first byte to be saved is USR "a", the address of the first of the
eight bytes that determine the pattern of the first userdefined graphics, and the number of bytes to be saved is 21\*8 - eight bytes for each of 21 graphics.

To load back you would normally use

# LOAD "chess" CODE

However, if you are loading back into a Spectrum with a different amount of memory, or if you have moved the user-defined graphics to a different address (you have to do this deliberately using more advanced techniques), you have to be more careful and use

# LOAD "chess" CODE USR "a"

**USR** allows for the fact that the graphics must be loaded back to a different address.

← Chapter 19

Chapter 21 ➡

## **Chapter 21: The ZX printer**

Summary

### LPRINT, LLIST, COPY

Note: Using the ZX Printer support in conjunction with certain 128k commands can have undersirable results. See the section on BASin and the 128k Spectrum for more details. If you are making use of 128k commands, you should avoid the use of the ZX Printer.

This chapter covers the BASIC statements needed to operate the ZX printer.

BASin emulates the ZX printer: you can view, save, and print the results from the ZX printer output window.

The first two statements, LPRINT and LLIST, are just like PRINT and LIST, except that they use the printer instead of the television. (The L is an historical accident. When BASIC was invented it usually used an electric typewriter instead of a television, so PRINT really did mean print. If you wanted masses of output you would use a very fast line printer attached to the computer, and an LPRINT statement meaning 'Line printer PRINT'.)

Try this program for example.

```
10 LPRINT "This program"
20 LLIST
30 LPRINT "prints out the characte
40 FOR n=32 TO 255
50 LPRINT CHR$ n;
60 NEXT n
```

The third statement, **COPY**, prints out a copy of the television screen. For instance, type **LIST** to get a listing on the screen of the program above, and type

### COPY

Note that **COPY** doesn't work with one of the listings that the computer puts up automatically, because that is cleared whenever a command is obeyed. You must either use **LIST** first, or use **LLIST** and forget about **COPY**.

You can always stop the printer when it is running by pressing the **Esc** key.

If you execute these statements without the printer attached, it should lose all the output and carry on with the next statement.

Try this:

# 10 FOR n=31 TO 3 STEP -1 20 PRINT AT 31-n,n; CHR\$ (CODE "3" 30 NEXT n

You will see a pattern of characters working down diagonally from the top righthand corner until it reaches the bottom of the screen, when the program asks if you want to scroll.

Now change AT 31-n, n in line 20 to TAB n. The program will have exactly the same effect as before.

Now change **PRINT** in line 20 to **LPRINT**. This time there will be no **scroll?**, which should not occur with the printer, and the pattern will carry straight on with the letters F to O.

Now change **TAB n** to **AT 31-n**, **n** still using **LPRINT**. This time you will get just a single line of symbols. The reason for the difference is that the output

from **PRINT** is not printed straight away, but arranges in a buffer store a picture one line long of what the computer will send to the printer when it gets round to it. The printing takes place

- 1. when the buffer is full,
- 2. after an **LPRINT** statement that does not end in a comma or semicolon,
- 3. when a comma, apostrophe or **TAB** item requires a new line, or
- 4. at the end of a program, if there is anything left unprinted.

(iii) explains why our program with TAB works the way it does. As for AT, the line number is ignored and the LPRINT position (like the PRINT position, but for the printer instead of the television) is changed to the column number. An AT item can never cause a line to be sent to the printer.

Exercise

1. Make a printed graph of **SIN** by running the program in Chapter 17 and then using **COPY**.

Chapter 20

Chapter 22 →

## **Chapter 22: Other equipment**

**Note:** BASin does not currently support the hardware and extra keywords listed here.

There is other equipment that you will be able to attach to the Spectrum.

The ZX Microdrive is a high speed mass storage device, and is much more flexible in the way it can be used than a cassette recorder. It will operate not only with SAVE, VERIFY, LOAD and MERGE, but also with PRINT, LIST, INPUT and INKEY\$.

The network is used for connecting several Spectrums so that they can talk to each other - one of the uses of this is that you then need only one Microdrive to serve several computers.

The RS232 interface is a standard connection that allows you to link a Spectrum with keyboards, printers, computers and various other machines even if they were not designed specifically for the Spectrum.

These will use some extra keywords that cannot be used without the extra attachments: they are OPEN #, CLOSE #, MOVE, ERASE, CAT and FORMAT.

← Chapter 21

Chapter 23 ➡

### **Chapter 23: IN and OUT**

Summary

#### OUT, IN

The processor can read from and (at least with RAM) write to memory by using **PEEK** and **POKE**. The processor itself does not really care whether memory is ROM, RAM or even nothing at all; it just knows that there are 65536 memory addresses, and it can read a byte from each one (even if it's nonsense), and write a byte to each one (even if it gets lost). In a completely analogous way there are 65536 of what are called I/O ports (standing for Input/Output ports). These are used by the processor for communicating with things like the keyboard or the printer, and they can be controlled from the BASIC by using the **IN** function and the **OUT** statement.

**IN** is a function like **PEEK**.

**IN** address

It has one argument, the port address, and its result is a byte read from that port.

**OUT** is a statement like **POKE**.

OUT address, value

writes the given value to the port with the given address. How the address is interpreted depends very much on the rest of the computer; quite often, many different addresses will mean the same. On the Spectrum it is most sensible to imagine the address being written in binary, because the individual bits tend to work independently. There are 16 bits, which we shall call (using A for address)

A15, A14, A13, A12, . . . . , A2, A1, A0

Here A0 is the 1s bit, A1 the 2s bit, A2 the 4s bit and so on. Bits A0, A1, A2, A3 and A4 are the important ones. They are normally 1, but if any one of them is 0

this tells the computer to do something specific. The computer cannot cope with more than one thing at a time, so no more than one of these five bits should be 0. Bits A6 and A7 are ignored, so if you are a wizard with electronics you can use them yourself. The best addresses to use are those that are 1 less than a multiple of 32, so that A0,...A4 are all 1. Bits A8, A9 and so on are sometimes used to give extra information.

The byte read or written has 8 bits, and these are often referred to (using D for data) as D7, D6,...., D1, D0. Here is a list of the port addresses used.

There is a set of input addresses that read the keyboard and also the EAR socket.

The keyboard is divided up into 8 half rows of 5 keys each.

- **IN** 65278 reads the half row Caps Shift to V
- **IN** 65022 reads the half row A to G
- **IN 64510** reads the half row Q to T
- **IN** 63486 reads the half row 1 to 5
- **IN 61438** reads the half row O to 6
- IN 57342 reads the half row P to 7
- **IN 49150** reads the half row Enter to H
- **IN 32766** reads the half row Space to B

(These addresses are  $254+256*(255-2\uparrow n)$  as n goes from 0 to 7.)

In the byte read in, bits D0 to D4 stand for the five keys in the given half row - D0 for the outside key, D4 for the one nearest the middle. The bit is 0 if the key is pressed, 1 if it is not. D6 is the value at the EAR socket.

Port address 254 in output drives the loudspeaker (C4) and the MIC socket (D3), and also sets the border colour (D2, D1 and D0).

Port address 251 runs the printer, both in reading and writing: reading finds out whether the printer is ready for more, and writing sends out dots to be printed.

Port addresses 254, 247 and 239 are used for the extra devices mentioned in Chapter 22.

Run this program

# 10 FOR n=0 TO 7: REM half-row number 20 LET a=254+256\*(255-2↑n) 30 PRINT AT 0,0; IN a: GO TO 30

and play around by pressing keys. When you get bored with each half-row, press BREAK and then type

### NEXT n

The control, data and address busses are all exposed at the back of the Spectrum, so you can do almost anything with a Spectrum that you can with a Z80. Sometimes, though, the Spectrum hardware might get in the way. Here is a diagram of the exposed connections at the back:



← Chapter 22

Chapter 24 →

## **Chapter 24: The memory**

Summary

#### CLEAR

Deep inside the computer, everything is stored as bytes, i.e. numbers between 0 and 255. You may think you have stored away the price of wool or the address of your fertilizer suppliers, but it has all been converted into collections of bytes and bytes are what the computer sees.

BASin's debugging tools allow you to work with the memory and system variables.

Each place where a byte can be stored has an address, which is a number between 0 and FFFFh (so an address can be stored as two bytes), so you might think of the memory as a long row of numbered boxes, each of which can contain a byte. Not all the boxes are the same, however. In the standard 16K RAM machine, the boxes from 8000h to FFFFh are simply missing altogether. The boxes from 4000h to 7FFFh are RAM boxes, which means you can open the lid and alter the contents, and those from 0 to 3FFFh are ROM boxes, which have glass tops but cannot be opened. You just have to read whatever was put in them when the computer was made.

		Not	used
Ø	4000h	8000h	FFFFh
10-17-1	=16384	=32768	=65535

To inspect the contents of a box, we use the **PEEK** function: its argument is the address of the box, and its result is the contents. For instance, this program prints out the first 21 bytes in ROM (and their addresses):

## 10 PRINT "Address"; TAB 8; "Byte" 20 FOR a=0 TO 20

# 30 PRINT a; TAB 8; PEEK a 40 NEXT a

All these bytes will probably be quite meaningless to you, but the processor chip understands them to be instructions telling it what to do.

To change the contents of a box (if it is RAM), we use the **POKE** statement. It has the form

POKE address, new contents

where 'address' and 'new contents' stand for numeric expressions. For instance, if you say

#### POKE 31000,57

the byte at address 31000 is given the new value 57 - type

#### PRINT PEEK 31000

to prove this. (Try poking in other values, to show that there is no cheating.) The new value must be between -255 and +255, and if it is negative then 256 is added to it.

The ability to poke gives you immense power over the computer if you know how to wield it, and immense destructive possibilities if you don't. It is very easy, by poking the wrong value in the wrong address, to lose vast programs that took you hours to type in. Fortunately, you won't do the computer any permanent damage.

We shall now take a more detailed look at how the RAM is used, but don't bother to read this unless you're interested.

The memory is divided into different areas (shown on the big diagram) for storing different kinds of information. The areas are only large enough for the information that they actually contain, and if you insert some more at a given point (for instance by adding a program line or variable) space is made by shifting up everything above that point. Conversely, if you delete information then everything is shifted down.

The display file stores the television picture. It is rather curiously laid out, so you probably won't want to PEEK or POKE in it. Each character position on the screen has an 8×8 square of dots, and each dot can be either 0 (paper) or 1 (ink) and by using binary notation we can store the pattern as 8 bytes, one for each row. However, these 8 bytes are not stored together. The corresponding rows in the 32 characters of a single line are stored together as a scan of 32 bytes, because this is what the electron beam in the television needs as it scans from the left hand side of the screen to the other. Since the complete picture has 24 lines of 8 scans each, you might expect the total of 172 scans to be stored in order, one after the other; you'd be wrong. First come the top scans of lines 0 to 7, then the next scans of lines 0 to 7, and so on to the bottom scans of lines 0 to 7; then the same for lines 8 to 15; and then the same for lines 16 to 23. The upshot of all this is that if you're used to a computer that uses PEEK and POKE on the screen, you'll have to start using SCREEN\$ and PRINT AT instead, or PLOT and POINT.

The attributes are the colours and so on for each character position, using the format of ATTR. These are stored line by line in the order you'd expect.

The printer buffer stores the characters destined for the printer.

The system variables contain various pieces of information that tell the computer what sort of state the computer is in. They are listed fully in the next chapter, but for the moment note that there are some (called CHANS, PROG, VARS, E LINE and so on) that contain the addresses of the boundaries between the various areas in memory. These are not BASIC variables, and their names will not be recognized by the computer.



The Microdrive maps are only used with the Microdrive. Normally there is nothing there.

The channel information contains information about the input and output devices, namely the keyboard (with the lower half of the screen), the upper half of the screen, and the printer.

Each line of BASIC program has the form:



Note that, in contrast with all other cases of two byte numbers in the Z80, the line number here is stored with its more significant byte first: that is to say, in the order that you write them down in.

A numerical constant in the program is followed by its binary form, using the character CHR\$ 14 followed by five bytes for the number itself.

The variables have different formats according to their different features. The letters in the names should be imagined as starting off in lower case.



The order of the elements is:

first, the elements for which the first subscript is 1; next, the elements for which the first subscript is 2; next, the elements for which the first subscript is 3; and so on for all possible values of the first subscript.

The elements with a given first subscript are ordered in the same way using the second subscript, and so on down to the last.

As an example, the elements of the 3\*6 array b in Chapter 12 are stored in the order b(1,1) b(1,2) b(1,3) b(1,4) b(1,5) b(1,6) b(2,1) b(2,2) ... b(2,6) b(3,1) b(3,2) ... b(3,6).

Contro	Less significant byte						
		5 bytes	5 bytes	5 bytes	2 bytes	1 byte	
Lette	, r-60h	Value	Limit	Step	Looping line	Statement number within line	



The calculator is the part of the BASIC system that deals with arithmetic, and the numbers on which it is operating are held mostly in the calculator stack.

The spare part contains the space so far unused.

The machine stack is the stack used by the Z80 processor to hold return addresses and so on.

The **GO SUB** stack was mentioned in Chapter 5.

The byte pointed to by RAMTOP has the highest address used by the BASIC system. Even NEW, which clears the RAM out, only does so as far as this - so it doesn't change the user defined graphics. You can change the address RAMTOP by putting a number in a CLEAR statement:

CLEAR new RAMTOP

This

- 1. clears out all the variables
- 2. clears the display file (like **CLS**)
- 3. resets the **PLOT** position to the bottom left hand

corner

4. clears the **GO SUB** stack and puts it at the new RAMTOP, assuming that this lies between the calculator stack and the physical end of RAM; otherwise it leaves RAMTOP as it was.

**RUN** also does **CLEAR**, although it never changes RAMTOP.

Using **CLEAR** in this way, you can either move RAMTOP up to make more room for the BASIC by overwriting the user defined graphics, or you can move it down to make more RAM that is preserved from **NEW**.

Type NEW, then CLEAR 23800 to get some idea of what happens to the machine when it fills up.

One of the first things you will notice if you start typing in a program is that after a while the computer stops accepting any more input. It means the computer is chock a block and you will have to empty it slightly. There are two error messages with roughly the same meaning, 4 Out of memory and G No room for line.

You can adjust the length of the error beep by poking a number into address 23608. The usual length has number 64.

Any number (except 0) can be written uniquely as  $\pm$  m × 2<sup>e</sup> where  $\pm$  is the sign, m is the mantissa, and lies between 0.5 and 1 (it cannot be 1), and e is the exponent, a whole number (possibly negative).

Suppose you write m in the binary scale. Because it is a fraction, it will have a binary point (like the decimal point in the scale of ten) and then a binary fraction (like a decimal fraction); so in binary: a half is written .1 a quarter is written .01 three quarters is written .11 a tenth is written .000110011001100110011 ... and so on.

With our number m, because it is less than 1, there are no bits before the binary point, and because it is at least 0.5, the bit immediately after the binary point is a 1.

To store the number in the computer, we use five bytes, as follows:

- 1. write the first eight bits of the mantissa in the second byte (we know that the first bit is 1), the second eight bits in the third byte, the third eight bits in the fourth byte and the fourth eight bits in the fifth byte;
- 2. replace the first bit in the second byte which we know is 1 by the sign: 0 for plus, 1 for minus;
- 3. write the exponent +128 in the first byte. For instance, suppose our number is 1/10: 1/10 = 4/5\*2-3

Thus the mantissa m is .11001100110011001100110011001100 in binary (since the 33rd bit is 1, we shall round the 32nd up from 0 to 1), and the exponent e is 3.

Applying our three rules gives the five bytes: [missing?]

There is an alternative way of storing whole numbers between -65535 and +65535:

- 1. the first byte is 0.
- 2. the second byte is 0 for a positive number, FFh for a negative one,
- 3. the third and fourth bytes are the less and more significant bytes of the number (or the number

+131072 if it is negative),4. the fifth byte is 0.

← Chapter 23

Chapter 25 ➡

### **Chapter 25: The system variables**

The bytes in memory from 23552 to 23733 are set aside for specific uses by the system. You can peek them to find out various things about the system, and some of them can be usefully poked. They are listed here with their uses.

These are called system variables, and have names, but do not confuse them with the variables used by the BASIC. The computer will not recognize the names as referring to system variables, and they are given solely as mnemonics for we humans.

Poking system variables in the BASin editor will not always have the desired effect, because there is a layer of abstraction between BASin and the emulated ZX Spectrum memory.

The abbreviations in column 1 have the following meanings:

```
X The variables should not be poked because the system might crash.
```

N Poking the variable will have no lasting effect.

The number in column 1 is the number of bytes in the variable. For two bytes, the first one is the less significant byte, the reverse of what you might expect. So to poke a value v to a two byte variable at address n, use

## POKE n,v-256\*INT (v/256) POKE n+1,INT (v/256)

and to peek its value, use the expression

```
PEEK n+256*PEEK (n+1)
```

Notes	Address	Name	Contents
N8	23552	KSTATE	Used in reading the keyboard.
N1	23560	LAST K	Stores newly pressed key.
1	23561	REPDEL	Time (in 50ths of a second, in 60ths of a second in N. America) that a key must be held down before it repeats. This starts off at 35, but you can <b>POKE</b> in other values.
1	23562	REPPER	Delay (in 50ths of a second, in 60ths of a second in N. America) between successive repeats of a key held down: initially 5.
N2	23563	DEFADD	Address of arguments of user defined function if one is being evaluated; otherwise 0.
Nl	23565	K DATA	Stores 2nd byte of colour controls entered from keyboard.
N2	23566	TVDATA	Stores bytes of coiour, AT and TAB controls going to television.
X38	23568	STRMS	Addresses of channels attached to streams.
2	23606	CHARS	256 less than address of character set (which starts with space and carries on to the copyright symbol). Normally in ROM, but you can set up your own in RAM and make CHARS point to it.
1	23608	RASP	Length of warning buzz.
1	23609	PIP	Length of keyboard click.
1	23610	ERR NR	1 less than the report code. Starts off at 255 (for 1) so <b>PEEK 23610</b> gives 255.
X1	23611	FLAGS	Various flags to control the BASIC system.
X1	23612	TV FLAG	Flags associated with the television.

X2	23613	ERR SP	Address of item on machine stack to be used as error return.
N2	23615	LIST SP	Address of return address from automatic listing.
N1	23617	MODE	Specifies K, L, C, E or G cursor.
2	23618	NEWPPC	Line to be jumped to.
	23620	NSPPC	Statement number in line to be jumped to. Poking first NEWPPC and then NSPPC forces a jump to a specified statement in a line.
2	23621	PPC	Line number of statement currently being executed.
1	23623	SUBPPC	Number within line of statement being executed.
1	23624	BORDCR	Border colour * 8; also contains the attributes normally used for the lower half of the screen.
2	23625	E PPC	Number of current line (with program cursor).
X2	23627	VARS	Address of variables.
N2	23629	DEST	Address of variable in assignment.
X2	23631	CHANS	Address of channel data.
X2	23633	CURCHL	Address of information currently being used for input and output.
X2	23635	PROG	Address of BASIC program.
X2	23637	NXTLIN	Address of next line in program.
X2	23639	DATADD	Address of terminator of last DATA item.

X2	23641	E LINE	Address of command being typed in.
2	23643	K CUR	Address of cursor.
X2	23645	CH ADD	Address of the next character to be interpreted: the character after the argument of <b>PEEK</b> , or the newline at the end of a <b>POKE</b> statement.
2	23647	X PTR	Address of the character after the <b>?</b> syntax error marker in traditional 48K BASIC.
X2	23649	WORKSP	Address of temporary work space.
X2	23651	STKBOT	Address of bottom of calculator stack.
X2	23653	STKEND	Address of start of spare space.
N1	23655	BREG	Calculator's b register.
N2	23656	MEM	Address of area used for calculator's memory. (Usually MEMBOT, but not always.)
	23658	FLAGS2	More flags.
X1	23659	DF SZ	The number of lines (including one blank line) in the lower part of the screen.
2	23660	S TOP	The number of the top program line in automatic listings.
2	23662	OLDPPC	Line number to which <b>CONTINUE</b> jumps.
	23664	OSPCC	Number within line of statement to which <b>CONTINUE</b> jumps.
N1	23665	FLAGX	Various flags.
N2	23666	STRLEN	Length of string type destination in assignment.

N2	23668	T ADDR	Address of next item in syntax table (very unlikely to be useful).
2	23670	SEED	The seed for <b>RND</b> . This is the variable that is set by <b>RANDOMIZE</b> .
3	23672	FRAMES	3 byte (least significant first), frame counter. Incremented every 20ms. See Chapter 18.
2	23675	UDG	Address of 1st user defined graphic. You can change this for instance to save space by having fewer user defined graphics.
1	23677	COORDS	x-coordinate of last point plotted.
1	23678		y-coordinate of last point plotted.
1	23679	P POSN	33 column number of printer position.
X2	23680	PR CC	Full address of next position for <b>LPRINT</b> to print at (in printer buffer).
2	23682	ECHO E	33 column number and 24 line number (in lower half) of end of input buffer.
2	23684	DF CC	Address in display file of <b>PRINT</b> position.
2	23686	DFCCL	Like DF CC for lower part of screen.
X1	23688	S POSN	33 column number for <b>PRINT</b> position.
X1	23689		24 line number for <b>PRINT</b> position.
X2	23690	SPOSNL	Like S POSN for lower part.
	23692	SCR CT	Counts scrolls: it is always 1 more than the number of scrolls that will be done before stopping with SCroll? If you keep poking this with a number bigger than 1 (say 255), the screen will scroll on and on without asking you.

1	23693	ATTR P	Permanent current colours, etc (as set up by colour statements).
1	23694	MASK P	Used for transparent colours, etc. Any bit that is 1 shows that the corresponding attribute bit is taken not from ATTR P, but from what is already on the screen.
N1	23695	ATTR T	Temporary current colours, etc (as set up by colour items).
N1	23696	MASK T	Like MASK P, but temporary.
	23697	P FLAG	More flags.
N30	23698	MEMBOT	Calculator's memory area; used to store numbers that cannot conveniently be put on the calculator stack.
2	23728		Used for the NMI in the +3 and +2A, and by the Interface 1.
2	23730	RAMTOP	Address of last byte of BASIC system area.
2	23732	P RAMT	Address of last byte of physical RAM.

This program tells you the first 22 bytes of the variables area:

# 10 FOR n=0 TO 21 20 PRINT PEEK (PEEK 23627+256\*PEEK 30 NEXT n

Try to match up the control variable n with the descriptions above. Now change line 20 to

### 20 PRINT PEEK (23755+n)

This tells you the first 22 bytes of the program area. Match these up with the

program itself.

← Chapter 24

Chapter 26 →

## **Chapter 26: Using machine code**

Summary

**USR** with numeric argument

This chapter is written for those who understand Z80 machine code, the set of instructions that the Z80 processor chip uses. If you do not, but would like to, there are plenty of books about it. You want to get one called something along the lines of "Z80 machine code [or assembly language] for the absolute beginner", and if it mentions the Spectrum, so much the better.

These programs are normally written in assembly language, which, although cryptic, are not too difficult to understand with practice. (You can see the assembly language instructions in Appendix A.) However, to run them on the computer you need to code the program into a sequence of bytes; in this form it is called machine code. This translation is usually done by the computer itself, using a program called an assembler. There is no assembler built in to the Spectrum, but you may well be able to obtain one. Failing that, you will have to do the translation yourself, provided that the program is not too long.

Let's take as an example the program

## ld bc, 99 ret

which loads the bc register pair with 99. This translates into the four machine code bytes 1, 99, 0 (for ld bc, 99) and 201 (for ret). (If you look up 1 and 201 in Appendix A, you will find ld bc, NN - where NN stands for any two byte number - and ret.)

When you have got your machine code program, the next step is to get it into the computer. (An assembler would probably do this automatically.) You need to decide whereabouts in memory to put it, and the best thing is to make extra

space for it between the BASIC area and the user defined graphics.

Suppose, for instance, that you have a 16K Spectrum. To start off with, the top end of RAM has



If you type

### CLEAR 32499

this will give you a space of 100 (for good measure) bytes starting at address 32500.



To put in the machine code program, you would run a BASIC program something like

```
10 LET a=32500
20 READ n:
    POKE a,n
30 LET a=a+1:
    GO TO 20
40 DATA 1,99,0,201
```

(This will stop with report E Out of DATA when it has filled in the four bytes you specified.)

To run the machine code, you use the function **USR** but this time with a numeric argument, the starting address. Its result is the value of the bc register on return from the machine code program, so if you do

#### PRINT USR 32500

you get the answer 99.

The return address to the BASIC is stacked in the usual way, so return is by a Z80 ret instruction. You should not use the iy and i registers in a machine code routine.

You can save your machine code program easily enough with

### SAVE "some name" CODE 32500,4

On the face of it, there is no way of saving it so that when loaded it automatically runs itself, but you can get round this by using a BASIC program.

## 10 LOAD "xxxx" CODE 32500,4 20 PRINT USR 32500

Do first

SAVE "some name" LINE 0

and then

SAVE "xxxx" CODE 32500,4

LOAD "some name"

will then load and automatically run the BASIC program, and the BASIC program will load and run the machine code.

← Chapter 25

Appendix A  $\Rightarrow$ 

### **Appendix A: The character set**

This is the complete Spectrum character set, with codes in decimal and hex. If one imagines the codes as being Z80 machine code instructions, then the right hand columns give the corresponding assembly language mnemonics. As you are probably aware if you understand these things, certain Z80 instructions are compounds starting with CBh or EDh; the two right hand columns give these.

-		-			-
Code	Character	Hex	Z80 Assembler	after CB	after ED
0	Not used	00	nop	rlc b	
1	Not used	01	ld bc,NN	rlc c	
2	Not used	02	ld (bc),a	rlc d	
3	Not used	03	inc bc	rlc e	
4	Not used	04	inc b	rlc h	
5	Not used	05	dec b	rlc l	
6	PRINT comma	06	ld b,N	rlc (hl)	
7	Edit	07	rlca	rlc a	
8	÷	08	ex af,af'	rrc b	
9	<b>→</b>	09	add hl,bc	rrc c	
10	+	0A	ld a,(bc)	rrc d	

11	<b>†</b>	0B	dec bc	rrc e	
12	Backspace	0C	incc	rrch	
13	Enter	0D	dec c	rrc	
14	number	0E	ld c,N	rrc (hl)	
15	Not used	0F	rrca	rrc a	
16	INK control	10	djnz DIS	rl b	
17	PAPER control	11	ld de,NN	rlc	
18	FLASH control	12	ld (de),a	rl d	
19	BRIGHT control	13	inc de	rl e	
20	INVERSE control	14	inc d	rl h	
21	OVER control	15	dec d	rl l	
22	AT control	16	ld d,N	rl (hl)	
23	TAB control	17	rla	rl a	
24	Not used	18	jr DIS	rr b	
25	Not used	19	add hl,de	rr c	
26	Not used	1A	ld a,(de)	rr d	

27	Not used	1B	dec de	rr e	
28	Not used	1C	inc e	rr h	
29	Not used	1D	dec e	rr l	
30	Not used	1E	ld e,N	rr (hl)	
31	Not used	1F	rra	rr a	
32	Space	20	jr nz, DIS	sla b	
33	!	21	ld hl,NN	sla c	
34	Π	22	ld (NN),hl	sla d	
35	#	23	inc hl	sla e	
36	\$	24	inc h	sla h	
37	%	25	dec h	sla l	
38	&	26	ld h,N	sla (hl)	
39	,	27	daa	sla a	
40	(	28	jr z,DIS	sra b	
41	)	29	add hl,hl	sra c	
42	*	2A	ld hl,(NN)sra d		
43	+	2B	dec hl	sra e	

44	,	2C	inc ll	sra h	
45	-	2D	dec l	sra l	
46		2F	ld l,N	sra (hl)	
47	/	2F	cpl	sra a	
48	0	30	jr nc,DIS		
49	1	31	ld sp,NN		
50	2	32	ld (NN),a		
51	3	33	inc sp		
52	4	34	inc (hl)		
53	5	35	dec (hl)		
54	6	36	ld (hl),N		
55	7	37	scf		
56	8	38	lr c,DIS	srl b	
57	9	39	add hl,sp	srl c	
58	:	3A	ld a,(NN)	srl d	
59	;	3B	dec sp	srl e	
60	<	3C	inc a	srl h	

61	=	3D	dec a	srl l	
62	>	3E	ld a,N	srl (hl)	
63	?	3F	ccf	srl a	
64	@	40	ld b,b	bit 0,b	in b,(c)
65	А	41	ld b,c	bit 0,c	out (c),b
66	В	42	ld b,d	bit 0,d	sbc hl,bc
67	С	43	ld b,e	bit 0,e	ld (NN),bc
68	D	44	ld b,h	bit 0,h	neg
69	E	45	ld b,l	bit 0,1	retn
70	F	46	ld b,(hl)	bit 0,(hl)	im 0
71	G	47	ld b,a	bit 0,a	ld i,a
72	Н	48	ld c,b	bit 1,b	in c,(c)
73	Ι	49	ld cc	bit 1,c	out (c),c
74	J	4A	ld c,d	bit i,d	adc hl,bc
75	К	4B	ld c,e	bit 1,e	ld bc,(NN)
76	L	4C	ld c,h	bit 1,h	

77	М	4D	ld c,l	bit 1,l	reti
78	Ν	4E	ld c,(hl)	bit 1,(hl)	
79	0	4F	ld c,a	bit 1,a	ld r,a
80	Р	50	ld d,b	bit 2,b	in d,(c)
81	Q	51	ld d,c	bit 2,c	out (c),d
82	R	52	ld d,d	bit 2,d	sbc hl,de
83	S	53	ld d,e	bit 2,e	ld (NN),de
84	Т	54	ld d,h	bit 2,h	
85	U	55	ld d,l	bit 2,l	
86	V	56	ld d,(hl)	bit 2,(hl)	im 1
87	W	57	ld d,a	bit 2,a	ld a,i
88	Х	58	ld e,b	bit 3,b	in e,(c)
89	Y	59	ld e,c	bit 3,c	out (c),e
90	Z	5A	ld e,d	bit 3,d	adc hl,de
91	[	5B	ld e,e	bit 3,e	ld de,(NN)
92	/	5C	ld e,h	bit 3,h	
93	]	5D	ld e,l	bit 3,1	

94	^	5E	ld e,(hl)	bit 3,(hl)	im 2
95	-	5F	ld e,a	bit 3,a	ld a,r
96	£	60	ld h,b	bit 4,b	in h,(c)
97	а	61	ld h,c	bit 4,c	out (c),h
98	Ь	62	ld h,d	bit 4,d	sbc hl,hl
99	с	63	ld h,e	bit 4,e	ld (NN),hl
100	d	64	ld h,h	bit 4,h	
101	e	65	ld h,l	bit 4,1	
102	f	66	ld h,(hl)	bit 4,(hl)	
103	g	67	ld h,a	bit 4,a	rrd
104	h	68	ld l,b	bit 5,b	in l,(c)
105	i	69	ld l,c	bit 5,c	out (c),l
106	j	6A	ld l,d	bit 5,d	adc hl,hl
107	k	6B	ld l,e	bit 5,e	ld hl,(NN),sp
108	1	6C	ld l,h	bit 5,h	
109	m	6D	ld 1,1	bit 5,l	
110	n	6E	ld l,(hl)	bit 5,(hl)	
111	0	6F	ld l,a	bit 5,a	rld
-----	---	----	-----------	------------	------------
112	р	70	ld (hl),b	bit 6,b	in f,(c)
113	q	71	ld (hl),c	bit 6,c	
114	r	72	ld (hl),d	bit 6,d	sbc hl,sp
115	s	73	ld (hl),e	bit 6,e	ld (NN),sp
116	t	74	ld (hl),h	bit 6,h	
117	u	75	ld (hl),l	bit 6,l	
118	v	76	halt	bit 6,(hl)	
119	w	77	ld (hl),a	bit 6,a	
120	x	78	ld a,b	bit 7,b	in a,(c)
121	у	79	ld a,c	bit 7,c	out (c),a
122	z	7A	ld a,d	bit 7,d	adc hl,sp
123	{	7B	ld a,e	bit 7,e	ld sp,(NN)
124		7C	lda,h	bit 7,h	
125	}	7D	ld al	bit 7,l	
126	-	7E	ld a,(hl)	bit 7,(hl)	
	İ				

127	©	7F	ld a,a	bit 7,a	
128		80	add a,b	res 0,b	
129	•	81	add a,c	res 0,c	
130	•	82	add a,d	res 0,d	
131		83	add a,e	res 0,e	
132	•	84	add a,h	res 0,h	
133		85	add a,l	res 0,1	
134	•	86	add a,(hl)	res 0,(hl)	
135	7	87	add a,a	res 0,a	
136	•	88	adc a,b	res 1,b	
137	•	89	adc a,c	res 1,c	
138	B	8A	adc a,d	res 1,d	
139	P	8B	adc a,e	res 1,e	
140	-	8C	adc a,h	res 1,h	
141	J	8D	adc a,l	res 1,i	
142	L	8E	adc a,(hl)	res 1,(hl)	
143		8F	adc a,a	res 1,a	

144	(a)	90	sub b	res 2,b		
145	(b)	91	sub c	res 2,c		
146	(c)	92	sub d	res 2,d		
147	(d)	93	sub e	res 2,e		
148	(e)	94	sub h	res 2,h		
149	(f)	95	sub l	res 2,l		
150	(g)	96	sub (hl)	res 2,(hl)		
151	(h)	97	sub a	res 2,a		
152	(i)	98	sbc a,b	res 3,b		
153	(j)	99	sbc a,c	res 3,c		
154	(k)	user	9A	sbc a,d	res 3,d	
155	(1)	graphics	9B	sbc a,e	res 3,e	
156	(m)	9C	sbc a,h	res 3,h		
157	(n)	9D	sbc a,l	res 3,1		
158	(0)	9E	sbc a,(hl)	res 3,(hl)		
159	(p)	9F	sbc a,a	res 3,a		
160	(q)	A0	and b	res 4,b	ld	

161	(r)	A1	and c	res 4,c	срі	
162	(s)	A2	and d	res 4,d	ini	
163	(t)	A3	and e	res 4,e	outi	
164	(u)	A4	and h	res 4,h		
165	RND	A5	and l	res 4,1		
166	INKEY\$	A6	and (hl)	res 4,(hl)		
167	PI	A7	and a	res 4,a		
168	FN	A8	xor b	res 5,b	ldd	
169	POINT	A9	xor c	res 5,c	cpd	
170	SCREEN\$	AA	xor d	res 5,d	ind	
171	ATTR	AB	xor e	res 5,e	outd	
172	AT	AC	xor h	res 5,h		•
173	ТАВ	AD	xor l	res 5,i		
174	VAL\$	AE	xor (hl)	res 5,(hl)		
175	CODE	AF	xor a	res 5,a		
176	VAL	В0	or b	res 6,b	ldir	

177	LEN	B1	or c	res 6,c	cpir
178	SIN	B2	or d	res 6,d	inir
179	COS	В3	or e	res 6,e	otir
180	TAN	B4	or h	res 6,h	
181	ASN	В5	or l	res 6,l	
182	ACS	B6	or (hl)	res 6,(hl)	
183	ATN	B7	or a	res 6,a	
184	LN	В8	cp b	res 7,b	lddr
185	EXP	В9	cp c	res 7,c	cpdr
186	INT	ВА	cp d	res 7,d	indr
187	SQR	BB	cp e	res 7,e	otdr
188	SGN	BC	cp h	res 7,h	
189	ABS	BD	cp l	res 7,l	
190	PEEK	BE	cp (hl)	res 7,(hl)	
191	IN	BF	cp a	res 7,a	

192	USR	C0	ret nz	set 0,b
193	STR\$	C1	pop bc	set 0,c
194	CHR\$	C2	jp nz,NN	set 0,d
195	NOT	C3	jp NN	set 0,e
196	BIN	C4	call nz,NN	set 0,h
197	OR	C5	push bc	set 0,1
198	AND	C6	add a,N	set 0,(hl)
199	<=	C7	rst 0	set 0,a
200	>=	C8	ret z	set 1,b
201	$\diamond$	С9	ret	set l,c
202	LINE	CA	jp z,NN	set l,d
203	THEN	СВ	set l,e	
204	то	СС	call z,NN	set l,h
205	STEP	CD	call NN	set 1,l
206	DEF FN	CE	adc a,N	set 1,(hl)
207	САТ	CF	rst B	set 1,a

208	FORMAT	D0	ret nc	set 2,b
209	MOVE	D1	pop de	set 2,c
210	ERASE	D2	jpnc,NN	set 2,d
211	OPEN #	D3	out (N),a	set 2,e
212	CLOSE #	D4	call nc,NN	set 2,h
213	MERGE	D5	push de	set 2,l
214	VERIFY	D6	sub N	set 2,(hl)
215	BEEP	D7	rst 16	set 2,a
216	CIRCLE	D8	ret c	set 3,b
217	INK	D9	exx	set 3,c
218	PAPER	DA	jpc,NN	set 3,d
219	FLASH	DB	in a,(N)	set 3,e
220	BRIGHT	DC	call c,NN	set 3,h
221	INVERSE	DD	[prefixes instructions using ix]	set 3,1
222	OVER	DE	sbc a,N	set 3,(hl)

2	223	OUT	DF	rst 24	set 3,a
2	224	LPRINT	E0	ret po	set 4,b
2	225	LLIST	E1	pop hl	set 4,c
2	226	STOP	E2	jp po,NN	set 4,d
2	227	READ	E3	ex (sp),hl	set 4,e
2	228	DATA	E4	call po,NN	set 4,h
2	229	RESTORE	E5	push hl	set 4,l
2	230	NEW	E6	and N	set 4,(hl)
2	231	BORDER	E7	rst 32	set 4,a
2	232	CONTINUE	E8	ret pe	set 5,b
2	233	DIM	E9	jp (hl)	set 5,c
2	234	REM	EA	jp pe,NN	set 5,d
2	235	FOR	EB	ex de,hl	set 5,e
2	236	G0 T0	EC	call pe,NN	set 5,h
2	237	GO SUB	ED		set 5,l
2	238	INPUT	EE	xor N	set 5,(hl)

239	LOAD	EF	rst 40	set 5,a		
240	LIST	F0	ret p	set 6,b		
241	LET	F1	pop af	set 6,c		
242	PAUSE	F2	jp p,NN	set 6,d		
243	NEXT	F3	di	set 6,e		
244	POKE	F4	call p,NN	set 6,h		
245	PRINT	F5	push af	set 6,1		
246	PLOT	F6	or N	set 6,(hl)		
247	RUN	F7	rst 48	set 6,a		
248	SAVE	F8	ret m	set 7,b		
249	RANDOMIZE	F9	ld sp,hl	set 7,c		_
250	IF		FA	jp m,NN	set 7,d	
251	CLS	FB	ei	set 7,e		
252	DRAW	FC	call m,NN	set 7,h		
253	CLEAR	FD	[prefixes instructions using iy]	set 7,l		

254	RETURN	FE	ср N	set 7,(hl)	_
255	СОРҮ	FF	rst 56	set 7,a	ſ

← Chapter 26

Appendix B  $\Rightarrow$ 

## **Appendix B: Reports**

These appear in the Error window whenever the computer stops executing some BASIC, and explain why it stopped, whether for a natural reason, or because an error occurred.

The report has a code number or letter so that you can refer to the table here, a brief message explaining what happened and the line number and statement number within that line where it stopped. (A command is shown as line 0. Within a line, statement 1 is at the beginning, statement 2 comes after the first colon or THEN, and so on.)

The behaviour of **CONTINUE** depends very much on the reports. Normally, **CONTINUE** goes to the line and statement specified in the last report, but there are exceptions with reports 0, 9 and D (also see Appendix C).

Here is a table showing all the reports. It also tells you in what circumstances the report can occur, and this refers you to Appendix C. For instance error A Invalid argument can occur with SQR, IN, ACS and ASN and the entries for these in Appendix C tell you exactly what arguments are invalid.

Code	Meaning	Situations
0	ОК	Any Successful completion, or jump to a line number bigger than any existing. This report does not change the line and statement jumped to by <b>CONTINUE</b> .
1	NEXT without FOR	<b>NEXT</b> The control variable does not exist (it has not been set up by a <b>FOR</b> statement), but there is an ordinary variable with the same name.
2	Variable not found	Any For a simple variable this will happen if the variable is used before it has been assigned to in a LET, READ or INPUT statement or loaded from tape or set up in a FOR statement. For a subscripted variable it will happen if the variable is used before it has been dimensioned in a DIM statement or loaded from tape.

3	Subscript wrong	Subscripted variables, Substrings A subscript is beyond the dimension of the array, or there are the wrong number of subscripts. If the subscript is negative or bigger than 65535, then error B will result.
4	Out of memory	LET, INPUT, FOR, DIM, GO SUB, LOAD, MERGE. Sometimes during expression evaluation, there is not enough room in the computer for what you are trying to do. If the computer really seems to be stuck in this state, you may have to clear out the command line using Backspace and then delete a program line or two (with the intention of putting them back afterwards) to give yourself room to manoeuvre with - say - CLEAR.
5	Out of screen	<b>INPUT, PRINT AT</b> An <b>INPUT</b> statement has tried to generate more than 23 lines in the lower half of the screen. Also occurs with <b>PRINT AT 22</b> ,
6	Number too big	Any arithmetic Calculations have led to a number greater than about 10 <sup>38</sup> .
7	RETURN without GO SUB	<b>RETURN</b> There has been one more <b>RETURN</b> than there were <b>GO SUB</b> s.
8	End of file	Microdrive, etc operations
9	STOP statement	STOP After this, CONTINUE will not repeat the STOP, but carries on with the statement after.
A	Invalid argument	SQR, LN, ASN, ACS, USR (with string argument) The argument for a function is no good for some reason.
В	Integer out of range	RUN, RANDOMIZE, POKE, DIM, GO TO, GO SUB, LIST, LLIST, PAUSE, PLOT, CHR\$, PEEK, USR (with numeric argument), Array access When an integer is required, the floating point argument is rounded to the nearest integer. If this is outside a suitable range then error B results. For array access, see also error 3.
С	Nonsense in BASIC	VAL, VAL\$ The text of the (string) argument does not form a valid expression.
		LOAD, SAVE, VERIFY, MERGE, LPRINT, LLIST, COPY.

D	BREAK - CONT repeats	Also when the computer asks <b>SCrOll?</b> and you type N, Space or <b>STOP</b> . Break was pressed during some peripheral operation. The behaviour of <b>CONTINUE</b> after this report is normal in that it repeats the statement. Compare with report L.
E	Out of DATA	READ You have tried to READ past the end of the DATA list.
F	Invalid file name	SAVE SAVE with name empty or longer than 10 characters (on an original Spectrum).
G	No room for line	Entering a line into the program There is not enough room left in memory to accommodate the new program line.
н	STOP in INPUT	<b>INPUT</b> Some <b>INPUT</b> data started with <b>STOP</b> , or - for <b>INPUT LINE</b> - ♥ was pressed. Unlike the case with report 9, after report H <b>CONTINUE</b> will behave normally, by repeating the <b>INPUT</b> statement.
I	FOR without NEXT	<b>FOR</b> There was a <b>FOR</b> loop to be executed no times (e.g. <b>FOR n=1 TO 0</b> ) and the corresponding <b>NEXT</b> statement could not be found.
J	Invalid I/O device	Microdrive, etc operations
К	Invalid colour	<b>INK, PAPER, BORDER, FLASH, BRIGHT, INVERSE, OVER</b> ; also after one of the corresponding control characters. The number specified is not an appropriate value.
L	BREAK into program	Any Break pressed. This is detected between two statements. The line and statement number in the report refer to the statement before Break was pressed, but <b>CONTINUE</b> goes to the statement after (allowing for any jumps to be done), so it does not repeat any statements.
М	RAMTOP no good	CLEAR; possibly in RUN The number specified for RAMTOP is either too big or too small.
N	Statement lost	RETURN, NEXT, CONTINUE Jump to a statement that no longer exists.

0	O Invalid stream Microdrive, etc operations		
Р	FN without DEF	<b>FN</b> A user-defined function was called but no corresponding definition could be found.	
QParameter errorFN Wrong number of arguments, or one of them is the wrong type (string in vice versa).		<b>FN</b> Wrong number of arguments, or one of them is the wrong type (string instead of number or vice versa).	
R	File loading error	VERIFY, LOAD or MERGE A file on disk could not be read in, or would not verify.	

The following reports are 128k only, and will only appear when using one of the 128k commands. They are not common to the 48k Spectrum, hence their code appearing in reports in lower case.

Code	Meaning	Situations
a	MERGE error	MERGE ! MERGE ! would not execute for some reason - either size or file type wrong.
b	Wrong file type	MERGE !, LOAD ! A file of inappropriate type was specified during silicon disc operation, for instance a CODE file in LOAD ! "name".
с	CODE error	LOAD ! "file" CODE The size of the file would overrun the top of memory.
d	Too many brackets	<b>PLAY</b> Too many brackets around a repeated phrase in one of the arguments.
e	File already exists	SAVE ! The file name specified has already been used.
f	Invalid name	ERASE !

		The file name specified is empty or above 10 characters in length.
h	File does not exist	<b>LOAD !</b> , <b>ERASE !</b> There is no file in the silicon disc that has the name specified.
i	Invalid device	<b>FORMAT</b> The device name following the FORMAT command does not exist or orrespond to a physical device.
j	Invalid baud rate	FORMAT The baud rate for the RS232 was set to zero.
k	Invalid note name	<b>PLAY</b> PLAY came across a note or command it didn't recognise, or a command that was in lower case.
1	Number too big	PLAY A parameter for a command is an order of magnitude too big.
m	Note out of range	<b>PLAY</b> A series of sharps or flats has taken a note beyond the range of the sound chip.
n	Out of range	<b>PLAY</b> A parameter for a command is too big or too small. If the error is very large, error "l" results.
0	Too many tied notes	PLAY An attempt was made to tie too many notes together.

← Appendix A

Appendix C  $\Rightarrow$ 

# **Appendix C**

Part 1: A description of the ZX Spectrum for reference

In the original paper version of this manual, the first section of this appendix was a repeat of that part of Chapter 1 concerning the keyboard and display.

The display window

Each character position has attributes specifying its paper (background) and ink (foreground) colours, a two-level brightness, and whether it flashes or not. The available colours are black, blue, red, magenta, green, yellow and white.

The edge of the screen can be set to any of the colours using the border statement.

A character position is divided into 8×8 pixels and high resolution graphics are obtained by setting the pixels individually to show either the ink or paper colour for that character position.

The attributes at a character position are adjusted whenever a character is written there or a pixel is plotted. The exact manner of the adjustment is determined by the pnnting parameters. of which there are two sets (called permanent and temporary) of six: the PAPER, INK, FLASH, BRIGHT, INVERSE and OVER parameters. Permanent parameters for the top part are set up by PAPER, INK, etc, statements, and last until further notice. (Initially they are black ink on white paper with normal brightness, no flashing, normal video and no overprinting). Permanent parameters for the bottom part use the border colour as the paper colour, with a black or white contrasting ink colour, normal brightness, no flashing, normal video and no overprinting.

Temporary parameters are set up by PAPER, INK, etc, items, which are embedded in PRINT, LPRINT, INPUT, PLOT, DRAW and CIRCLE statements, and also by PAPER, INK, etc control characters when they are printed to the television - they are followed by a further byte to specify the parameter value. Temporary parameters last only to the end of the PRINT (or

whatever) statement, or, in **INPUT** statements, until some **INPUT** data is needed from the keyboard, when they are replaced by the permanent parameters.

**PAPER** and **INK** parameters are in the range 0 to 9. Parameters 0 to 7 are the colours used when a character is printed:



Parameter 8 ('transparent') specifies that the colour on the screen is to be left unchanged when a character is printed.

Parameter 9 ('contrast') specifies that the colour in question (paper or ink) is to be made either white or black to show up against the other colour.

**FLASH** and **BRIGHT** parameters are 0, 1 or 8: 1 means that flashing or brightness is turned on, 0 that it is turned off, and 8 ('transparent') that it is left unchanged at any character position.

**OVER** and **INVERSE** parameters are 0 or 1:

**OVER 0** new characters obliterate old ones

**OVER 1** the bit patterns of the old and new characters are combined using an 'exclusive or' operation (overprinting)

**INVERSE 0** new characters are printed as ink colour on paper colour (normal video)

**INVERSE** 1 new characters are printed as paper colour on ink colour (inverse video)

When a TAB control character is received by the television, two more bytes are expected to specify a tab stop n (less significant byte first). This is reduced modulo 32 to  $n_0$  (say), and then sufficient spaces are printed to move the printing position into column  $n_0$ . When a comma control character is received, then sufficient spaces (at least one) are printed to move the printing position into column 16. When an **Enter** control character is received, the printing position is moved on to the next line.

The printer

Output to the ZX printer is via a buffer one line (32 characters) long, and a line is sent to the printer

- 1. when printing spills over from one line to the next,
- 2. when an **Enter** character is received,
- 3. at the end of the program, if there is anything left unprinted,
- 4. when a **TAB** control or comma control moves the printing position on to a new line.

TAB controls and comma controls output spaces in the same way as on the television.

The AT control changes the printing position using the column number, and ignores the line number.

The printer is affected by **INVERSE** and **OVER** controls (and also statements) in the same way as the screen is, but not by **PAPER**, **INK**, **FLASH** or **BRIGHT**.

The printer will stop with error B if **Esc** is pressed. If the printer is absent the output will simply be lost.

Part 2: The BASIC

Numbers are stored to an accuracy of 9 or 10 digits. The largest number you can get is about  $10^{38}$ , and the smallest (positive) number is about  $4 \times 10^{-39}$ .

A number is stored in floating point binary with one exponent byte e ( $1 \le e \le 255$ ), and four mantissa bytes m ( $1/2 \le m < 1$ ). This represents the number m×2e<sup>-128</sup>.

Since  $1/2 \le m \le 1$ , the most significant bit of the mantissa m is always 1. Therefore in actual fact we can replace it with a bit to show the sign - 0 for positive numbers, 1 for negative.

Small integers have a special representation in which the first byte is 0, the second is a sign byte (0 or FFh) and the third and fourth are the integer in twos complement form, the less significant byte first.

Numeric variables have names of arbitrary length, starting with a letter and continuing with letters and digits. Spaces and colour controls are ignored and all letters are converted to lower-case letters.

Control variables of **FOR-NEXT** loops have names a single letter long.

Numeric arrays have names a single letter long, which may be the same as the name of a simple variable. They may have arbitrarily many dimensions of arbitrary size. Subscripts start at 1.

Strings are completely flexible in length. The name of a string consists of a single letter followed by **\$**.

String arrays can have arbitrarily many dimensions of arbitrary size. The name is a single letter followed by \$ and may not be the same as the name of a string. All the strings in a given array have the same fixed length, which is specified as an extra, final dimension in the DIM statement. Subscripts start at 1.

Slicing: Substrings of strings may be specified using slicers. A slicer can be

- 1. empty or
- 2. numerical expression or
- 3. optional numerical expression **TO** optional numerical expression

and is used in expressing a substring either by

```
(a) string expression (slicer)
(b) string array variable (subscript, ..., subscript, slicer)
which means the same as
string array variable (subscript..., subscript) (slicer)
```

In (a), suppose the string expression has the value **S\$**.

If the slicer is empty, the result is **S\$** considered as a substring of itself.

If the slicer is a numerical expression with value M, then the result is the Mth character of S\$ (a substring of length 1).

If the slicer has the form (iii), then suppose the first numerical expression has the value M (the default value is 1), and the second, N (the default value is the length of s\$).

If  $1 \le m \le n \le the$  length of s then the result is the substring of s starting with the Mth character and ending with the Nth. If  $0 \le n \le m$  then the result is the empty

string. Otherwise, error 3 results.

Slicing is performed before functions or operations are evaluated, unless brackets dictate otherwise.

Substrings can be assigned to (see LET).

If a string quote is to be written in a string literal, then it must be doubled.

### Functions

The argument of a function does not need brackets if it is a constant or a (possibly subscripted or sliced) variable.

ABS	Gives the absolute magnitude of a <i>numeric-expression</i> .
ACS	Gives the arc-cosine of a numeric expression.
AND	Logical conjunction.
ASN	Gives the arc-sine of a numeric expression.
AT	Moves the print position to a point on the screen.
ATN	Gives the arc-tangent of a numeric expression.
ATTR	Gives the number whose binary form codes the attributes of a line.
BIN	Binary representation.
CHR\$	Gives the character whose code is a given integer.
CODE	Gives the code of the first character in a string.

COS	Gives the cosine of a numeric expression.
EXP	Gives the value of e to a given power.
FN	Gives the return value from a user-defined function.
IN	Gives the result from inputting at processor level from a port.
INKEY\$	Reads the keyboard.
INT	Gives the integer part of a numeric expression.
LEN	Gives the length of a string expression.
LN	Gives the natural logarithm (to base e) of a numeric expression.
NOT	Logical inversion.
OR	Logical disjunction.
PEEK	Gives the value of a byte in memory.
PI	Gives the value of the constant $\pi$ .
POINT	Determines whether a pixel is ink colour or paper colour.
RND	Gives the next pseudo-random number in a sequence.
SCREEN\$	Gives the character that appears on the screen at the given position.
SGN	Gives the sign of a numeric expression.

SIN	Gives the sine of a numeric expression.	
SQR	Gives the square root of a numeric expression.	
STR\$	Gives the string of characters that would be displayed if a numeric expression were printed.	
ТАВ	Shift the print position to a column on the current line.	
TAN	Gives the tangent of a numeric expression.	
USR	Calls a machine code subroutine, or gives the address of the first bit pattern for a user-defined graphic.	
VAL	Gives the value of a string expression evaluated as a numeric expression.	
VAL\$	Gives the value of a string expression evaluated as a numeric expression and converted back to a string.	

## The following are binary operations:

+	Addition (on numbers), or concatenation (on strings)
-	Subtraction
*	Multiplication
/	Division
1	Raising to a power. Error B if the left operand is negative
=	Equals

>	Greater than	
<	Less than	
<=	Less than or equal to	Both operands must be of the same type. The result is a number: 1 if the comparison holds and 0 if it does not
>=	Greater than or equal to	
<>	Not equal to	

Functions and operations have the following priorities:

Operation	Priority
Subscripting and slicing	12
All functions except <b>NOT</b> and unary minus	11
1	10
Unary minus (i.e. minus just used to negate something)	10
*, /	8
+, - (minus used to subtract one number from another)	6
=, >, <, <=, >=, <>	5
ΝΟΤ	4
AND	3

OR	2

Statements

Note that arbitrary expressions are allowed everywhere (except for the line number at the beginning of a statement).

All statements can be used either as commands or in programs (although they can be more sensible in one than the other). A command or program line can have several statements, separated by colons (:). There is no restriction on whereabouts in a line any particular statement can occur - although see IF and REM.

BEEP	Sounds a note through the loudspeaker.
BORDER	Sets the colour of the border of the screen.
BRIGHT	Sets brightness of characters subsequently printed.
САТ	Microdrive file handling command.
CIRCLE	Draws an arc of a circle.
CLEAR	Deletes all variables, freeing the space they occupied.
CLOSE #	Microdrive file handling command.
CLS	Clears the display file.
CONTINUE	Continues the program, starting where it left off last time it stopped.

ATAPart of the DATA list.EF FNUser-defined function definition.IMSets up an array with specified dimensions.RAWDraws a line from the current plot position.RASEMicrodrive file handling command.LASHDefines whether characters will be flashing or steady.ORSets up a FOR-NEXT loop.
EF FNUser-defined function definition.IMSets up an array with specified dimensions.RAWDraws a line from the current plot position.RASEMicrodrive file handling command.LASHDefines whether characters will be flashing or steady.ORSets up a FOR-NEXT loop.
IMSets up an array with specified dimensions.RAWDraws a line from the current plot position.RASEMicrodrive file handling command.LASHDefines whether characters will be flashing or steady.ORSets up a FOR-NEXT loop.
RAW       Draws a line from the current plot position.         RASE       Microdrive file handling command.         LASH       Defines whether characters will be flashing or steady.         OR       Sets up a FOR-NEXT loop.
RASE       Microdrive file handling command.         LASH       Defines whether characters will be flashing or steady.         OR       Sets up a FOR-NEXT loop.
LASH       Defines whether characters will be flashing or steady.         OR       Sets up a FOR-NEXT loop.
OR Sets up a FOR-NEXT loop.
URMA I Microdrive file handling command.
<b>0</b> SUB Jumps to a subroutine.
<b>0 TO</b> Jumps to a line.
F Conditional execution.
<b>NK</b> Sets the ink colour of characters subsequently printed.
<b>NPUT</b> Waits for input of an expression from the keyboard.
<b>NVERSE</b> Controls inversion of characters subsequently printed.
<b>ET</b> Assigns the value of an expression to a variable.

LINE	Removes quotes on string variable INPUTs, Autostarts saved programs.
LIST	Lists the program to the upper part of the screen.
LLIST	Lists the program to the printer.
LOAD	Loads a program or data.
LPRINT	Outputs characters to the printer.
MERGE	Loads a program and merges it with the existing one.
MOVE	Microdrive file handling command.
NEW	Starts the BASIC system off anew, deleting program and variables.
NEXT	Continues a FOR-NEXT loop.
OPEN #	Microdrive file handling command.
OUT	Outputs a byte at a port at the processor level.
OVER	Controls overprinting for characters subsequently printed.
PAPER	Sets the paper colour of characters subsequently printed.
PAUSE	Stops computing for the specified number of frames or until a key is pressed.
PLOT	Prints an ink spot at a pixel.
РОКЕ	Writes a value to the byte in memory.

PRINT	Outputs characters to the screen.
RANDOMIZE	Seeds the random number generator.
READ	Assigns to variables using successive expressions in the DATA list.
REM	No effect.
RESTORE	Restores the DATA pointer to the first DATA statement.
RETURN	Returns from a subroutine.
RUN	Runs the program.
SAVE	Saves a program or data.
STOP	Stops the program with report 9.
ТАВ	Stops the program with report 9.
THEN	Separates an IF condition from the code to be executed.
то	Denotes the limit of a FOR statement, and a string slicer.
VERIFY	Compares a saved program with the one in memory.

## ← Appendix B

## **Manual credits**

*Sinclair ZX Spectrum: BASIC Programming* By Steven Vickers. Edited by Robin Bradbeer.

Original edition published in 1982 by Sinclair Research Limited. Subsequently converted to ASCII text and published in 1995 by Chris Owen on behalf of Amstrad plc.

HTML version by Pete Robinson.

This version reformatted and modified for BASin by Paul E Collins and published in 2004.

This manual is freely distributable but must not be distributed without this notice.

© Copyright 1995 Amstrad plc - all rights reserved.

## **ABS** ABSolute value

Function

ABS gives the absolute magnitude of a numeric value, that is the value without a positive or negative sign.

How to use ABS

ABS is followed by a numeric value. An expression must be *enclosed in brackets*, for example

50 LET x=ABS(y-z)

ABS returns the absolute value of the numeric value.

Example

The command

### PRINT ABS -34.2

displays 34.2.

Format

- ABS num-const
- ABS num-var
- ABS (num-expr)

See also

Chapter 9.

## ACS Arc CoSine

Function

ACS calculates the value of an angle from its cosine.

How to use ACS

ACS is followed by a numeric value. An expression must be *enclosed in brackets*, for example

```
60 LET x=ACS(y*z)
```

The value following ACS (y\*z above) is the cosine of the required angle and may range from -1 to 1. ACS the returns the value of the angle in *radians*. To convert radians to *degrees*, multiply the value returned by ACS by 180/PI.

Example

The command

### PRINT 180/PI \* ACS 0.5

displays 60, the angle in degrees that has a cosine of 0.5.

Format

- ACS num-const
- ACS num-var
- ACS (num-expr)

See also

Chapter 10.

## AND

Logical Operator/Function

AND acts as a logical operator to test the truth of a combination of conditions. Only if *all* conditions are true is the overall combination true. AND also acts as a function to perform binary operations on two numeric or string values.

How to use AND

As a logical operator, AND links two conditions in a statement where the truth of the whole is to be tested, for example

### 90 IF x=y+z AND time<10 THEN PRINT "Correct"

Only if both conditions are true will the computer display "Correct". If either or both conditions are false, then the whole combination is false and in this example, the program proceeds to the next line.

AND as a function

As a function, AND can operate on two numeric values, for example

### 50 LET x=y AND z

AND returns the first value (y) if the second (z) is not equal to 0, and returns 0 if the second value(z) is 0.

AND may also operate on a string value providing it precedes AND. A numeric value must always follow AND, for example

### 50 LET a\$=b\$ AND z

AND returns the first value (b\$) if the second (z) is *non-zero* and a null string ("") if the second value (z) is 0.

Note that the BASIC assigns a value of 1 to a true condition and 0 to a false condition, and recognises any non-zero value as true and 0 as false. It does not evaluate combinations of numeric values in accordance with standard truth tables.

Examples

```
60 LET correct=(x=y+z) AND time<10
70 LET score=score+10*(1 AND correct)
80 LET a$=("Out Of Time Or Not " AND NOT correc</pre>
```

If the two conditions in line 60 are true then the numeric variable *correct* is assigned a value of 1. The *score* is increased by 10 and a\$ becomes "Correct". If either of the conditions is false, then *correct* has a value of 0; *score* is unchanged, and a\$ becomes "Out Of Time Or Not Correct".

Format

- cond **AND** cond
- num-expr AND num-expr
- string-expr **AND** num-expr

See also

Chapter 13.

## ASN Arc SiNe

Function

ASN calculates the value of an angle from its sine.

How to use ASN

ASN is followed by a numeric value. An expression must be *enclosed in brackets*, for example

```
60 LET x=ASN(y*z)
```

The value following ASN (y\*z above) is the sine of the required angle and it may range from -1 to 1. ASN then returns the value of the angle in *radians*. To convert radians to *degrees*, multiply the value returned by ASN by 180/PI.

Example

The command

### PRINT 180/PI \* ACS 0.5

display 30, the angle in degress that has a sine of 0.5.

Format

- ASN num-const
- ASN num-var
- ASN (num-expr)

See also

Chapter 10.

# AT

### PRINT item

AT is used in **PRINT** statements to change the position at which new characters are displayed on the screen. AT is also used to change the place that **INPUT** items are displayed in the lower screen.

How to use AT

AT may be used to form a statement. It is preceded by **PRINT** or **INPUT**, and is followed by two numeric values *separated by a comma*, for example

### 10 PRINT AT 21,0;"The lowest line"

The first value following AT (21 above) defines the row component of the new position. This is rounded down to the nearest integer if necessary, and may range from 0 to 21. The second value is also rounded down if necessary and may then range from 0 to 31, and defines the column of the new position. Any characters then displayed will start at this new position.

Using AT with INPUT

In an **INPUT** statement, AT will change the position on the lower screen that new prompts appear. The first value moves the prompt position up as it increases. The second value changes the column as above.

Format

• **AT** int-num-expr, int-num-expr

See also

Chapter 15; INPUT; LPRINT; PRINT.

## ATN Arc TaNgent

Function

ATN calculates the value of an angle from its tangent.

How to use ATN

ATN is followed by a numeric value. An expression must be *enclosed in brackets*, for example

```
60 LET x=ATN(y*z)
```

The value following ATN (y\*z above) is the cosine of the required angle and may range from -1 to 1. ATN the returns the value of the angle in *radians*. To convert radians to *degrees*, multiply the value returned by ATN by 180/PI.

Example

The command

### PRINT 180/PI \* ATN 0.5

displays 45, the angle in degrees that has a tangent of 0.5.

Format

- ATN num-const
- ATN num-var
- ATN (num-expr)

See also

Chapter 10.
# **ATTR** ATTRibutes

#### Function

ATTR gives the attributes of a specified character position on the screen. These are the ink and paper colours, brightness and flash status of the character at the position.

How to use ATTR

ATTR is followed by two numeric values *separated by a comma and enclosed in brackets*, for example

# 150 IF ATTR(v,h)=115 THEN GO SUB 2000

The first value following ATTR (v above) may range from 0 to 23 and is the line number of a position on the screen. The second value (h above) may range from 0 to 31, and is the column number of the position. ATTR then returns a number from 0 to 255. This number is the sum of the attributes at the specified position, and is made up as follows:

Ink colour (0 to 7)

Paper colour (8 times 0 to 7)

Bright (64)

Flashing (128)

Example

If a character at position 11,16 is displayed in ink colour 3 (magenta), paper colour 6 (yellow), and is bright but not flashing, then the command

## PRINT ATTR(11,16)

displays 115 (3 + 8×6 + 64 + 0).

#### ATTR in binary form

ATTR returns one byte in which bit 7 (most significant) is 1 for flashing or 0 for normal, bit 6 is 1 for bright or 0 for normal, bits 5 to 3 are the paper colour (in binary) and bits 2 to 0 are the ink colour.

Format

• **ATTR** (num-expr, num-expr)

# BEEP

Statement/Command

BEEP makes the loudspeaker produce a single note of a given duration and pitch.

How to use BEEP

BEEP may be used to form a statement in a program or a direct command. It is followed by two numeric values *separated by a comma*, for example

## 80 BEEP x,y

The first value (x) may range from 0 to 10 and defines the duration of the note in seconds. The second value (y) may range from -60 to 69 and defines the pitch of the note in semitones below middle C if negative and above middle C if positive.

Example

The command

### BEEP 0.5,1

causes the note C# above middle C to sound for half a second.

Format

• **BEEP** num-expr, num-expr

See also

Chapter 19.

# **BIN** BINary number

BIN turns a binary number into a decimal number.

How to use BIN

BIN is followed by a binary number consisting of up to sixteen 1s and 0s, for example

## 50 POKE USR "a", BIN 10101010

BIN returns the decimal value of the binary number. It is commonly used in conjunction with POKE and USR as above for creating user-defined graphics characters, with 1 signifying a pixel of ink colour, and 0 a pixel of paper colour.

Example

The command

### PRINT BIN 1111110

displays 254, the decimal value of the binary number.

Format

• **BIN** [1][0]

See also

Chapter 14.

# BORDER

Statement/Command

BORDER specifies the colour of the border around the screen display area.

How to use BORDER

BORDER may be used as a direct command or as a statement in a program. It is followed by a numeric value, for example

## 30 BORDER RND\*7

The value following BORDER is rounded to the nearest integer and specifies the colour of the border as follows:

0 Black

- 1 Blue
- 2 Red
- 3 Magenta
- 4 Green
- 5 Cyan
- 6 Yellow
- 7 White

Note that BORDER also sets the paper colour of the lower part of the screen. Unlike INK and PAPER, a BORDER statement cannot be embedded (inserted) in a PRINT statement.

Format

• **BORDER** int-num-expr

See also

Chapter 16.

# BRIGHT

Statement/Command

BRIGHT causes characters to be displayed in brighter colours than normal.

How to use BRIGHT

BRIGHT may be used as a direct command but is normally used to form a statement in a program. It is followed by a numeric value, for example

## 80 BRIGHT 1

The value following BRIGHT is rounded to the nearest integer if necessary and may then be either 0, 1 or 8. A value of 1 causes all characters subsequently displayed by PRINT or INPUT statements to appear in a brighter ink and paper colour, and a value of 8 causes bright character positions to remain bright and normal character positions to remain normal when new characters are printed there. BRIGHT followed by 0 cancels both BRIGHT 1 and BRIGHT 8 so that all characters subsequently displayed are normal.

BRIGHT may also be embedded (inserted) within display statements formed by PRINT, INPUT, PLOT, DRAW and CIRCLE. BRIGHT follows the keyword but precedes the data or display parameters; it is followed by the same values *and a semicolon*, for example

# 50 PRINT BRIGHT 1; "WARNING"

The effect of BRIGHT is then local and applies only to the characters displayed, point plotted or line drawn by the display statement. Note that BRIGHT 1 brightens the paper colour of the *whole* character position of 8×8 pixels if any pixel in the position in plotted in an ink colour.

Format

• **BRIGHT** int-num-expr [;]

See also

Chapter 16.

# **CAT** CATalogue

Microdrive file handling command.

CAT and the Silicon Disc

The 128k Silicon Disc used the **CAT** command to list the contents of the disc to the screen. Files are displayed sequentially in alphabetical order - and start onscreen at the current PRINT position, using one line per file. The format of the **CAT** for this purpose is...

CAT !

See also

- Chapter 20
- 128k Commands in BASin

# CHR\$ CHaRacter string

#### Function

The characters and keywords available on the keyboard plus any user-defined graphics make up the BASIC character set. By using CHR\$ and a code number, each one can be obtained as a string. The character set also contains several display codes that affect the display of characters. These codes can be brought into operation on characters displayed by using PRINT before CHR\$. The complete character set can be found here.

How to use CHR\$

CHR\$ is followed by a numeric value, for example

## 80 PRINT CHR\$ x

An expression must be *enclosed by brackets*. The value following CHR\$ (x above) is rounded to the nearest integer. If it is in the range 32 to 255, CHR\$ returns the a keyboard character, user-defined graphics character or a keyword as a string. The BASIC uses the ASCII code for values from 32 to 95 and 97 to 126. If x is assigned a value of 65, the above statement displays A, for example.

CHR\$ control codes

Values from 1 to 31 either return control codes or are not used. CHR\$ 6 (PRINT comma), 8 (back space) and 13 (new line or ENTER) affect displays on the screen if inluded in a PRINT statement. CHR\$ may be followed by the code value *and a semicolon*, for example

## 60 PRINT "A";CHR\$ 6;"B"

This statement displays

A B

Another way of using CHR\$ control codes is to form a composite string containing them. The statement

## 60 PRINT "A"+CHR\$ 6+"B"

has exactly the same effect as the previous example.

Codes 16 to 23 affect colour and position and each may be used in a composite string together with CHR\$ followed by a colour code value from 0 to 7 for CHR\$ 16 (INK control) and CHR\$ 17 (PAPER), or by 0 or 1 for CHR\$ 18 to CHR\$ 21 (FLASH, BRIGHT, INVERSE, and OVER controls). The command

## PRINT CHR\$ 16+CHR\$ 3+CHR\$ 17+CHR\$ 6+CHR\$ 18+CHR

displays BAS in in flashing red and yellow. Alternatively, as above, each plus (+) sign may be replaced by a semicolon.

CHR\$ 22 (AT control) is followed by two CHR\$ values to indicate the line and column numbers. The command

### PRINT CHR\$ 22+CHR\$ 11+CHR\$ 16+CHR\$ 42

displays a star in the centre of the screen.

CHR\$ 23 (TAB control) is also followed by two values in the same way. The second value is normally 0 and the first gives the TAB position. The command

### PRINT CHR\$ 23+CHR\$ 16+CHR\$ 0+CHR\$ 42

displays a star halfway across the screen.

Note that only these controls are available. Using **PRINT** CHR\$ with a keyword value greater than 164 simply displays the keyword and does not bring it into operation.

Format

• CHR\$ int-num-const [;] [+]

- CHR\$ int-num-var [;] [+]
- **CHR\$** (int-num-expr) [;] [+]

See also

Chapter 14.

# CIRCLE

Statement/Command

CIRCLE draws a circle on the screen.

How to use CIRCLE

Circle is followed by three numeric values *each separated by a comma*, for example

## 80 CIRCLE x,y,z

Each of the three values is rounded to the nearest integer if necessary. CIRCLE then draws a circle on the high resolution graphics grid in the current ink colour. The first two values (x,y) define the horizontal and vertical coordinates of the centre, and the third value (z) defines the length of the radius. The dimensions must be such that the circle does not extend beyond the display area.

CIRCLE is affected by colour statements or commands and may include embedded colour statements with the same effects as PLOT and DRAW.

Example

The command

## CIRCLE 128,88,87

draws a circle taking up most of the display area.

Format

• **CIRCLE** [statement;] int-num-expr, int-num-expr, int-num-expr

See also

Chapter 17.

# CLEAR

Statement/Command

CLEAR deletes the current values of all variables, freeing the memory space that the values occupied and space as far as RAMTOP, the top address of the BASIC system area. CLEAR may also be used to reset RAMTOP.

How to use CLEAR

CLEAR may be used as a direct command or it may form a statement in a program. It requires no parameters, for example

#### 50 CLEAR

CLEAR then delets the values that are currently assigned to all variables, including arrays. It also executes CLS and RESTORE to clear the screen and restore the data pointer to the first item of data. In assition, the PLOT position is reset to the bottom left-hand corner of the display area and the GO SUB stack is cleared.

Note that CLEAR is not required before re-dimensioning arrays as **DIM** deletes an existing array of the same name. Note also that **RUN** executes CLEAR.

#### CLEAR and RAMTOP

CLEAR may also be followed by a numeric value, for example

#### CLEAR 65267

Clear then executes CLEAR as above and also sets RAMTOP, the highest address of the BASIC system area, to the given value. RAMTOP is normally set at 65367 in BASIC, and lies below the area reserved for user-defined graphics. NEW clears out the memory as far as RAMTOP, so using CLEAR to lower RAMTOP (by 100 bytes in the above example) provides more memory that is immune from NEW. Raising RAMTOP gives more space for BASIC at the

expense of user-defined graphics. Note that the GO SUB stack is then located at RAMTOP.

The current RAMTOP address can be located by the following command

# PRINT PEEK 23730+256\*PEEK 23731

Format

• CLEAR [num-expr]

See also

Chapter 24.

# CLOSE #

Microdrive file-handling command

# **CLS** CLear Screen

Statement/Command

CLS clears all text and graphics from the display area leaving it blank in the current paper (background) colour.

How to use CLS

CLS may be used as a direct command or it may form a statement in a program. It requires no parameters, for example

## 250 IF a\$="NO" THEN CLS

The display area (but not the border) is then cleared to the colour selected by the previous PAPER statement or command or to the default paper colour of white.

Note that CLS must be used after PAPER and before PRINT or any other display statement to produce a coloured background over the whole display area.

Format

• CLS

See also

Chapter 15.

# CODE

#### Function

CODE gives the code number of a character in the BASIC character set (see here for the complete set).

How to use CODE

CODE is followed by a string value, for example

## 90 IF CODE a $\leq$ 65 OR CODE a $\geq$ 90 THEN GO TO 80

A string expression must be *enclosed in brackets*. CODE returns the code number of the *first* character in the string. If this is a null string ("") then CODE returns 0.

#### SAVE/LOAD/VERIFY CODE

CODE is used in a different way with SAVE, LOAD and VERIFY. See their respective entries for more information.

Format

- **CODE** string-const
- **CODE** string-var
- **CODE** (string-expr)

See also

Chapter 14.

# CONTINUE

### Command

If a program stops, CONTINUE can be used to restart the program from the point at which it stopped. If an error has occurred to halt the program, then it must be rectified before CONTINUE will allow the program to resume.

How to use CONTINUE

CONTINUE is used as a direct command when a program has stopped. It requires no parameters. After CONTINUE a program then normally resumes at the same statement at which it stopped. If the cause was an error, then a commond can be entered to rectify the error and CONTINUE will allow the program to continue from that statement. If the program stopped at a STOP statement giving report 9 ("STOP statement") or if it halted because the BREAK key was pressed giving report L ("BREAK into program"), then CONTINUE causes the program to resume from the next statement. A rectifying command can be entered first if necessary.

If CONTINUE is used to resume a direct command, then it will go into a loop if the command stopped at the first statement in the command. The display disappears, but control can be regained by pressing BREAK. CONTINUE gives report 0 ("Ok") if the command stopped at the second statement and report N ("Statement lost") at the third or subsequent statements.

Format

### • CONTINUE

See also

Chapter 2.

# COPY

Command/Statement

COPY makes Sinclair ZX Printers produce a copy of the screen display.

How to use COPY

COPY is used as a direct command or as a statement within a program. It requires no parameters. After COPY, and providing the printer is connected, a copy of the first 22 lines of the screen display is then printed. Note that all ink (foreground) colours are printed in black; paper (background) colours are not printed. The printer can be stopped by BREAK.

If a program listing appears on the screen, it can be printed by using the COPY provided it was produced by a LIST command or statement. Note that a listing will appear in the screen on pressing ENTER after a program has been completed or stopped, but this 'automatic' listing cannot be printed with COPY.

Format

• COPY

See also

Chapter 21.

# COS COSine

Function

COS calculates the cosine of an angle.

How to use COS

COS is followed by a numeric variable, for example

```
140 LET x=COS y
```

An expression must be *enclosed in brackets*. The value following COS is the angle *in radians*. COS then returns the cosine of the angle. Degrees may be converted into radians by multiplying by PI/180.

Note that COS returns a negative value for angles from 90 to 270 degrees and a positive value for angles from 0 to 90 and 270 to 360 degrees.

Example

The command

```
PRINT COS (60*PI/180)
```

displays 0.5, the cosine of 60 degrees.

Format

- COS num-const
- COS num-var
- **COS** (num-expr)

See also

Chapter 10.

# DATA

#### Statement

DATA provides a list of items of data within a program. These items may be values or variables or strings to be displayed, for example. Each item is assigned to a variable by a READ statement.

Assignment is carried out in the order in which items of data appear in the program, but **RESTORE** can be used to begin assignment at the first DATA item on or following a program line.

How to use DATA

DATA can only be used to form a statement in a program. It is normally followed by a list of numeric or string contants *each separated by a comma*, for example

### 50 DATA 31, "JAN", 28, "FEB"

Each constant is then assigned to a variable by a **READ** statement that reads the DATA. The DATA statement may be positioned anywhere in the program. The number, kind (numeric or string) and order of the constants must correspond to the number of times the **READ** statement is executed and the kind and order of variables in the **READ** statement. The list of data may be split up into several DATA statements if there are too many items to fit into one statement.

Example

The following program

```
10 FOR n=1 TO 2
20 READ x,a$
30 PRINT a$,x;" days"
40 NEXT n
50 DATA 31,"JAN",28,"FEB"
```

displays

JAN	31	days
FEB	28	days

Using DATA with variables

The items of data in a DATA statement may consist of numeric or string variables or expressions provided the variables have previously been assigned values. In the above example, the DATA statement may be changed to

```
50 DATA d, m$, d-3, "FEB"
```

If d is previously assigned a value of 31, and m\$ a value of "JAN", then the same display is given.

LOAD DATA, SAVE DATA, and VERIFY DATA

DATA may also be used with LOAD, SAVE and VERIFY to store arrays on tape. See LOAD DATA, SAVE DATA and VERIFY for more information.

Format

- **DATA** num-expr, [num-expr][, string-expr]
- **DATA** string-expr, [num-expr][, string-expr]

See also

Chapter 6.

# **DEF FN** DEFine FuNction

Statement

DEF FN enables the user to define a function that is not available as a keyword. A variety of parameters can be passed to the function in an FN statement, which calls the function and may return either a numeric or string value as a result.

How to use DEF FN

DEF FN may only be used as a statement in a program. If a numeric function is to be defined, DEF FN is followed by any single letter and then by one or more numeric variables *each separated by a comma and enclosed in brackets*. For example, DEF FN r(x,y). This is followed by an equals sign and then a numeric expression containing the variables, for example

1000 DEF FN  $r(x,y)=SQR(x\uparrow 2+y\uparrow 2)$ 

The letter following DEF FN (r above) is a name that identifies the function. The variables may also only be single letters. Note that in both cases, the BASIC does not distinguish between capital and lower-case letters.

The expression that follows the equals sign uses the variables (x and y above) to define the function.

A DEF FN statement may be placed anywhere in the program. To call a function that it defines, a FN statement is used. This is then followed by the function name letter and a list of numeric values *each separated by a comma and enclosed in brackets*, for example

## 50 PRINT FN r(3,4)

The values in the brackets are passed to the function in the same order as the variables in the DEF FN statement. Thus, in this example, x is assigned a value of 3 and y a value of 4. FN evaluates the expression and returns the value.

DEF FN may also be followed by a letter and a pair of brackets only, for example

# 1000 DEF FN r()=INT (x+0.5)

The value currently assigned to the variable (x above) is passed to the function when it is called by FN. In this case, FN r() returns the value currently assigned to x rounded to the nearest integer.

DEF FN and strings

DEF FN and FN may also be used the same way to define and call a string function. In this case, the function name is a single letter followed by \$ and one or more of the variables in the statement is a letter followed by \$. A corresponding string expression forms the definition, for example

1000 DEF FN a\$(b\$,x,y)=b\$(x TO y)

The string expression following the equals sign in this example is a string slicer, and x and y are the first and last characters of a section of b\$. FN must be followed by the function name and, in brackets, a string value together with any other parameters that are to be passed to the function. In this case, the command

PRINT FN a\$("FUNDAMENTAL",1,3)

displays FUN, and the command

# PRINT FN a\$("FUNDAMENTAL", 5, 8)

displays AMEN.

Format

- **DEF FN** letter ([letter] [,letter])=num-expr
- **DEF FN** letter\$ ([letter\$] [letter] [,letter\$])=string-expr
- **FN** letter ([num-expr] [,num-expr])
- **FN** letter\$ ([string-expr] [,num-expr] [,string-expr])

See also

Chapter 9.

# **DIM** DIMension

Statement

DIM is used to dimension (set up) an array of a given number of numeric or string variables. An array is a list of variables of the same name that are distinguished by a *subscripts* (values that identify each variable or element in the array).

How to use DIM with numeric arrays

DIM is used to form a statement in a program. It is followed by a single letter that names the array, and one or more numeric values *each separated by a comma and enclosed in brackets*, for example

10 DIM x(10) 20 DIM z(20,5)

In the first case, a one-dimensional array is created containing ten elements with subscripts from 1 to 10. The array has the name X and the subscripted variables are  $\times(1)$  to  $\times(10)$  inclusive. Any existing array of the same name is deleted, and the variables are each assigned a value of 0. Note that in dimensioning an array, the BASIC does not distinguish between names with capital and lowercase letters - variable  $\times(2)$  is the same as  $\times(2)$ . However, simple numeric variables having the same letter as an array name (X or X) can coexist and may be used separately if required.

The number of values in brackets equals the number of dimensions in a numeric array. The second example sets up a two-dimensional array of 100 elements with 20 elements in the first dimension and 5 in the second. These elements are numbered z(1, 1) to to z(20, 5).

Arrays of any number of dimensions may be created.

The elements of a numeric array may subsequently be identified by the array value *in brackets*, for example

```
70 PRINT x(a)
160 PRINT z(7,b)
```

DIM and string arrays

DIM is used in the same way as with numeric arrays except that a single letter followed by \$ is used for the array name. Furthermore, an extra value must be added to the dimension values in brackets in order to define the length of each string. For example

30 DIM a\$(20,5) 90 DIM b\$(20,5,10)

The first statement creates an array of 20 elements, each of which contains a string of 5 characters. The subscripted variables are named a\$(1) to a\$(20) inclusive, and they are initially assigned a null (empty) string (""). Any existing array of the same name is deleted and, unlike numeric arrays, a simple string variable of the same name cannot coexist.

The second example creates a two-dimensional array of 100 elements with 20 elements in the first dimension and 5 in the second. All elements have a length of 10 characters.

When string values are assigned to a string array, they are padded out with spaces at the end of the string or truncated to the defined length if necessary.

The elements of a string array are identified by the array name followed, in brackets, by one or more numeric values giving the subscript number(s). For example, element a(2) may be "SMITH" and element b\$(12,4) may be "DERBYSHIRE". However, an extra value may be added to define a particular character in a string. In these examples, a\$(2,2) would be "M" (the second character in "SMITH"), and b\$(12,4,5) would be "Y".

Zero-dimension string arrays

It is possible to create a zero-dimension string array by using only one value in brackets, for example

# 10 DIM c\$(15)

This array has only one element, which is c\$, and its length is fixed at the defined value (15 characters).

Format

- **DIM** letter (num-expr [,num-expr])
- **DIM** letter\$ (num-expr [,num-expr])

See also

Chapter 12.

# DRAW

Statement/Command

DRAW is used to draw straight lines and curves on the screen.

How to use DRAW

DRAW is normally used to for a statement in a program. If a straight line is required, it is followed by two numeric values *separated by a comma*, for example

#### 40 DRAW x,y

A straight line is then drawn on the high-resolution graphics grid from from the position defined by the previous PLOT statement or the previous DRAW statement, whichever is last. Both values following DRAW are rounded to the nearest integer if necessary. The first value (x above) defines the horizontal distance from this position, and the second value (y) the vertical distance. These values are negative if the line is to go to the left or down respectively, and the position reached must be within the display area.

If there is no previous **PLOT** or DRAW statement, DRAW commences at position 0,0 (the bottom left-hand corner of the screen).

DRAW is affected by colour statements or commands and may include embedded statements with the same effects as with PLOT and CIRCLE.

DRAWing curved lines

DRAW may be followed by a third value to produce a curve that is a part of a circle, for example

### 40 DRAW x, y, z

The third value (z above) defines the angle (in radians) through which the line

turns as it is drawn. The line turns to the left if this is positive, and to the right if it is negative. Values of **PI** or **-PI** produce a circle.

Example

The following program draws a triangle:

10 PLOT 127,50 20 DRAW 70,-100 30 DRAW -140,0 40 DRAW 70,100

Adding 1 or -1 to the DRAW statement causes the sides to curve in or out respectively.

Format

• **DRAW** [statement;] int-num-expr, int-num-expr[, int-num-expr]

See also

Chapter 17.

# ERASE

Microdrive file handling command.

ERASE and the Silicon Disc

Use ERASE with the 128k Silicon Disc to delete files previously saved onto it. It is recommended that you delete files in reverse order of their creation, i.e the last to be saved should be the first to be deleted - this will save a lot of time spent deleting many files. The format for the ERASE command is...

## ERASE ! filename

Where filename is the name of a file saved previously, as a string expression or string literal.. You can generate a list of files using the CAT command.

See also

- Chapter 20
- Keyword Reference CAT
- 128k Commands in BASin

# **EXP** EXPonent

Function

EXP is a mathematical function that raises the exponent *e* to a given power.

How to use EXP

EXP is followed by a numeric value, for example

60 LET y=EXP x

An expression mst be *enclosed in brackets*. EXP then returns the exponent *e* raised to the power of the argument (x above).

Example

The command

#### PRINT EXP 1

displays 2.7182818, the value of *e*.

Format

- EXP num-const
- **EXP** num-var
- **EXP** (num-expr)

See also

Chapter 10.

# FLASH

Statement/Command

FLASH causes character positions to flash, making ink and paper colours alternate at a constant rate.

How to use FLASH

FLASH may be used as a direct command but it is normally used to form a statement in a program. It is followed by a numeric value, for example

50 FLASH 1

The value is rounded to the nearest integer if necessary and may then be either 0, 1 or 8. A value of 1 causes all characters subsequently displayed by **PRINT** or **INPUT** to flash. A value of 8 causes flashing character positions to remain flashing and normal character positions to remain normal when new characters are printed there. FLASH followed by 0 cancels both FLASH 1 and FLASH 8 so that all characters subsequently displayed are normal.

FLASH may be embedded (inserted) within statements formed by **PRINT**, **INPUT**, **PLOT**, **DRAW** and **CIRCLE**. FLASH follows the keyword, but precedes the data or display parameters; it is followed by the same values *and a semicolon*, for example

### 120 PRINT FLASH 1; INK 2; PAPER 6; "WARNING"

The effect of the FLASH is then local and applies only to the characters displayed, point plotted or line drawn by the display statement. Note that FLASH 1 causes the whole 8×8 pixel position to flash if any pixel is plotted in an ink colour.

Format

• **FLASH** int-num-expr[;]

See also

Chapter 16.

# **FN** FuNction

#### Function

FN calls a user-defined function. It is always used in conjunction with DEF FN, which defines the function to be called.

How to use FN

If a numeric function is to be called, FN is followed by a letter and then a pair of brackets. If any parameters are to be passed to the function, then these are *each separated by a comma and enclosed in brackets*, for example

170 LET x=FN r(3,4)

The parameters are then passed to the function called "r". FN then returns the result. If no parameters are to be passed, then the pair of brackets must still be included, for example

# 70 PRINT FN r()

In this case, the function uses the values currently assigned to its variables.

FN calls a string function the same way, except that \$ must be added after the letter (see DEF FN).

FN does work recursively, but care should be taken not to cause an infinite recursion and eventual memory waste.

Format

- **FN** letter ([num-expr] [,num-expr])
- **FN** letter\$ ([string-expr] [num-expr] [,num-expr] [,string-expr])

See also

Chapter 9.

# FOR

Statement/Command

FOR is always used with the keywords **TO** and **NEXT** to create a FOR-NEXT loop. This structure enables a section of the program to repeat a given number of times.

How to use FOR

FOR always forms a statement with **TO**. FOR is followed by a letter, an equals sign, and then two numeric values separated by **TO**, for example

#### 60 FOR a=1 TO 9

The letter (a above) forms a control variable. The statements that are to be repeated follow, and one or more of these normally makes use of the control variable. The loop then ends with a NEXT statement, in which NEXT is followed by the control variable, for example

#### 90 NEXT a

On execution, FOR deletes any variable of the same name as the control variable and assigns it an initial value equal to the value before TO (1 above). The statements are then executed with the control variable having this value. On reaching NEXT, the value of the control variable is increased by 1. If this value is less than or equal to the value after TO (the limit value 9 above), the program returns to the FOR statement and the FOR-NEXT loop is repeated. If the control variable has a greater value than the limit value, then the loop ends and the program continues with the statement after NEXT.

In the above example, the loop is repeated nine times with the control variable *a* increasing from 1 to 9. On leaving the loop, *a* has a value of 10.

Note that the BASIC does not distinguish between capital and lower-case letters when naming the control variable.
Using STEP in a FOR-NEXT loop

**STEP** is a keyword that can be incorporated in a FOR statement if the control variable is to increase by a value other than 1 or decrease. **STEP** follows the limit value and is followed by a numeric value, for example

## 60 FOR a=1 TO 9 STEP 2

The control value is increased by the step value (2 above) until it is greater than the limit value. The control variable *a* has successive values of 1, 3, 5, 7 and 9 and leaves the loop with a value of 11.

A negative step value causes the control variable to decrease. In this case, the initial value must be greater than the limit value and the loop ends when the value of the control variable is less than the limit value, for example

## 60 FOR a=9 TO 1 STEP -1

The value of *a* decreases from 9 to 1 and leaves the loop with a value of 0.

Nesting loops

One or more FOR-NEXT loops may be placed inside each other, a procedure called "nesting" loops. The order of the control variables in the NEXT statements must be the reverse of the of the order of the control variables in the FOR statements. FOR-NEXT loops may be nested to any depth, that is as many loops as required may be placed inside each other.

Format

- **FOR** letter=num-expr **TO** num-expr [**STEP** num-expr]
- **NEXT** letter

See also

Chapter 4.

# FORMAT

Microdrive file handling command.

## GO SUB GO to SUBroutine

Statement/Command

GO SUB causes the program to branch to a subroutine, which is a seperate section of the program. This is useful if a subroutine is required several times in a program.

How to use GO SUB

GO SUB may be used as a statement or direct command and it is followed by a numeric value, for example

GO SUB 1000

On execution, the value following GO SUB (1000 above) is rounded to the nearest integer, and the program branches to the line number having this value. The use of a variable or expression enables the program to branch to a subroutine at a calculated line number. Note that if the line number does not exist, the program still branches and continues with the first statement that is then encountered.

A subroutine ends with **RETURN**, and the program then branches back to the statement following the GO SUB statement. Subroutines may be nested so that one is reached from another, in which case the **RETURN** sends the program back to the statement following the last GO SUB statement executed.

The GO SUB stack

Whenever GO SUB is executed, its line number is placed on the GO SUB stack in memory. If two or more GO SUBs are executed before **RETURN**, their line numbers stack up so that the last number is on top of the stack. **RETURN** always takes the top line number from the stack and goes to this line to continue the program.

Note that error 4 (Out of memory) can occur if there are not enough **RETURN** 

statements.

Format

• **GO SUB** int-num-expr

See also

Chapter 5.

# GO TO

Statement/Command

GO TO Makes a program branch to a particular line.

How to use GO TO

GO TO may be used as a direct command to run a program from a given line number *without* first clearing the screen. It may also be used to form a statement in a program. GO TO is followed by a numeric value, for example

60 GO TO 350

On execution the value following GO TO is rounded to the nearest integer and the program branches to the line number having this value. The use of a variable or expression allows the program to branch to a calculated line number. Note that if the line does not exist, then the program branches and continues with the first statement that is then encountered.

Format

• GO TO int-num-expr

See also

Chapter 2.

# IF

Statement/Command

If is always used with THEN to prompt a decision that affects subsequent action. To do this, the computer tests something to find out whether or not it is true. If it is true, then one course of action follows. If it is untrue, another occurs.

How to use IF and THEN

IF normally forms a statement with THEN. IF is first followed by a numeric value or by a condition, and second by THEN and one or more BASIC statements, for example

# 80 IF x THEN GO TO 250 240 IF a="NO" THEN PRINT "THE END": STOP

A constant, variable or expression (such as x above) is considered to be true if it has a non-zero value. In this case the statement following THEN and any more statements *in the same line* are executed. The program then proceeds to the next line. If the value is 0, then the constant, variable or expression is considered to be false. The following statements are then not executed and the program skips to the next line. In the example, the program will not GO TO line 250 if x is 0.

If a condition (a\$="NO") following IF is true then the statements following THEN are executed. If the condition is false, then the program moves to the next line. In this example, if a\$ has the value "NO" then "THE END" is displayed and the program stops. If a\$ has any other value, the program continues from the next line.

The BASIC gives a true condition a value of 1 and a false condition a value of 0. It recognises any non-zero value as true and 0 as false. A variable can be assigned the value of a condition by a statement such as

70 LET x=a\$="NO"

Note that, unlike in some other BASICs, THEN cannot be ommitted before GO TO.

Format

- IF num-expr THEN statement [:statement]
- **IF** cond **THEN** statement [:statement]

See also

Chapter 3.

# IN

#### Function

IN checks the status of the keyboard and other input and output devices. It reads a byte from a given port address that indicates the status of the device connected to the port.

How to use IN

IN is followed by a numeric value, for example

#### 150 LET x=IN y

The value following IN may range from 0 to 65535 inclusive, and specifies the port address that is to be read. IN then returns the byte read from this port.

Keyboard addresses

The keyboard has eight addresses, each of which may contain one of five different bytes depending on which key is pressed. The addresses are 65278, 65022, 64510, 63486, 61438, 57342, 49150 and 32766. Byte values at these addresses may be 175, 183, 187, 189 or 190.

Format

- IN num-const
- IN num-var
- IN (num-expr)

See also

Chapter 23.

# INK

Statement/Command

INK specifies the foreground colour in which the characters are displayed, points plotted and lines and curves drawn.

How to use INK

INK May be used as a direct command but is normally used to form a statement in a program. It is followed by a numeric value, for example

### 70 INK x

The value following INK is rounded to the nearest integer and may range from 0 to 9. The following foreground colours are then given.

0 Black

- 1 Blue
- 2 Red
- 3 Magenta (Purple)
- 4 Green
- 5 Cyan (Blue-Green)
- 6 Yellow
- 7 White
- 8 Transparent
- 9 Contrasting Black or White

INK 8 specifies that the existing colour remains unchanged at any position on the screen where INK 8 is used. INK 9 causes the ink colours to be either black or white so that it shows up against the paper (background) colour.

Global and local ink colours

When INK forms a statement alone, as above, the colour is global and all subsequent displays occur in this foreground colour. INK may also be embeeded

in display statements formed by **PRINT**, **INPUT**, **PLOT**, **DRAW** and **CIRCLE**. INK follows the keyword but precedes the data or display parameters; it is followed by the same values *and a semicolon*, for example

## 60 CIRCLE INK 4;128,88,87

The effect of INK is then local and applies only to the characters displayed, point plotted or line drawn by the display statement, this example drawing a green circle. Thereafter the ink colour reverts to the global colour or default colour of black.

Format

• **INK** int-num-expr [;]

See also

Chapter 16.

## **INKEY\$** INput KEY string

Function

INKEY\$ is used to detect the pressing of the keys on the keyboard.

How to use INKEY\$

INKEY\$ requires no argument and is generally used to assign a character to a string variable or to test for a particular character, for example

## 70 LET a\$=INKEY\$ 130 IF INKEY\$="N" THEN STOP

On execution, INKEY\$ returns the character given by the key that is being pressed at that instant. If no key is being pressed, then INKEY\$ returns a null (empty) string (""). Note that INKEY\$ distinguishes between capital and lower-case letters and other shifted and un-shifted characters. (Use IN to detect any key without distinguishing characters).

Unlike **INPUT**, INKEY\$ does not wait but goes immediately to the next statement. It is therefore normally placed inside a loop that repeats until the required key is pressed.

Example

This line suspends operation until the Y key is pressed (without CAPS SHIFT or CAPS LOCK).

60 IF INKEY\$<>"y" THEN GO TO 60

Format

• INKEY\$

See also

Chapter 18.

# INPUT

Statement/Command

INPUT enables data to be entered during the running of a program.

How to use INPUT

INPUT normally forms a statement in a program and is used in a very similar way to **PRINT**. In its simplest form, it is followed by a numeric or string variable, for example

60 INPUT x 70 INPUT a\$

The computer then waits until either a number or a string is entered. The value is displayed at the beginning of the bottom line as it is keyed in. On pressing ENTER, the value is assigned to the named variable and the program continues.

An INPUT statement may include more than one variable and will display characters to form a prompt. This is done in exactly the same way as with PRINT, using quote marks to enclose the prompt characters and semicolons or commas as necessary to separate items. Display statements such as INK, FLASH and PAPER may be embedded, for example

80 INPUT INK 2;"What is your name?";n\$,("How ol

Note the following differences to **PRINT**. INPUT waits when it comes to a variable, so all variables and expressions (such as that including n\$ above) which are to be included in prompts must be *enclosed in brackets*. Display begins at the start of the bottom line and then scrolls up if more than one line is used. AT may be used in an INPUT statement in the same way as with **PRINT**. Used with INPUT, AT 0,0 displays at the start of the line above the bottom line and the display scrolls up if more than two lines are displayed.

How to halt INPUT

If INPUT is followed by a numeric variable and **STOP** is entered, then the program stops. With a string variable, the first quote mark that appears may be deleted and then **STOP** entered to halt the program.

Using INPUT with LINE

INPUT LINE may be used with string variables only. Normally, INPUT with a string variable causes a pair of quotes to be displayed. As the string is keyed in, it appears between the quotes. To remove these quotes, use INPUT LINE followed by the string variable. If a prompt is required, it is placed between the INPUT and LINE, for example

## 70 INPUT "What is your name? "; LINE n\$

Format

- **INPUT** [prompt] [;] [,] ['] num-var
- **INPUT** [prompt] [;] [,] ['] string-var
- **INPUT** [prompt] [;] [,] ['] **LINE** num-var

See also

Chapter 2, Chapter 15.

## **INT** INTeger

Function

INT changes non-integers (numbers that are not whole numbers) into integers or whole numbers.

How to use INT

INT is followed by a numeric value, for example

```
70 LET x=INT y
```

An expression must be *enclosed in brackets*. INT then returns the value *rounded down* to an integer.

Example

The command

## PRINT INT 45.67, INT -7.66

displays

```
45 -8
```

Format

- INT num-const
- **INT** num-var
- **INT** (num-expr)

See also

Chapter 9.

# **INVERSE**

Statement/Command

INVERSE causes colours to be inverted at character positions so that the ink becomes the paper and vice-versa.

How to use INVERSE

INVERSE is normally used to form a statement in a program. It is followed by a numeric value, for example

#### 70 INVERSE 1

The value following INVERSE is rounded to the nearest integer and may then be either 0 or 1. INVERSE 1 causes all subsequent displays made by **PRINT** and **INPUT** to be produced using these colours. INVERSE 0 restores the ink and paper colours to normal.

Note that INVERSE can be embedded (inserted) within display statements in the same way as INK. However, if used with PLOT, DRAW or CIRCLE then INVERSE 1 causes a line or point to be plotted in the paper colour so it disappears.

Format

• **INVERSE** int-num-expr

See also

Chapter 16.

## LEN LENgth of string

Function

LEN gives the length of a string.

How to use LEN

LEN is followed by a string value, for example

50 LET x=LEN a\$

An expression must be *enclosed in brackets*. LEN returns the number of characters in the string.

Example

The following line

120 INPUT a\$: IF LEN a\$>9 THEN GO TO 120

passes only strings that contain up to 9 characters.

Format

- LEN string-const
- LEN string-var
- **LEN** (string-expr)

See also

Chapter 9.

# LET

Statement/Command

LET is used to assign a value to a variable. In this version of BASIC, LET cannot be omitted in an assignment statement.

How to use LET

LET normally forms a statement in a program but may be used as a direct command. It is followed by a numeric or string variable, an equals sign, and then a value. The value may be numeric or string, depending on the variable preceding LET, for example

60 LET x=x+1 80 LET a\$="Correct"

The value is then assigned to the variable.

Note that simple variables are undefined until assigned values by LET, READ or INPUT. Array variables however are initialised to 0 or a null string ("") by DIM.

Format

- **LET** num-var = num-expr
- **LET** string-var = num-expr

See also

Chapter 2.

## LINE

Modifier

LINE is used in **INPUT** statements to remove the quote marks around string inputs, and with **SAVE** to force a saved program to autostart after loading.

How to use LINE with INPUT

In an **INPUT** statement with a string variable, you can remove the quotation marks around the input area and type **STOP** to break into a program. By using LINE before the variable, the quotation marks are removed, for example

10 INPUT "What is your name? ";LINE n\$

displays the prompt after "What is your name? " without any quotes.

Using LINE with SAVE

LINE can also be used in a SAVE statement to save your program in such a manner as it auto-runs after re-loading. LINE follows the filename in a SAVE statement, and is followed by a numeric value, for example

## SAVE "myprog" LINE 1

When a program is saved in this way, the value following LINE is rounded down to the nearest integer if necessary, and then may range from 1 to 9999. When the program is re-loaded, it will automatically start from that line number, or the line after if it does not exist. Note that when starting, a GO TO is performed, so variables are not cleared and the screen display is also not cleared.

Format

- **SAVE** string-expr **LINE** nt-num-expr
- **INPUT** [input-items;] **LINE** string-var

See also

Chapter 15; INPUT; SAVE

# LIST

Command/Statement

LIST produces a listing of the program currently in memory.

How to use LIST

LIST is normally used as a direct command but may form a statement in a program. To list a complete program from the first line, it is used alone. After the direct command

### LIST

the first page of the listing appears and subsequent pages will scroll up the screen at the touch of any key except N, the space bar or BREAK at the "Scroll?" prompt.

LIST may also be followed by a line number, in the form of a numeric value, for example

## LIST 100

The value following LIST is then rounded to the nearest integer if necessary, and the listing commences at this line. If there is no line with this number then the listing commences at the next line.

Format

• **LIST** [int-num-expr]

See also

Chapter 2.

## LLIST Line printer List

Command/Statement

LLIST makes Sinclair-type printers produce a print-out listing of the program currently in memory.

How to use LLIST

LLIST is used in exactly the same way as **LIST** (see **LIST** for further details). Note that the screen display does not change as the listing is printed.

Format

• LLIST [int-num-expr]

See also

Chapter 21.

## LN Logarithm (natural)

Function

LN gives the natural logarithm (the logarithm to base *e*) of a value. It acts as the inverse of EXP.

How to use LN

LN is followed by a numeric value, for example

#### 60 LET x=LN y

An expression must be *enclosed in brackets*. The value following LN must be greater than 0. LN then returns the natural logarithm of this value.

Format

- LN num-const
- LN num-var
- LN (num-expr)

See also

Chapter 10.

# LOAD

Command/Statement

LOAD loads a complete program into the memory from a tape.

How to use LOAD

LOAD is normally used as a direct command, but it may form a statement in a program in order to load a new program. LOAD is followed by a filename, which is a string value up to ten characters long, for example

## LOAD "filename"

On execution, the program currently in memory, and all the values of its variables, are deleted. The BASIC then searches for the named program and loads it when it is located. Note that the computer distinguishes between capital and lower-case letters in program names.

If a null string ("") follows LOAD, as in this command

#### LOAD ""

Then the BASIC loads the first complete program that it locates. **Note:** In BASin, **LOAD** "" will bring up the Windows file requester for you to locate a program to load yourself.

Note that LOAD is used differently when a microdrive is attached.

LOAD and the silicon disc

LOAD can also be used to access a file stored on the 128k Spectrum's silicon disc. To do this, insert a ! character between the keyword LOAD and the filename, for example

LOAD !"foo" CODE

You can use any of the commands used with LOAD such as CODE, DATA or SCREEN\$. Perhaps the most useful is the SCREEN\$ command, as loading from the silicon disc is very rapid.

Loading other types of data

LOAD can be used to load types of data other than programs; see the help entries for LOAD CODE, LOAD DATA and LOAD SCREEN\$ for further information.

Format

• **LOAD** string-expr

See also

Chapter 20.

• 128k Commands in BASin

## LPRINT Line printer PRINT

Statement/Command

LPRINT makes Sinclair-type printers print an item of data in the same way that **PRINT** causes the item to appear on the screen.

How to use LPRINT

LPRINT may form a statement in a program or a direct command. It is followed by items of data that may be separated by commas, semicolons or apostrophes, for example

60 LPRINT "Number ";x'"Name ";n\$,"Age ";a

When output to the printer, the items are printed in the same format as **PRINT** would cause them to be displayed on the screen. An LPRINT statement or command may also include TAB, certain CHR\$ controls, INVERSE and OVER statements and control codes with the same effect as **PRINT**. An AT statement may also be included, but the line number is ignored, and the item of data printed at the given column position on the same line.

Format

• LPRINT [TAB (int-num-expr);] [AT int-num-expr, int-num-expr;] [CHR\$ (int-num-expr);] [statement;] [num-expr] [string-expr] [;] [,] [']

See also

Chapter 21.

# MERGE

Statement/Command

MERGE allows two programs to be merged together.

How to use MERGE

MERGE may be used to form a statement in a program or more commonly as a direct command. It is followed by a filename in the form of a string value, for example

## 500 MERGE "prog2"

The filename following MERGE is the name of the program to be merged with the program currently in memory. This name is subject to the same restrictions as program names used with LOAD. MERGE then loads the new program without first deleting the existing program. However, the new program overwrites any lines in the existing program that have the same line numbers as lines in the new program, and variables with the same name are also overwritten.

MERGE and the silicon disc

MERGE can also be used to access a file stored on the 128k Spectrum's silicon disc. To do this, insert a ! character between the keyword MERGE and the filename, for example

#### MERGE !"bar"

If a program is well structured, then up to 91kb of BASIC code can be stored in the 128k memory model.

Format

• **MERGE** string-expr

See also

Chapter 20.

• 128k Commands in BASin

# MOVE

Microdrive file-handling command.

## NEW

Statement/Command

NEW clears the BASIC memory area (the area as far as RAMTOP) removing any program currently in this part of memory.

How to use NEW

NEW is normally used as a direct command but may form a statement in a program. It is used alone. On execution, the program and variables are deleted. The memory is cleared as far as RAMTOP so that user-defined graphics characters, which are stored above RAMTOP, are not affected.

Format

• NEW

See also

Chapter 2.

# NEXT

Statement/Command

NEXT is always used in conjunction with FOR to create a FOR-NEXT loop.

How to use NEXT

NEXT is normally used to form a statement in a program to complete a FOR-NEXT loop. It is followed by a letter that is the control variable in the loop, for example

#### 90 NEXT a

In Sinclair BASIC, the control variable must be included. See FOR for further details of FOR-NEXT loops.

Format

#### • **NEXT** letter

See also

Chapter 4.

# NOT

Logical Operator

NOT is used to reverse the truth of a condition so that a false condition becomes true and vice-versa.

How to use NOT

# NOT is followed by a condition or by a numeric value, for example

## 90 IF NOT x=y+z THEN PRINT "Wrong"

or

## 90 LET correct=x=y+z: IF NOT correct THEN PRINT

When NOT is followed by a condition (x=y+z above), the BASIC first assigns a value of 1 to the condition if it is true and 0 if it is false. NOT then acts as a function, reversing the value produced, so that the reverse of the condition can be tested. Note that a condition should be enclosed in brackets if it contains AND or OR.

If NOT is followed by a numeric value, it returns 0 if the if the value following is non-zero and 1 if the value following is 0. Thus in the above examples, the BASIC prints "Wrong" if x <> y+z or if *correct* has a value of 0.

Format

- **NOT** cond
- NOT num-expr

See also

Chapter 13.

# **OPEN**#

Microdrive file-handling command.

# OR

Logical Operator/Function

OR acts as a logical operator to test the truth of a combination of conditions. If one or more of the conditions are true, then the overall combination is true. OR also acts as a function to perform binary operations on two numeric values.

How to use OR

As a logical operator, OR links two conditions in a statement where the truth is to be tested, for example

## 70 IF INKEY\$="N" OR INKEY\$="n" THEN STOP

If any of the conditions is true, then the overall combination is true. In the line above, one of the conditions (INKEY\$="N" and INKEY\$="n") becomes true as soon as the N key is pressed, regardless of whether CAPS SHIFT or CAPS LOCK is operating or not. The whole combination is then true and the program stops.

OR as a function

The BASIC assigns a numeric value of 1 to a true condition and 0 to a false condition. It recognises any non-zero value as true and 0 as false. OR may therefore be preceded or followed by a numeric value, for example

#### 40 LET x=y OR z

The variable x is then assigned a value of 1 if z is non-zero or a true condition, or a value of y if z is 0 or a false condition.

This is useful in arithmetic. In the following example, the fare is halved if the age is less than 14.

## 60 PRINT fare\*(0.5 OR age>13)

If the age is less than 14, the condition **age>13** is false, so the fare is multiplied by 0.5. If **age>13** is true, then the fare is multiplied by 1.

Note that the BASIC does not evaluate combinations of numeric values in accordance with standard truth tables.

Format

- cond **OR** cond
- num-expr **OR** num-expr

See also

Chapter 13.

## **OUT** OUTput to port

Statement/Command

OUT sends a byte to a given input/output port address in order to drive an output device.

How to use OUT

OUT may be used to form a statement or as a direct command. It is followed by two numeric values, *separated by a comma*, for example

#### 40 OUT 254,3

Both values are rounded to the nearest integer. The first value (254 above) may then range from 0 to 65535 and is the port address. The second value (3) may range from 0 to 255 and this is the byte to be sent to the port address.

Bits 0 to 2 of the byte output to port 254 set the border colour; the above example therefore turns the border magenta. Bit 3 at this address drives the MIC output, and bit 4 the loudspeaker (for BEEP). Port address 251 drives the printer and ports 254, 247 and 239 are used with other peripherals.

Format

• **OUT** int-num-expr,int-num-expr

See also

Chapter 23.
# **OVER**

Statement/Command

OVER is used to overprint one character on another. It can also be used to plot points or draw lines or curves in a paper colour instead of an ink colour.

How to use OVER

OVER is normally used to form a statement in a program. It is followed by a numeric value, for example

80 OVER 1

The value following OVER is rounded to the nearest integer and may then be either 0 or 1. OVER 0, which is the default (preset) state, causes any character to obliterate a previous character at the same character position and replace it. OVER 1 causes any two characters displayed at the same character position to be combined.

OVER may be embedded (inserted) in a **PRINT** or **INPUT** statement in the same way as **INK** so that it affects only the characters displayed by the statement. This statement for example, underlines a word

PRINT AT 11,15;"YES"; OVER 1; AT 11,15;"\_\_\_\_"

However, note that characters are combined so that the paper colour is given where the ink colours overlap.

OVER in high resolution

OVER may be used with PLOT, DRAW and CIRCLE. Without OVER, lines and curves can overlap each other, but they must have the same ink colour otherwise the ink colour in the whole character position changes where they cross. If OVER 1 is used, lines or curves produce the paper colour where they overlap or meet characters. Plotting points or drawing lines or curves again in exactly the

same position with OVER 1 cases them to disappear.

Format

• **OVER** int-num-expr

See also

Chapter 16.

# PAPER

Statement/Command

PAPER is used to select the paper or background colour used for the screen display. This may be either the colour of the background of the whole display area, or the colour behind individual characters, points and lines that appear in single character positions.

How to use PAPER

PAPER may be used to form a statement in a program or as a direct command. It is followed by a numeric value, for example

#### 80 PAPER x

The value following PAPER is rounded to the neaerst integer and it may then range from 0 to 9. The paper colours that are given are the same as those given by INK. Paper colours may also be global or may be made local by embedding (inserting) them in display statements in exactly the same way as ink colours. See INK for further details.

Whenever characters are printed following a PAPER statement, whether global or local, the background over the whole character position affected changes to the selected colour. This is also true when points are plotted or lines or circles drawn with an embedded PAPER statement but which have not followed a global command or statement.

To produce a coloured background over the whole display area, it is necessary to use CLS after a PAPER statement. The entire display is cleared to this colour, which remains the overall background colour.

Format

• **PAPER** int-num-expr[;]

See also

Chapter 16.

# PAUSE

Statement/Command

PAUSE can be used to suspend a program for a definite or indefinite time.

How to use PAUSE

PAUSE is normally used to form a statement in a program. It is followed by a numeric value, for example

### 130 PAUSE 100

The value following PAUSE is rounded to the nearest integer and may then range from 0 to 65535. It defines the delay that occurs as this number of frames of the television picture, so that a value of 50 produces a pause of 1 second in th UK and europe where the frame frequency is 50Hz.

However, note that any pause may be cut short by pressing any key and that **PAUSE 0** gives an unlimited pause that lasts until a key is pressed.

Format

• **PAUSE** int-num-expr

See also

Chapter 18.

## PEEK

#### Function

PEEK gives the value of the byte stored at a particular location in the memory.

How to use PEEK

PEEK is followed by a numeric value, for example

```
80 LET x=PEEK(256*y)
```

Note that an expression must be *enclosed in brackets*. The value following PEEK is rounded to the nearest integer if necessary, and may range from 0 to 65535 to give an address in the memory. PEEK then returns the value of the byte (a number from 0 to 255) at the address specified.

Example

The number of frames of the display that have occurred since the program was last reset is stored at addresses 23672 to 23674. As the frames are produced at a regular rate, PEEKing these locations gives a method of measuring time. The following line displays the time in seconds since the last reset (less any time spent in producing sounds).

### 10 PRINT PEEK(23672+256\*PEEK 23673+65536\*PEEK 2

Format

- **PEEK** int-num-const
- **PEEK** int-num-var
- **PEEK** (int-num-expr)

See also

Chapter 14, Chapter 18.

# ΡΙ

Function

PI gives the value of pi ( $\pi$ ) for use in calculations. Pi is the ratio of the circumference of a circle to its diameter.

How to use PI

DRAW 255,0,-PI

PI returns a value of 3.1415927, so that the above command draws a large semicircle on the screen.

Format

• PI

See also

Chapter 10.

# PLOT

Statement/Command

PLOT is used in high-resolution graphics to plot a pixel or dot of colour at a particular position on the screen.

How to use PLOT

PLOT is used to form a statement in a program or as a direct command. It is normally followed by two numeric values *separated by a comma*, for example

### 50 PLOT 128,87

Both values following PLOT are rounded to integers if necessary. The first value may then range from 0 to 255 and defines the horizontal coordinate of a position on the screen. The second value may range from 0 to 175 and defines a vertical coordinate. A pixel is then normally plotted in the current ink colour at the defined position - in the above example at the centre of the screen.

Note the following effects of colour statements or commands on PLOT. After OVER 1, an existing dot at the same position is changed to the paper colour. Following INVERSE 1, the dot is plotted in the current paper colour. After BRIGHT 1 or FLASH 1, the whole character position on the low-resolution screen in which the pixel is plotted will be bright or flashing.

These four keywords and INK or PAPER may also be embedded (inserted) within a PLOT statement in the same way as with PRINT, for example

### 160 PLOT INK 2;x,y

Their effect is the same but is then local and limited to statement. If colour statements are embedded this way then the character position on the low-resolution screen which is occupied by the point plotted will be affected by the colour changes and assume the given values.

Note that PLOT also defines the starting position of the next DRAW statement.

Format

• **PLOT** [statement;] int-num-expr,int-num-expr

See also

Chapter 17.

# PLAY

Statement/Command

PLAY outputs multi-channel sound effects and music using the 128k Spectrum's AY-3-8912 sound chip. PLAY is a 128k command and as such introduces certain limitations to the BASIC you can use.

How to use PLAY

PLAY may be used as a direct command but is normally used to form a statement in a program. It is commonly followed by one to three string values, and up to eight if being used to control a MIDI device, for example

### 10 PLAY "ccc", "@", "&&g;"

The values following PLAY must equate to strings, and must contain specific characters. Musical notes are represented by their alphabetical names, with the current octave in lower-case, with upper-case representing one octave higher.

Note lengths are measured in crotchets, and may range from 1 to 9. Notes to be played in triplets (three notes played in the time of two) are preceded by 10, 11 and 12 for triplets in semi-quavers, quavers, and crotchets.

Command characters in a PLAY string

Music performed by the PLAY command can be interspersed with *command characters* which are single upper-case letters usually followed by a number. A command character will affect all music in that channel from that point onwards.

String	Function
ag, AG	Specifies the pitch of the note within the current octave range.
\$	Specifies that the note which follows must be flattened.

The following table summarises the Command Characters.

#	Specifies that the note which follows must be sharpened.		
0	Specifies the octave number to be used (followed by 0 to 8).		
112	Specifies the length of notes to be used.		
&	Specifies that a rest is to be played.		
_	Specifies that a tied note is to be played.		
Ν	Separates two numbers.		
V	Specifies the volume to be used (followed by 0 to 15).		
W	Specifies the volume effect to be used (followed by 0 to 7).		
U	Specifies that volume effects are to be used in the string.		
Х	Specifies the duration of the volume effect (followed by 0 to 65535).		
Т	Specifies the tempo of the music (followed by 60 to 240).		
()	Specifies that the enclosed phrase is to be repeated.		
!!	Specifies that the enclosed text is to be skipped over.		
Н	Specifies that the PLAY command must stop.		
М	Specifies the channel(s) to be used (followed by 0 to 63).		
Y	Specifies that a MIDI channel is to be used (followed by 1 to 16).		
Z	Specifies MIDI programming code (followed by code number).		

For further details, see also Chapter 19 of the Sinclair BASIC manual.

Format

• **PLAY** str-expr [,str-expr] [,str-expr]

See also

Chapter 19.

# POINT

Function

POINT is used to find out whether the colour at a particular position on the high-resolution screen is either an ink colour or a paper colour. POINT does not check the actual colour itself.

How to use POINT

POINT is followed by two numeric values *separated by a comma and enclosed in brackets*, for example

## 240 IF POINT(x, y)=1 THEN GO SUB 600

The two values following POINT are rounded to integers if necessary. The first value may range from 0 to 255 and defines the horizontal coordinate of a pixel on the screen. The second value may range from 0 to 175 and defines the vertical component of the coordinate. POINT the returns 1 if the pixel at the defined position is ink colour or 0 if it is paper colour.

Format

• **POINT** (int-num-expr,int-num-expr)

See also

Chapter 17.

# POKE

Statement/Command

POKE is used to change the value of the byte at a particular address in memory. Values are normally POKEd to memory locations in order to produce actions not given by BASIC keywords.

How to use POKE

POKE is used to form a statement in a program or as a command. It is followed by two numeric values *separated by a comma*, for example

### POKE 23609,255

The two values following POKE are rounded to the nearest integers if necessary. The first value may then range from 0 to 65535 and is an address in RAM (Note: The BASIC ROM inhabits the range from 0 to 16384 and is read-only, so POKEing to that range will have no effect). The second value may range from -255 to 255 and is the byte to be written to the defined address.

In the above example, 255 is POKEd to address 23609, which controls the sound produced when a key is pressed. A value of 255 gives a long beep instead of the normal click, with other values producing a shorter beep.

Format

• **POKE** int-num-expr,int-num-expr

See also

Chapter 14.

# PRINT

Statement/Command

PRINT displays data on the screen. The data may be any single character or a sequence of characters. A PRINT statement may incorporate other keywords to define the position and colour of the data.

How to use PRINT

PRINT may be used alone or it may be followed by data. This data may be in the form of any numeric or string expressions, or a mixture of these.

When using PRINT with data, two or more separate items must *each be separated by a semicolon, comma or apostrophe.* 

Certain other keywords may be inserted in any order between PRINT and the data, provided each statement formed by the keyword *ends in a semicolon*. These keywords are CHR\$, TAB, AT, INK, PAPER, FLASH, BRIGHT, INVERSE or OVER.

PRINT with strings

PRINT alone or followed by a null string ("") displays a blank line and moves the print-position to the beginning of the next line.

PRINT followed by a string constant (any characters within double quote marks) displays the characters as they appear between the quote marks. The command

PRINT "3/542/76/32"

displays

3/542/76/32

PRINT followed by a string variable or expression displays the string or strings

they represent.

PRINT with numbers

PRINT followed by any numeric expression displays the expression's value. Numbers are displayed in decimal notation with up to eight significant digits and no trailing zeros after the decimal point.

Very large numbers and very small numbers are displayed in a shorter scientific notation, as two figures separated by the letter E. This indicates a number in which the first part (the mantissa) is multiplied by 10 to the power of the second part (the exponent). The command

### PRINT 3/542/76/21

for example displays

### 3.4680798E-6

PRINT formatting with punctuation signs

PRINT followed by items of data separated by a semicolon displays the items placed next to each other without a space. The command

PRINT 1;2;3

displays

### 123

PRINT followed by items of data separated by a comma displays each item at the beginning or in the middle of the a line depending on the position of the first item. The command

PRINT 1,2,3

displays

1	2
3	

PRINT followed by items of data separated by an apostrophe displays the item after the apostrophe at the beginning of the next line. The command

### PRINT 1'2'3

displays

1 2 3

If a PRINT statement or command ends with a semicolon, comma or apostrophe, then the item displayed by the next PRINT statement is affected in the same way.

PRINT and other keywords

PRINT may be followed by TAB, a numeric value, a semicolon and then an item of data, for example

### 60 PRINT TAB x;a\$

The value following TAB (x above) is rounded to the nearest integer if necessary and is then divided by 32 and the remainder returned to give a value of between 0 and 31. The item of data is then displayed at this column position in the same or the next line.

PRINT may be followed by AT and then two numeric values *separated by a comma*, a semicolon and an item of data. For example,

### 50 PRINT AT 1,c;"Data"

The first value (l above) may range from 0 to 21 and defines the number of the line or row in which the data will be displayed. The second value (c above) may range from 0 to 31 and defines the number of the column in which the first

character or digit of the data will be displayed. Non-integer values are accepted and rounded to the nearest integer. The command

### PRINT AT 11,16;"\*"

displays a star in the centre of the screen.

PRINT may also be followed by one or more CHR\$ functions. See CHR\$ for more details.

PRINT and colour keywords

The display produced by PRINT is affected by colour statements or commands given by INK, PAPER, FLASH, BRIGHT, INVERSE and OVER that are currently in operation. PRINT may also be followed by one or more of these six statements *each followed by a semicolon* before the item of data, for example

### 50 PRINT AT 11,16;INK 2;FLASH 1;"\*"

The item of data is then displayed with the attributes specified by the colour keyword(s). These attributes are local and only apply to the item displayed. Following execution of the PRINT statement, they revert to their default or previously declared global values.

Format

• **PRINT** [**TAB** int-num-expr;] [**AT** int-num-expr,int-num-expr;] [**CHR\$** (int-num-expr);] [num-expr] [string-expr] [;] [,] [']

See also

Chapter 2, Chapter 15.

# RANDOMIZE

#### Command/Statement

RANDOMIZE is used in conjunction with RND to produce sequences of numbers that are either random or predictable.

How to use RANDOMIZE

RANDOMIZE is used either to form a statement in a program or as a command. It is optionally followed by a numeric value, for example

#### RANDOMIZE 1

#### 10 RANDOMIZE

The value following RANDOMIZE is rounded to the nearest integer if necessary and may then range from 0 to 65535. A value greater than 0 sets the system variable SEED to this value, following which RND always generates the same sequence of numbers. The actual sequence depends on the value of RANDOMIZE.

If RANDOMIZE is followed by 0 or no value, then SEED is given the value of another system variable called FRAMES, which counts the frames displayed since the BASIC was last reset. As FRAMES changes 50 times a second, the sequence of numbers generated by RND following RANDOMIZE or RANDOMIZE 0 is highly random.

If RANDOMIZE is not used, **RND** generates the same sequence of numbers from reset and after **NEW**.

Format

• **RANDOMIZE** [int-num-expr]

See also

Chapter 11.

# READ

Statement/Command

READ is used in conjuntion with DATA to assign values to variables using the values in a DATA statement.

How to use READ

READ is normally used to form a statement in a program. It is followed by one or more numeric variables or string variables *each separated by a comma*, for example

#### 20 READ a\$, x

When READ is first executed, it takes the same number of values as there are variables from the first DATA list and assigns the values to the variables in order. When READ is next executed, the next set of DATA values is assigned to the variables named in the READ statement and so on.

For further details, see DATA.

Format

- **READ** num-var [,num-var] [,string-var]
- **READ** string-var [,num-var] [,string-var]

See also

Chapter 6.

## **REM** REMark

#### Statement

REM is used to put remarks or reminders into your program. These may be the title and author of the program, and explanations of lines in the program such as the purpose of a variable. The remarks play no part in the running of the program and can be seen only in the listing.

How to use REM

REM forms either a line of its own in a program or the last statement in a line. It is followed by any remark that can be keyed in as required, for example

#### 80 INPUT n\$: REM n\$ is a name

When the BASIC encounters REM, it ignores everything that follows REM on the line.

Format

• **REM** any characters

See also

Chapter 2.

# RESTORE

Statement/Command

RESTORE is used in conjunction with READ and DATA to make READ take values from a particular DATA statement instead of the first or next DATA statement in the program.

How to use RESTORE

RESTORE normally forms a statement in a program. It is optionally followed by a numeric value, for example

#### 160 RESTORE 800

The value is rounded down to the nearest integer if necessary, and should then be the number of a line in the program containing a DATA statement. Following RESTORE, the next READ statement will assign the value contained in this DATA statement. If the numbered line does not exist or does not contain a DATA statement, then READ goes to the next DATA statement after this line. If there are no further DATA statements in the program, then error E (Out of DATA) results.

If RESTORE is followed by 0 or no value, then the next **READ** statement goes to the first **DATA** statement in the program.

Format

• **RESTORE** [int-num-expr]

See also

Chapter 6.

# RETURN

Statement/Command

RETURN is used to terminate a subroutine and return the computer to the main program or a previous subroutine.

How to use RETURN

RETURN is normally used to form a statement in a program. It is used alone at the end of a subroutine, for example

#### 1080 RETURN

On execution, the program branches to the statement following the last GO SUB executed.

See GO SUB for further details.

Format

• RETURN

See also

Chapter 5.

## **RND** RaNDom number

Function

RND is used to generate a random number.

How to use RND

RND is used alone in a statement or command, for example

#### 60 LET x=RND

RND then returns a random number less than 1 and greater than or equal to 0.

When BASIC is reset or started, or NEW is used, numbers are subsequently returned by RND in the same sequence. The sequence is generated by taking the powers of 75 (75, 75\*75, 75\*75\*75 and so on) dividing each power by 65537 and using the remainder only, then subtracting 1 from the remainder and dividing this result by 65536.

If a more random sequence or another fixed sequence is required, then use **RANDOMIZE** before RND.

Random whole numbers

Many of the BASIC's statements and functions, such as INK and CHR\$, round numbers to the nearest integer and RND may be used with them directly, for example

#### INK RND\*7

produces an ink colour at random. Others require integers and any whole number from 1 to x is given by

INT  $(RND^*x)+1$ 

To generate a random integer from 0 to x, use

INT (RND\*x+0.5)

Format

### • RND

See also

Chapter 11.

# RUN

Command/Statement

RUN makes a program run, normally from the first line.

How to use RUN

RUN may be used as a direct command or it may form a statement in a program line. It is optionally followed by a numeric value, for example

#### RUN 50

If no value follows RUN then the program runs from the first line. If a value is included, it is rounded down to the nearest integer if necessary and the program then runs from this line. If the line does not exist, the program runs from the next line in the program. Note that RUN performs CLEAR before running the program, so that variable values are are erased. To avoid this, use GO TO followed by a line number.

If a program has been saved using LINE, then it runs automaticaly on loading and RUN is not required.

Format

• **RUN** [int-num-expr]

See also

Chapter 2.

# SAVE

Statement/Command

SAVE sends a program to a file in order to store it for later retrieval.

How to use SAVE

SAVE is normally used as a direct command but may form a statement in a program. It is followed by a filename which is a string value, for example

### SAVE "filename"

The filename may contain up to ten characters. Specifying an empty or null ("") filename will bring up a windows requester to prompt you for a filename. On execution, the message

### Start tape, then press any key

is displayed. On pressing any key, the program is saved to a file, and on conclusion, the report 0 Ok, 0:1 appears.

Automatic running

If the stored program is to run automatically on loading, then SAVE should be used in conjunction with LINE. The program name is followed by LINE and a numeric value, for example

### SAVE "filename" LINE 1

The value following LINE is rounded down if necessary, and should then be either 1 or the number of a line in the program. The program is then sent to the file in the same way as a normal SAVE. On loading, the program starts automatically from the line having defined the line number or, if no such line exists, from the next line in the program. In practice, using LINE 1 causes the whole program to run automatically. Note that an auto-running program will not clear variables or the screen (effectively, the program uses a GO TO instead of a RUN to start the program).

SAVE and the silicon disc

SAVE can also be used to store a file on the 128k Spectrum's silicon disc. To do this, insert a ! character between the keyword SAVE and the filename, for example

### SAVE !"udgs" CODE USR "A",21\*8

You can use any of the commands used with SAVE such as CODE, DATA or SCREEN\$.

The silicon disc stores just under 64kb of data, and is very useful for rapid access to stored information which would otherwise take up too much memory.

Saving other types of data

SAVE can also store other types of data - sections of memory and variable arrays can be stored, and SAVE also has a special command to save the screen display. See SAVE CODE, SAVE DATA, and SAVE SCREEN\$ for further information

Format

• **SAVE** string-expr [**LINE** int-num-expr]

See also

Chapter 20.

• 128k Commands in BASin

## **SCREEN\$**

#### Statement/Command

SCREEN\$ detects which character appears at a particular location on the screen.

How to use SCREEN\$

SCREEN\$ is followed by two numeric values *separated by a comma and enclosed in brackets*, for example

### 160 IF SCREEN\$(1,c)="X" THEN PRINT "CRASH"

The values following SCREEN\$ are rounded down to the nearest integer if necessary. The first value (l above) may then range from 0 to 21 and gives the line (or row) number of a position on the screen. The second value (c above) may range from 0 to 31 and gives the column number of the position. SCREEN\$ then returns the character displayed at this position as a string constant (the character in quote marks, "X" above for example). If no character is present (or recognisable) at this position, SCREEN\$ returns a null (empty) string (""). SCREEN\$ cannot detect user-defined graphics unless they are identical to regular alpha-numeric characters.

Note that SCREEN\$ may also be used with SAVE and LOAD to store the screen display in a file and also to load it from that file. See SAVE SCREEN\$ and LOAD SCREEN\$ for further details.

Format

• **SCREEN\$** (int-num-expr, int-num-expr)

See also

Chapter 15.

## SGN SIGN

Function

SGN indicates whether a number is positive, negative or zero.

How to use SGN

SGN is followed by a numeric value, for example

#### 50 LET x=SGN y

An expression must be *enclosed in brackets*. SGN then returns 1 if the value of the argument (y above) is positive, -1 if it is negative and 0 if it is zero.

Format

- SGN num-const
- SGN num-var
- **SGN** (num-expr)

See also

Chapter 9.

## SIN SINe

Function

SIN gives the sine of an angle.

How to use SIN

SIN is followed by a numeric value, for example

80 LET x=SIN y

A expression must be *enclosed in brackets*. The value following SIN is the angle in *radians*, and SIN returns the sine of the angle. Degrees can be converted into radians by multiplying by PI/180.

Note that SIN returns a positive value for angles between 0 and 180 degrees, and a negative value for angles between 180 and 360 degrees.

Example

The command

PRINT SIN (30\*PI/180)

displays 0.5, the sine of 30 degrees.

Format

- SIN num-const
- SIN num-var
- SIN (num-expr)

See also

Chapter 10.

# SQR SQare Root

Function

SQR gives the square root of a number.

How to use SQR

SQR is followed by a numeric value, for example

### 70 LET x=SQR y

An expression must be *enclosed in brackets*. The value following SQR (y above) must be greater than zero, and SQR returns its square root.

Format

- SQR num-const
- SQR num-var
- SQR (num-expr)

See also

Chapter 9.

# STEP

See FOR, also Chapter 4.

# **STOP**

Statement/Command

STOP halts a program at a particular point. It may be necessary to use STOP to end the main section of a program in order to confine subroutines to a seperate section. STOP is also valuable in debugging a program.

How to use STOP

STOP is normally used to form a statement in a program. It is used on its own, for example

### 650 STOP

On execution, the program stops and the report

### 9 STOP statement

appears with the line and statement at which the program halted.

Debugging procedures, such as displaying and changing the values of variables, may then be undertaken. Entering **CONTINUE** subsequently causes the program to resume at the next statement with the new values.

Format

#### • STOP

See also

Chapter 2, Chapter 3.

## STR\$ STRing

Function

STR\$ converts a number into a string.

How to use STR\$

STR\$ is followed by a numeric value, for example

### 90 LET a\$=STR\$ x

An expression must be *enclosed in brackets*. STR\$ returns the value of its argument (x above) as a string constant. If x were assigned the value of 65, then the above statement assigns a\$ the value "65".

Format

- **STR\$** num-const
- **STR\$** num-var
- **STR\$** (num-expr)

See also

Chapter 9.

## TAB TABulate

PRINT/INPUT modifier

TAB changes the position on a line that **PRINT** and **INPUT** items appear.

How to use TAB

TAB is followed by a numeric value, for example

```
PRINT TAB 15;"Columm 15"
```

The value following TAB is rounded down to the nearest integer if necessary, and may range from 0 to 31 - values greater than 31 will be divided by 32 and the remainder used as the TAB value. Any characters subsequently displayed will appear at that column on the screen.

TAB can also be used in exactly the same manner with **INPUT** items.

Format

• **TAB** int-num-expr

See also

LPRINT; PRINT; Chapter 15.
## **TAN** TANgent

Function

TAN gives the tangent of an angle.

How to use TAN

TAN is followed by a numeric value, for example

### 130 LET X=TAN Y

An expression must be *enclosed in brackets*. The value following TAN is the angle in *radians*, and TAN returns the tangent of the angle. Degrees may be converted to radians by multiplying by PI/180.

Note that TAN returns a positive value for angles between 0 and 90 degrees. For angles between 90 and 180 degrees and angles between 270 and 360 degrees, TAN returns a negative value.

Format

- TAN num-const
- **TAN** num-var
- TAN (num-expr)

See also

Chapter 10.

# THEN

Statement

THEN Separates the condition from the code to be executed in an IF statement.

How to use THEN

THEN is preceded by IF, and a condition. It is followed by one or more statements, for example

10 IF x=0 THEN PRINT "YOU LOSE!" : STOP

If the condition (X=0 above) is true then the program PRINTs "YOU LOSE!" and then executes a STOP statement. If x is any other value then neither of the statements following THEN will be executed, and the program will continue on to the next line.

Format

• IF cond THEN statement [: statement]

See also

Chapter 3.

# ТО

String Slicer

TO has two different uses in BASIC. It is used in conjunction with FOR to set up a FOR-NEXT loop (see FOR for further details) and it is also used in string slicing (the splitting up of strings into smaller sub-strings).

How to use TO for string slicing

TO is used to define the first and last characters of a sub-string within a main string. TO is preceded by a string value, an opening bracket, then an optional numeric value. It is followed by an optional numeric value and then a closing bracket, for example

### 80 PRINT a\$(4 TO 7)

A string expression must also be *enclosed in brackets*. The string value (a\$ above) is the string that is to be sliced. The two numeric values (4 and 7) define the positions of the first and last characters of the sub-string within the string. TO then returns the sub-string (characters 4 to 7 inclusive, of a\$).

The first numeric value has a default value of 1 and the second has a default value equal to the position of the last character in the string. The first value can therefore be omitted if the sub-string begins with the first character in the string, and the second value can be omitted if the sub-string ends with the last character in the string.

Format

- string-const ([num-expr] **TO** [num-expr])
- string-var ([num-expr] **TO** [num-expr])
- (string-expr) ([num-expr] **TO** [num-expr])

See also

Chapter 4, Chapter 8.

## USR User SubRoutine

Function

USR is used to call a machine code subroutine that has been placed in memory at a specified address. It is also used to place the data for a user-defined graphics character in the reserved locations at the top of the memory.

USR and machine code

To use machine code, USR is followed by a numeric value, for example

### 80 PRINT USR 65000 100 RANDOMIZE USR 65000

An expression should be *enclosed in brackets*. The value following USR is rounded down to the nearest integer and is then the starting address in the memory at which the machine code subroutine has been placed. Any statement containing USR then calls the subroutine at this address and USR returns the contents of the BC register pair. RANDOMIZE USR or RESTORE USR, for example runs the subroutine only (though bear in mind that any command that takes a numeric parameter will act on the result of the USR call), whereas PRINT USR additionally displays the BC register value.

USR and user-defined graphics

To create user-defined graphics, USR is used with POKE. It is followed by a string constant or variable to return an address for the POKE, for example

### 50 POKE USR "a",255

the string value following USR may be a single letter ranging from A to U, or a to u, capital letters not being distinguished from lower-case letters.

USR then returns the starting address of one of the 21 sections of memory reserved for user-defined graphics. Each section contains eight addresses to

which eight bytes are **POKE**d to create one graphics character. The bytes may be given in decimal form or in binary (see **BIN**).

Format

- **USR** int-num-const
- **USR** int-num-var
- **USR** (int-num-expr)
- **USR** string-const
- **USR** string-var
- **USR** (string-expr)

See also

Chapter 14, Chapter 26.

## VAL VALue

Function

VAL changes a string with a numeric value contained within into a number.

How to use VAL

VAL is followed by a string constant or variable, for example

### 70 LET x=VAL a\$

The value of the string constant or variable is stripped of its quotation marks, and must then be a numeric value. VAL then evaluates this, returning it as a numeric value.

### Examples

If a\$ has the value "435" then the above statement assigns a value of 435 to x. However, VAL can also evaluate expressions, for example

### 10 INPUT a\$,x 20 PRINT VAL a\$

The string value that is assigned to a\$ should be an expression using x, for example " $X^*X$ ". A numeric value is then assigned to x, for example 5. VAL strips the quotation marks from the string value to get  $X^*X$  and evaluates it using the value assigned to x, displaying the result 25.

Format

- VAL string-const
- VAL string-var
- VAL (string-expr)

See also

Chapter 9.

### VAL\$ VALue (string)

Function

VAL\$ evaluates a string as a string expression.

How to use VAL\$

VAL\$ is followed by a string variable, for example

```
130 PRINT VAL$ a$
```

The value of the string variable is stripped of its quotation marks, and must then be a string expression. VAL\$ evaluates the expression and returns the value as a string constant.

Examples

Try this program

10 INPUT a\$,x\$ 20 PRINT VAL\$ a\$

The string value that is assigned to a\$ should be an expression using x\$, for example "x\$+x\$". A string value is then assigned to x\$, for example "DO". VAL\$ strips the quotes of the value of a\$ to get x\$+x\$ and evaluates it using the value assigned to x\$, displaying the result DODO.

Format

- VAL\$ string-var
- VAL\$ string-const
- VAL\$ (string-expr)

## VERIFY

Command/Statement

VERIFY checks that a program has been correctly stored following SAVE.

How to use VERIFY

VERIFY is normally used as a direct command in exactly the same way as LOAD and is followed by the program name, for example

```
VERIFY "filename"
```

When the VERIFY process is started, the name of every file is displayed and any program found that has the same name is compared with the program in memory. If the two are found to be the same, byte for byte, the report

### 0 OK, 0:1

is given.

VERIFY CODE and VERIFY DATA

VERIFY CODE can be used in exactly the same way as LOAD CODE to verify that a section of memory information has been stored correctly. VERIFY DATA works in the same way as LOAD DATA to check that an array has been stored without error. See LOAD CODE and LOAD DATA for further details.

Format

- **VERIFY** string-expr
- **VERIFY** string-expr **CODE** [int-num-expr] [,int-num-expr]
- **VERIFY** string-expr **DATA** letter[\$]()

See also

Chapter 20.

# **Contact Retro Computers Ltd**

By email

Should you have any questions about BASin then you should send a message to the publisher.

The address is:

http://everychildcancode.com/contact-us/

Via Social Media

You can contact Retro Computers Ltd via Facebook and Twitter:

https://www.facebook.com/retrocomputersltd

https://twitter.com/sinclairzxvega

## **Credits**

BASin was written by **Paul Dunn**. The help files were written by **Paul E Collins** and Paul Dunn, with contributions from **Andrew Owen**, and were compiled into Microsoft HTML Help format by Paul E Collins.

BASin would not have been possible without the invaluable help of all those who reported bugs, suggested features, and assisted with the coding. The author would especially like to thank the SPIN team (**Mark Boyd**, **Woody**, and **Damien Guard**) for their help and encouragement - notably Mark Boyd's contributions to the user interface and assembler, Woody's help with the main emulation core, and Damien's help with the user interface - and for creating such fantastic fonts for the editor to use.

BASin for Beginners is published by Retro-Computers Ltd.

# The display area

The display area, to the right of the editor, shows the output from your program just as it would be displayed on a Spectrum screen.

On start-up, the Editor will appear, and after a short pause, the display will run through the Spectrum ROM's start-up sequence. The display will go from white, to black (with red vertical lines) and finally back to white with the message

### © 2015 Sinclair Research Ltd

at the bottom. The Editor will then become active, ready for programming.



Display capabilities

For BASIC programming purposes, the Spectrum display has three distinct regions:

- The **printable area**, which is the main central part of the screen and receives most text output.
- The lower part of the printable area, which consists of its bottom two lines and is typically used for INPUT prompts and error messages. Text can be printed here with PRINT #0.
- The **border**, which is a region of plain colour (set by the **BORDER** statement) surrounding the printable area. The lower part usually adopts the border colour.

Graphics can be drawn anywhere in the printable area, including the lower part. However, you cannot produce text or graphics in the border.

BASin, by default, displays the Spectrum screen as a  $320 \times 240$  pixel bitmap. On a higher display resolution, this can make the display appear quite small. To circumvent this limitation, the display window can be re-sized, by using the mouse to grab the strip between the editor and the display area. Aspect ratio is always preserved.

To learn about the resolution and colour depth of the Spectrum display, see What is a Sinclair Spectrum?.

# **Editor Fonts tab**

The **Editor Fonts** tab can be found in the Options window.

Options			X
Emulation	Display	Sound	Files
BASin	Editor fonts	Error	Reporting

BASin's editor can use other fonts if you wish, rather than the standard sinclair font. These fonts are any file (either a binary dump or a **. bsc** file) which holds 768 bytes. This data is then used to form the font used by the editor.

Fonts can be created and edited using the UDG Editor. They can also be loaded by your programs and used (with a little modification to the CHARS system variable) as the default font by BASIC.

Font Folder

BASin will, on startup and when the options window opens, search for fonts that can be used by the editor. BASin will search the folder specified here, and recurse any subdirectories in that folder. The list of fonts gathered will be used in the **Font Filename** option.

If no fonts can be found, the only option available to you will be the default Sinclair Font which is built into BASin and into the ROM.

Changing the Editor Font

Clicking the **Font Filename** list-box will bring up a small tree-view of all the folders in the **Font Folder**, and all the fonts found in them. Folders with no fonts will not be shown. You can scroll through the list and find a font which you like the look of, and select it to activate it.



You can get a preview image of your font at the bottom of the window. This will show all 96 characters used in the character set. As you can see, *some* fonts are not really suitable for use in the editor, but will look very nice in the right context - perhaps a game.



Syntax Highlighting

As you work in BASin, with Syntax highlighting enabled you will notice that some words are coloured differently. This is more than just cosmetic - you can identify errors (especially with variables) in your code quite easily this way.

You can enable or disable syntax highlighting with the checkbox below the fonts list. Setting the options opens a new window:



The options here are applicable for each element that you can highlight. The checkbox next to each item can turn on or off highlighting for that element, and the colour palette below chooses the colour you want for that item. You can also choose for it to be either **bold-faced** or *Italicised*.

The list contains these items:

- **Keywords** Any reserved word used by BASIC, such as **PRINT**, **LET**, **GO** TO etc.
- **Functions** User defined functions created with DEF FN and referenced throughout the program with the FN keyword.
- **Comments** Any **REM** statement
- **Symbols** Any symbol character i.e., from ASCII 33, "!" character through to 127, the Copyright symbol.
- **Variables** Variables that are valid, and have a corresponding entry in the VARS space in memory.
- **Undefined Variables** Any variable name that has either no corresponding LET statement or whose LET statement has not yet been executed at runtime will be an invalid variable name.

- Numbers Any digits (except line numbers).
- Line Numbers Line numbers at the start of a line.
- **Strings** All string literals any string of any characters enclosed in quote marks.
- **Editor Foreground** The colour use to render any non-highlighted text in the editor.
- **Editor Background** The colour used to render the background (default grey) and also used as a reference colour for the darker line-number gutter.

# The Tape Image file formats

TAP and TZX files store images of Sinclair Spectrum tapes. Because the Spectrum stored it's data by outputting pulses of sound to an audio device, emulators originally used wave recordings which recreated the sounds on the tape. Later, the **.tap** format came along. It was found that the format of these sounds matched identically with the bits of bytes being saved, and that when a certain section of the Spectrum ROM was entered, you could discover exactly what was being saved and what was being loaded. To this end, the .tap format stored a copy of each block of memory saved by the ROM. This has the advantage of being a lot smaller than a corresponding wave-sound file.

Each file on a tape comes, most usually, in two parts. The "header" gives information to the Spectrum about the file that is coming from the tape, such as the name, the type of data (a program, a memory block or a variable) and the size of the data. Following this header is the main body of the file.

**Note:** It is worth mentioning that BASin treats each header and body pair as a single file, unless the data has no header in which case it is flagged as "headerless".

Because the data saved to the tape image is a direct copy of the bytes saved, it is trivial to just "inject" the data back to memory – after all, the header provides information about where it came from originally. Unfortunately, the **.tap** format cannot reproduce the sounds made by the newer, faster loaders employed by games companies in their nigh-on constant fight to get quicker load times and prevent software piracy. The .tzx format, however, can reproduce any sound the Spectrum can produce. The .tzx format, as far as BASin is concerned, is very similar to the **.tap** format – it just adds a couple of extra bytes to the blocks with some more detailed information about it.

The .TAP File format

For the technically minded, the .tap format for a block is as follows:

Byte offset

01	Length of the block
2	Flag byte (0 for a header, 255 for the body of the file)
3xx	These bytes are the data of the block itself - either the header or the main body
Last Byte	Checksum Byte

### The .TZX File format

### The .tzx equivalent is as follows:

Byte offset	Contents
0	ID Byte (16 for a ROM-Saved block)
12	The length of the pause after this block in milliseconds (1000ms default)
34	Length of the block
5	Flag byte (0 for a header, 255 for the body of the file)
6xx	These bytes are the data of the block itself - either the header or the main body
Last Byte	Checksum Byte

The header format is similar to that employed by

## .bsc and .bsd files, as follows:

Byte offset	Program	Number Var	String Var	Memory Block	Notes

0	0	1	2	3	Data Type
110	Filename	Filename	Filename	Filename	10 Characters
1112	Length	Length	Length	Length	
1314	Auto Start	Variable az	Variable az	Start Address	Various
1516	Program Length	Unused	Unused	Unused	

Note that the Number and String variable types have their "name" from A to Z stored in one byte – the high byte at position 14, and do not use the byte at position 13. Programs store their length twice – once for the whole block and once for just the BASIC text, without the variables.

# **Files tab**

The **Files** tab can be found in the **Options window**.

Options			
BASin	Editor fonts	Error	Reporting
Emulation	Display	Sound	Files

Files in BASin can be manipulated in various ways, and as these options become available, they can be set here. For this release, the options are:

Snapshots

#### Save as .z80 version ...

The .**Z80** format was developed by Gerton Lunter for his spectrum emulator, Z80. This format underwent a few revisions as his emulator gained new features, and the different versions are backwardly incompatible with eachother. For this reason, you can elect to save your .**Z80** Snapshots using any of the major revision types. The most common nowadays is probably the v3.5 option.

#### Save as hardware mode

When saving a snapshot, BASin will examine your program and silicon disc (if it has been used) and decide which hardware model to use to save (48k or 128k). This can be overriden here by choosing to save as one particular type (if there is a choice) or to ask you before it saves.

.bas files

#### **Break multi-statement lines**

Lines with more than one statement can be made more readable in .bas files (particularly when reading them in a text editor) by splitting the statements up so that only one statement occupies each line. This is done by

appending a backslash (" $\$ ") character to the end of each line, after the trailing colon (":"). This will not affect the way that BASin handles this file in the future, it is provided for better readability only.

Tape image files

### Trap LOAD/MERGE/VERIFY

This will toggle BASin's ability to load directly from tape images held in the Tape Creator Window, instead of loading files from your PC's Hard Disk. This can be useful for loading a complete tape into memory without having to manually extract the files first. While this option is enabled, you will not be able to load files from disk.

### **Trap SAVE operations**

As with the LOAD/MERGE/VERIFY option above, this will toggle BASin's ability to redirect all tape (and file) output to the currently loaded TZX file. The TZX file must be present in the Tape Creator Window.

### Rewind when tapes end

This option will, when loading from a tape, automatically rewind the tape when it reaches the end, by setting the current tape position in the Tape Creator Window to the first block. Be aware that continuous LOAD commands will continue to load from the tape until the user BREAKs the operation with the ESC key.

# SAVE CODE

#### Statement/Command

SAVE CODE sends a section of information in the memory to a file. The information can then be placed back in the memory using LOAD CODE.

How to use SAVE CODE

SAVE CODE may be used as a direct command or to form a statement in a program. SAVE is followed by a filename which is a string value, and then followed by CODE which is in turn followed by two numeric values *separated by a comma*, for example

### SAVE "picture" CODE 16384,6912

The filename following SAVE may contain up to ten characters. The two values following CODE are each rounded down to the nearest integer if necessary. The first then gives the starting address (16384 above) of the information in the memory, and the second value (6912) gives the number of bytes that are to be sent to the file in the same way as a program, using SAVE.

The information saved by the above command is the screen display area.

Format

• **SAVE** string-expr **CODE** int-num=expr,int-num-expr

See also

Chapter 20.

# LOAD CODE

### Command/Statement

LOAD CODE is used to load a section of memory with information that has been stored in a file. The information consists of a set of bytes, and these are sent to a set of addresses in memory. LOAD CODE can be used to load a display, or to load information for user-defined graphics, for example.

How to use LOAD CODE

LOAD CODE may be used as a direct command or it may form a statement in a program. LOAD is followed by a filename, which is a string value, and then CODE, for example

### LOAD "data" CODE

The filename following LOAD is the name of the information to be loaded and is subject to the same restrictions as program names (see LOAD. LOAD CODE then searches for the named information and when found, displays Bytes: followed by the name. The BASIC then loads the bytes into memory at the addresses from which they were saved. Any existing information is overwritten.

CODE may also be followed by one or two numeric values, *separated by a comma*, for example

### LOAD "picture" CODE 16384,6912

The values following CODE are rounded to the nearest integer and then define the starting address (16384 above) at which the named information is to be loaded, and the number of bytes (6912) that are to be sent to locations beginning at this address. If the number is wrong, the tape loading error report is given. If only one value follows CODE, it defines the starting address from which *all* the bytes are to be located.

The above example can also be carried out by the keywords LOAD SCREEN\$.

For details on storing bytes, see SAVE CODE.

Format

• LOAD string-expr CODE [int-num-expr] [,int-num-expr]

See also

Chapter 20.

## SAVE DATA

Statement/Command

SAVE DATA stores an array variable in a file. The array can then be loaded using LOAD DATA.

How to use SAVE DATA

SAVE DATA may be used to form a statement in a program or as a direct command. SAVE is followed by a filename, then DATA, a letter or letter with \$, and finally a pair of *empty brackets*, for example

450 SAVE "numbers" DATA n()

750 SAVE "names" DATA n\$()

The array's filename may contain up to ten characters. The letter or letter\$ following DATA us the name of the array in the program that is to be stored in a file. The array is then sent to the file in the same was as a program is using SAVE.

Format

• **SAVE** string-expr **DATA** letter[\$]()

See also

Chapter 20.

# **System Variables window**

The System Variables window is available from the View menu.

K System V	ariables	
Name	Address	Value 🔥
FLAGS	23611 [1]	192
TV_FLAG	23612 [1]	1
ERR_SP	23613 [2]	65360 (80,255)
LIST_SP	23615 [2]	0 (0,0)
MODE	23617 [1]	0
NEWPPC	23618 [2]	0 (0,0)
NSPPC	23620 [1]	255
PPC	23621 [2]	65534 (254,255)
SUBPPC	23623 [1]	1
BORDCR	23624 [1]	56
E_PPC	23625 [2]	0 (0,0)
VARS	23627 [2]	23755 (203,92)
Deer	101 00200	39064 /EE 091
Update on Ste	p 💌	Refresh Close

This window allows the user to inspect and modify the contents of the System Variables used by the BASIC to monitor certain aspects of the program. The list shows the contents of all the variables, and updates as dictated by the user. You can elect, using the drop down box at the bottom left of the window, to update:

- **Do not update** refreshes the list only when you click the **Refresh** button.
- **Update on step** refreshes the list each time you step through a BASIC statement.
- **Update every** *time period* refreshes the list after a set number of seconds.

If you have decided against any updating and set the window to "never update" then you can force an update using the "Refresh" button at the bottom right.

Each Sysvar occupies a byte or two of memory, and the address that they live in is to the right of their name, followed by their contents. In the case of two and three byte Sysvars, the value shown is the Word value of their combined bytes (Low Byte plus the High byte \* 256), with the individual component bytes in brackets after.

You can get an explanation of each variable by hovering the mouse over it's entry and waiting for the ToolTip to appear.

You can edit any of the variables by either right clicking and choosing "Edit this Sysvar", or by clicking the "Edit" button at the bottom of the window. In the main, this will produce an editor very similar to the Variables Window's editor for numeric variables, except in the case of the longer multi-byte system variables (such as STRMS) which will open the Memory Viewer in edit mode at the location of this variable.

Lastly, you can elect to add any of the variables to the Watch Window by right clicking and choosing "Watch this Sysvar".

# ZX printer output window

The ZX printer output window is available from the View menu.



This window holds all output sent to the ZX Printer via the LLIST, LPRINT and COPY commands, along with any commands that have their output sent to the printer stream.

**Note:** This window will only accept printer output when it is visible (open). If it is closed, then the printer will be "Offline" and will not display any new printer data until it is opened again.

As printer information is received by the emulated printer, it is interpreted graphically, and displayed at the bottom of the upper display area of the window. This graphical display area will automatically scroll up to accommodate more data as it is sent. You can use the scrollbar at the right of this display to review older output.

BASin can accelerate the ROM printing commands, by trapping the stream of bytes to be sent to the printer and converting them into graphics a lot quicker than by just trapping individual pixels. This behaviour can be turned on or off in the BASin Options Window, using the Emulation Tab. Also in that section of the options is an option to save the current printer output when you quit BASin - and to automatically reload it when you next start up.

Along the bottom of the window are a set of buttons which provide functionality for working with the printed output. These, from left to right, are:

#### Buttons

8

Opens the Print Preview Window with the ZX Printer output ready configured to print.

This button will simulate the "tearing off" of the paper currently printed, and give you a fresh clean output sheet. You will be given the option to save the output as a Windows Bitmap (.bmp) file before it is cleared.

•

This will scroll the graphical display up one line.

COPY

The ROM COPY command does not include the lower two lines of the display when printing. This button will swiftly send the entire display to the printer. There is a choice of two methods when using this button - the first will send just the Display File area of the Spectrum's memory, with no attributes. This is very similar to the way the ROM's COPY command works. The other option will send the attributes too, to get a colour screen grab.

#### Close

Closes the **ZX Printer Output window**. Note that when the printer window is closed, the printer is classed as "offline" and *no further output will be sent to it*.

# The Display Window menu

This menu controls the size and visibility of the display window, and provides an option to print the contents of the display window.

- **Visible**: Determines whether the display window is shown or hidden. This is useful for viewing output when editing, as the window will hide behind the editor if emulation is not running.
- **100%** / **200%**: Specifies the size of the display window.
- Force 1:1 Aspect: Maintains correct aspect ration (width to height) and thus prevents distortion of the image as you resize the window. This for example, keeps a circle onscreen looking circular rather than elliptical when sizing. (this is the same option as Maintain Aspect Ratio on the Display tab).

The Edit menu provides Cut, Copy, Paste and Delete commands. These are similar to the commands on the Edit menu in the editor, but they apply to the display window instead - in this context, these operations work on any INPUT command being executed.

**Note:** In the display window, text can only be selected with the keyboard - not with the mouse. Using the mouse is planned for a future release of BASin.

The Print option

The current display can be sent as-is to the Print Preview Window for sending to

your PC printer. The display will be sent exactly as it appears in the display window including any scaling effects.

# **Appendix D: Example programs**

This appendix contains some example programs.

The first of these programs requires a date to be input and gives the day of the week which corresponds to this date.

10	REM convert date to day
20	DIM d\$(7,6):
	REM days of week
30	FOR n=1 TO 7:
	READ d\$(n):
	NEXT n
40	DIM m(12):
	REM lengths of months
50	FOR n=1 TO 12:
	READ m(n):
	NEXT n
100	REM input date
110	INPUT "day?";day
120	INPUT "month?";month
130	INPUT "year (20th century onl
140	IF year<1901 THEN
	PRINT "20th century starts at
	GO TO 100

- 150 IF YEAR>2000 THEN PRINT "20th century ends at 2 GO TO 100
- 160 IF month<1 THEN GO TO 210
- 170 IF MONTH>12 THEN GO TO 210
- 180 IF year/4-INT(year/4)=0 THEN
  LET m(2)=29:
  REM leap year
- 190 IF day>m(month) THEN PRINT "This month has only "; GO TO 500
- 200 IF day>0 THEN
  - GO TO 300
- 210 PRINT "Stuff and nonsense. Gi
- 220 GO TO 500
- 300 REM convert date to number of
- 310 LET y=year-1901
- 320 LET b=365\*y+INT (y/4):
  - REM number of days to start o
- 330 FOR n=1 TO month-1: REM add on previous months

```
340 LET b=b+m(n):
     NEXT n
 350 LET b=b+day
 400 REM convert to day of week
 410 LET b=b-7*INT (b/7)+1
 420 PRINT day;"/";month;"/";year
 430 FOR n=6 TO 3 STEP -1:
     REM remove trailing spaces
 440 IF d$(b,n) <> " " THEN
     GO TO 460
 450 NEXT n
 460 LET e$=d$(b, TO n)
 470 PRINT" is a "; e$; "day"
 500 LET m(2)=28:
     REM restore February
 510 INPUT "again?", a$
 520 IF a$="n" THEN
     GO TO 540
 530 IF a$ <> "N" THEN
     GO TO 100
1000 REM days of week
1010 DATA "Mon", "Tues", "Wednes"
1020 DATA "Thurs", "Fri", "Satur",
```

```
1100 REM lengths of months
1110 DATA 31, 28, 31, 30, 31, 30
1120 DATA 31, 31, 30, 31, 30, 31
```

This program handles yards, feet and inches.

<pre>INPUT "yards?",yd,"feet?",ft,</pre>
GO SUB 2000:
REM print the values
PRINT '" = ";
GO SUB 1000:
REM the adjustment
GO SUB 2000:
REM print the adjusted values
PRINT
GOTO 10
REM subroutine to adjust yd,
LET ins=36*yd+12*ft+ins:
REM now everything is in inch
LET s=SGN ins:
LET ins=ABS ins:
REM we work with in positive,
LET ft=INT (ins/12):
LET ins=(ins-12*ft)*s:
```
REM now ins is ok
1080 LET yd=INT (ft/3)*s:
    LET ft=ft*s-3*yd:
    RETURN
2000 REM subroutine to print yd, f
2010 PRINT yd;"yd";ft;"ft";ins;"in
    RETURN
```

Here is a program to throw coins for the I Ching. (Unfortunately it produces the patterns upside down. but you might not worry about this.)

```
5 RANDOMIZE
10 FOR m=1 TO 6:
    REM for 6 throws
20 LET c=0:
    REM initialize coin total to 0
30 FOR n=1 TO 3:
    REM for 3 coins
40 LET c=c+2+INT (2*RND)
50 NEXT n
60 PRINT " ";
70 FOR n=1 TO 2:
    REM 1st for the thrown hexagra
80 PRINT "____";
```

```
90 IF c=7 THEN
    PRINT " ";
100 IF c=8 THEN
    PRINT " ";
110 IF c=6 THEN
    PRINT "X";:
    LET c=7
120 IF c=9 THEN
    PRINT "0";:
    LET C=8
130 PRINT "____
                н.
140 NEXT n
150 PRINT
160 INPUT a$
170 NEXT m:
    NEW
```

To use this, type it in and run it, and then press Enter five times to get the two hexagrams. Look these up in a copy of the Chinese Book of Changes. The text will describe a situation and the courses of action appropriate to it, and you must ponder deeply to discover the parallels between that and your own life. Press Enter a sixth time, and the program will erase itself this is to discourage you from using it frivolously.

Many people find the texts are always more apt than they would expect on grounds of chance; this may or may not be the case for you. In general, computers are pretty godless creatures.

Here is a program to play 'Pangolins'. You think up an animal, and the computer tries to guess what it is, by asking you questions that can be answered 'yes' or 'no'. If it's never heard of your animal before, it asks you to give it some question that it can use next time to find out whether someone's given it your new animal.

```
5 REM Pangolins
 10 LET nq=100:
    REM number of questions and a
 15 DIM q$(nq,50):
    DIM a(nq,2):
    DIM r$(1)
 20 LET qf=8
 30 FOR n=1 TO qf/2-1
 40 READ q$(n):
    READ a(n, 1):
    READ a(n, 2)
 50 NEXT n
 60 FOR n=n TO qf-1
 70 READ q$(n):
    NEXT n
100 REM start playing
110 PRINT "Think of an animal.","
120 PAUSE 0
130 LET c=1:
    REM start with 1st question
```

```
140 IF a(c, 1)=0 THEN
    GO TO 300
150 LET p$=q$(c):
    GO SUB 910
160 PRINT "?":
    GO SUB 1000
170 LET inn=1:
    IF r$="y" THEN
    GO TO 210
180 IF r$="Y" THEN
    GO TO 210
190 LET inn=2:
    IF r$="n" THEN
    GO TO 210
200 IF r$<>"N" THEN
    GO TO 150
210 LET c=a(c,inn):
    GO TO 140
300 REM animal
310 PRINT "Are you thinking of"
320 LET P$=q$(c):
    GO SUB 900:
    PRINT "?"
```

```
330 GO SUB 1000
340 IF r$="y" THEN
    GO TO 400
350 IF r$="Y" THEN
    GO TO 400
360 IF r$="n" THEN
    GO TO 500
370 IF r$="N" THEN
    GO TO 500
380 PRINT "Answer me properly whe
    GO TO 300
400 REM guessed it
410 PRINT "I thought as much.": G
500 REM new animal
510 IF qf>nq-1 THEN
    PRINT "I'm sure your animal i
    GO TO 800
520 LET q$(qf)=q$(c):
    REM move old animal
530 PRINT "What is it, then?":
    INPUT q$(qf+1)
540 PRINT "Tell me a question whi
550 LET p$=q$(qf):
```

```
PRINT " and"
560 LET p$=q$(qf+1):
    GO SUB 900:
    PRINT " "
570 INPUT s$:
   LET b=LEN s$
580 IF s$(b)="?" THEN
    LET b=b-1
590 LET q$(c)=s$(TO b):
    REM insert question
600 PRINT "What is the answer for
610 LET p$=q$(qf+1):
    GO SUB 900:
    PRINT "?"
620 GO SUB 1000
630 LET inn=1:
    LET io=2:
    REM answers for new and old a
640 IF r$="y" THEN
    GO TO 700
650 IF r$="Y" THEN
    GO TO 700
```

GO SUB 900:

```
660 LET inn=2:
   LET io=1
670 IF r$="n" THEN
    GO TO 700
680 IF r$="N" THEN
    GO TO 700
690 PRINT "That's no good. ":
    GO TO 600
700 REM update answers
710 LET a(c, inn)=qf+1:
   LET a(c,io)=qf
720 LET qf=qf+2:
    REM next free animal space
730 PRINT "That fooled me."
800 REM again?
810 PRINT "Do you want another go
   GO SUB 1000
820 IF r$="y" THEN
    GO TO 100
830 IF r$="Y" THEN
    GO TO 100
840 STOP
900 REM print without trailing sp
```

```
910 FOR n=50 TO 1 STEP -1
 920 IF p$(n)<>" " THEN
     GO TO 940
 930 NEXT n
 940 PRINT p$(TO n);:
    RETURN
1000 REM get reply
1010 INPUT r$:
     IF r$="" THEN
     RETURN
1020 LET r$=r$(1):
     RETURN
2000 REM initial animals
2010 DATA "Does it live in the sea
2020 DATA "Is it scaly", 3, 5
2030 DATA "Does it eat ants", 6, 7
2040 DATA "a whale", "a blancmange
```

Here is a program to draw a Union Flag.

905 PRINT " ";

5 REM union flag 10 LET r=2: LET w=7:

```
LET b=1
 20 BORDER 0:
    PAPER b:
    INK w:
    CLS
 30 REM black in bottom of screen
 40 INVERSE 1
 50 FOR n=40 TO 0 STEP -8
 60 PLOT PAPER 0;7,n:
    DRAW PAPER 0;241,0
 70 NEXT n:
    INVERSE 0
100 REM draw in white parts
105 REM St. George
110 FOR n=0 TO 7
120 PLOT 104+n,175:
    DRAW 0,-35
130 PLOT 151-n,175:
    DRAW 0,-35
140 PLOT 151-n,48:
    DRAW 0,35
150 PLOT 104+n,48:
    DRAW 0,35
```

```
160 NEXT n
200 FOR n=0 TO 11
210 PLOT 0,139-n:
    DRAW 111,0
220 PLOT 255,139-n:
    DRAW -111,0
230 PLOT 255,84+n:
    DRAW -111,0
240 PLOT 0,84+n:
    DRAW 111,0
250 NEXT n
300 REM St. Andrew
310 FOR n=0 TO 35
320 PLOT 1+2*n,175-n:
    DRAW 32,0
330 PLOT 224-2*n,175-n:
    DRAW 16,0
340 PLOT 254-2*n,48+n:
    DRAW-32,0
350 PLOT 17+2*n,48+n:
    DRAW 16,0
360 NEXT n
370 FOR n=0 TO 19
```

```
380 PLOT 185+2*n,140+n:
DRAW 32,0
```

390 PLOT 200+2\*n,83-n: DRAW 16,0

```
400 PLOT 39-2*n,83-n:
DRAW 32,0
```

```
410 PLOT 54-2*n,140+n:
DRAW -16,0
```

```
420 NEXT n
```

```
425 REM fill in extra bits
```

```
430 FOR n=0 TO 15
```

```
440 PLOT 255,160+n:
DRAW 2*n-30,0
```

```
450 PLOT 0,63-n:
DRAW 31-2*n,0
```

```
460 NEXT n
```

```
470 FOR n=0 TO 7
```

```
480 PLOT 0,160+n:
DRAW 14-2*n,0
```

```
485 PLOT 255,63-n:
DRAW 2*n-15,0
```

```
490 NEXT n
```

```
500 REM red stripes
```

```
510 INVERSE 1
520 REM St George
530 FOR n=96 TO 120 STEP 8
540 PLOT PAPER r;7,n:
    DRAW PAPER r;241,0
550 NEXT n
560 FOR n=112 TO 136 STEP 8
570 PLOT PAPER r;n,168:
    DRAW PAPER r;0,-113
580 NEXT n
600 REM St Patrick
610 PLOT PAPER r;170,140:
    DRAW PAPER r;70,35
620 PLOT PAPER r;179,140:
    DRAW PAPER r;70,35
630 PLOT PAPER r;199,83:
    DRAW PAPER r;56,-28
640 PLOT PAPER r;184,83:
    DRAW PAPER r;70,-35
650 PLOT PAPER r;86,83:
    DRAW PAPER r; -70, -35
660 PLOT PAPER r;72,83:
    DRAW PAPER r; -70, -35
```



If you're not British, have a go at drawing your own flag. Tricolours are fairly easy, although some of the colours - for instance the orange in the Irish flag - might present difficulties. If you're an American, you might be able to fit the character \* in.

Here is a program to play hangman. One player enters a word, and the other guesses.

5	REM hangman
10	REM set up screen
20	INK 0:
	PAPER 7:
	CLS
30	LET x=240:
	GO SUB 1000:
	REM draw man
40	PLOT 238,128:
	DRAW 4,0:

```
100 REM set up word
110 INPUT w$:
  REM word to guess
120 LET b=LEN w$:
   LET v$=" "
130 FOR n=2 TO b:
   LET v$=v$+" "
140 NEXT n:
   REM v$=word guessed so far
150 LET c=0:
    LET d=0:
    REM guess & mistake counts
160 FOR n=0 TO b-1
170 PRINT AT 20,n;" ";
180 NEXT n:
  REM write 's instead of lett
200 INPUT "Guess a letter ";g$
210 IF g$="" THEN
    GO TO 200
220 LET g$=g$(1):
    REM 1st letter only
230 PRINT AT 0,c;g$
```

REM mouth

```
240 LET c=c+1:
    LET u$=v$
250 FOR n=1 TO b:
    REM update guessed word
260 IF w$(n)=g$ THEN
    LET v$(n)=g$
270 NEXT n
280 PRINT AT 19,0;v$
290 IF v$=w$ THEN
    GO TO 500:
    REM word guessed
300 IF v$<>u$ THEN
    GO TO 200:
    REM guess was right
400 REM draw next part of gallows
410 IF d=8 THEN
    GO TO 600:
    REM hanged
420 LET d=d+1
430 READ x0, y0, x, y
440 PLOT x0, y0:
    DRAW x,y
450 GO TO 200
```

```
500 REM free man
```

- 510 OVER 1:
  - REM rub out man
- 520 LET x=240: GO SUB 1000
- 530 PLOT 238,128: DRAW 4,0: REM mouth
- 540 OVER 0: REM redraw man
- 550 LET x=146: GO SUB 1000
- 560 PLOT 143,129: DRAW 6,0, PI/2: REM smile
- 570 GO TO 800
- 600 REM hang man
- 610 OVER 1: REM rub out floor
- 620 PLOT 255,65: DRAW -48,0

```
630 DRAW 0,-48:
REM open trapdoor
```

```
640 PLOT 238,128:
    DRAW 4,0:
    REM rub out mouth
650 REM move limbs
655 REM arms
660 PLOT 255,117:
    DRAW -15, -15:
    DRAW -15,15
670 OVER 0
680 PLOT 236,81:
    DRAW 4,21:
    DRAW 4, 21
690 OVER 1:
    REM legs
700 PLOT 255,66:
    DRAW -15,15:
    DRAW -15, -15
710 OVER 0
720 PLOT 236,60:
    DRAW 4,21:
    DRAW 4,-21
730 PLOT 237,127:
    DRAW 6,0,-PI/2:
```

```
740 PRINT AT 19,0;w$
 800 INPUT "again? ";a$
 810 IF a$="" THEN
     GO TO 850
 820 LET a$=a$(1)
 830 IF a$="n" THEN
     STOP
 840 IF a$(1)="N" THEN
     STOP
 850 RESTORE:
     GO TO 5
1000 REM draw man at column x
1010 REM head
1020 CIRCLE x,132,8
1030 PLOT x+4,134:
     PLOT x-4,134:
     PLOT x,131
1040 REM body
1050 PLOT x,123:
     DRAW 0,-20
1055 PLOT x,101:
     DRAW 0,-19
```

REM frown

```
1060 REM legs
1070 PLOT x-15,66:
    DRAW 15,15:
    DRAW 15,-15
1080 REM arms
1090 PLOT x-15,117:
    DRAW 15,-15:
    DRAW 15,15
1100 RETURN
2000 DATA 120,65,135,0,184,65,0,91
2010 DATA 168,65,16,16,184,81,16,-
2020 DATA 184,156,68,0,184,140,16,
2030 DATA 204,156,-20,-20,240,156,
```

← Appendix C

Appendix E 🏓

# **Appendix E: Binary and hexadecimal**

This appendix describes how computers count, using the binary system.

Most European languages count using a more or less regular pattern of tens - in English, for example, although it starts off a bit erratically, it soon settles down into regular groups:

twenty, twenty one, twenty two, . . . twenty nine thirty, thirty one, thirty two, . . . thirty nine forty, forty one, forty two, . . . forty nine

and so on, and this is made even more systematic with the Arabic numerals that we use. However, the only reason for using ten is that we happen to have ten fingers and thumbs.

Instead of using the decimal system, with ten as its base, computers use a form of binary called hexadecimal (or hex, for short), based on sixteen. As there are only ten digits available in our number system we need six extra digits to do the counting. So we use A, B, C, D, E and F. And what comes after F? Just as we, with ten fingers, write 10 for ten, so computers write 10 for sixteen. Their number system starts off:

Hex	English
Θ	nought
1	one
2	two
:	:
:	:
9	nine

just as ours does, but then it carries on

A	ten
В	eleven
С	twelve
D	thirteen
E	fourteen
F	fifteen
10	sixteen
11	seventeen
:	:
:	:
9	twenty five
1A	twenty six
1B	twenty seven
:	:
:	:
1F	thirty one
20	thirty two
21	thirty three
:	:
:	:
9E	one hundred and fi
9F	one hundred and fi

one	hundred	and	si
one	hundred	and	si
	:		
one	hundred	and	ei
	:		
two	hundred	and	fi
two	hundred	and	fi
two	hundred	and	fi
	one one one two two two	one hundred one hundred : one hundred : two hundred two hundred	one hundred and one hundred and : one hundred and : two hundred and two hundred and

If you are using hex notation and you want to make the fact quite plain, then write 'h' at the end of the number, and say 'hex'. For instance, for one hundred and fifty eight, write '9Eh' and say 'nine E hex'.

You will be wondering what all this has to do with computers. In fact, computers behave as though they had only two digits, represented by a low voltage, or off (0), and a high voltage, or on (1). This is called the binary system, and the two binary digits are called bits: so a bit is either 0 or 1.

In the various systems, counting starts off

English	Decimal	Не
nought	Θ	Θ
one	1	1
two	2	2
three	3	3
four	4	4
five	5	5

six	6	6
seven	7	7
eight	8	8
nine	9	9
ten	10	А
eleven	11	В
twelve	12	С
thirteen	13	D
fourteen	14	E
fifteen	15	F
sixteen	16	10

The important point is that sixteen is equal to two raised to the fourth power, and this makes converting between hex and binary very easy.

To convert hex to binary, change each hex digit into four bits, using the table above.

To convert binary to hex, divide the binary number into groups of four bits, starting on the right, and then change each group into the corresponding hex digit.

For this reason, although strictly speaking computers use a pure binary system, humans often write the numbers stored inside a computer using hex notation.

The bits inside the computer are mostly grouped into sets of eight, or bytes. A single byte can represent any number from nought to two hundred and fifty five (11111111 binary or FF hex), or alternatively any character in the ZX Spectrum character set. Its value can be written with two hex digits.

Two bytes can be grouped together to make what is technically called a word. A

word can be written using sixteen bits or four hex digits, and represents a number from 0 to (in decimal)  $256^2 = 65535$ .

A byte is always eight bits, but words vary in length from computer to computer.

The **BIN** notation in Chapter 14 provides a means of writing numbers in binary: '**BIN 0**' represents nought, '**BIN 1**' represents one, '**BIN 10**' represents two, and so on.

You can only use 0's and 1's for this, so the number must be a non negative whole number; for instance you can't write 'BIN -11' for minus three - you must write '-BIN 11' instead. The number must also be no greater than decimal 65535 - i.e. it can't have more than sixteen bits.

**ATTR** really was binary. If you convert the result from **ATTR** into binary, you can write it in eight bits.

The first is 1 for flashing, 0 for steady. The second is 1 for bright, 0 for normal. The next three are the code for the paper colour, written in binary. The last three are the code for the ink colour, written in binary.

The colour codes also use binary: each code written in binary can be written in three bits, the first for green, the second for red and the third for blue.

Black has no light at all, so all the bits are 0 (off). Therefore the code for black is 000 in binary, or nought.

The pure colours, green, red and blue have just one bit 1 (on) out of the three. Their codes are 100, 010 and 001 in binary, or four, two and one.

The other colours are mixtures of these, so their codes in binary have two or more bits 1.

← Appendix D

# **GO SUB stack window**

The **GO SUB** stack window is available from the View menu.

🛚 GOSUB Stack Viewer 🛛 🚺
Stack Size: 392 Items
1020:3
1020:3
1020:3
1020:3
1020:3
Show Source Close

The **GO SUB** stack is a region of memory which keeps track of where your program came from when it executes a **GO SUB** command. When a **RETURN** is subsequently executed, the last location stored on this "stack" is taken off, and program execution is sent to that location. This enables you to run subroutines in your program to take the tedium out of doing repetitive functions, and to organise things into blocks of code. As each **GO SUB** is executed, this stack grows in size. Correspondingly, each **RETURN** executed will shrink the stack.

Each entry takes the form:

*line-number*: *statement-number* 

The topmost entry corresponds to the most recently called subroutine.

Click **Show Source** to jump to the selected line and statement in the **editor**.

# **BEEP Composer window**

The Beep Composer window is available from the Tools menu.

This tool is not yet fully implemented. When finished, it will allow you to compose music visually, using staff notation, and convert it to **BEEP** statements.

# Sound tab

The **Sound** tab can be found in the **Options window**.

Options			
BASin	Editor fonts	Error Reporting	
Emulation	Display	Sound	Files

BASin produces sound from the emulated Spectrum's beeper (generated by DirectSound) and sounds in the editor (generated by the standard waveOut functions in Windows).

**Note:** In general, higher quality sound requires more calculation. If BASin appears to run slowly on your computer, try reducing the sound quality or disabling sound altogether.

Sound settings

You can use the options here to turn on or off the Beeper or Editor sounds, and what type of Editor sounds you want to hear.

## **Beeper/AY**

If this box is checked, BASin plays sounds produced by the **BEEP** command or by manipulation of port \$FE (254) in machine code, and also the AY chip's output from the AY's registers or the **PLAY** command. If it is not checked, beeper and AY sound is muted, but the emulated Spectrum still pauses while it is being played.

#### **Editor sounds**

If this box is checked, the editor generates a click for each keypress, a high beep when a program line is successfully entered, and a low beep when a line is rejected because it contains a syntax error, or already exists.

#### Key click type

Determines whether the keypress click sound resembles that of the 48K Spectrum or that of the 128K Spectrum. 48k sounds will use the softer, sharper "click" of the classic 48k Spectrum, and 128k sounds will use a louder, more beep-ish sound of the 128k. The syntax checking sounds are taken from the 128k machine, and will be used regardless of the key click type.

## **MIDI Device**

BASin can, when emulating the **PLAY** command, drive any MIDI device attached to your PC. Generally, this will be your sound card but you can use this menu to choose any device you have - this includes keyboards, drum machines etc. Choosing "None" will disable MIDI output.

## Volume

This part of the sound options provides a simple slider which allows you to set the volume of the sounds produced by BASin. To the left is the minimum, or silent volume, and to the right of the slider is the maximum. This affects all sound within both the Editor Window and the main emulated sound from your programs.

## Quality settings

Here you can set how accurate and smooth the sound made by BASin is. Higher settings will result in much crisper, cleaner sounds at the expense of greatly increased CPU usage, whereas lower quality will sound much more coarse, but will consume less CPU time.

## Frequency (Hz)

This setting, at either 44100, 22050 or 11250 samples per second (Hertz, or Hz) will set how often BASin "samples" the beeper port to create the sound. Higher settings will consume more CPU time, but will produce much higher quality sound.

## Bit depth

The bit depth can be either 8 or 16 bit, which increases the range of volume granularity that BASin can produce. This setting will use more CPU at 16 bit

than at 8 Bit, but will probably have very little effect on sound quality as the Spectrum's Beeper is only capable of a square-wave type sound, which has no graduated volume setting.

## Stereo

This will allow you to choose either stereo or monaural output. This, like the bit depth, will not have much impact other than increased CPU output if set to stereo, as the Spectrum was not a stereo capable computer. Some Russian clones which incorporate the AY chip of the 128k Spectrum can output stereo sound, but in line with the original Spectrum 128k, BASin's AY output via the PLAY command is mono.

## **DSound Synch**

Enabling this item will tell BASin to synchronise the speed of the emulation with the "play cursor" in the internal Directsound buffers. This can, on good soundcards with good drivers, provide very accurate timings with the least stuttering in the sound. However, not all drivers are created equal, and some can mis-report the position of the "play cursor" or even omit it altogether. If you find that BASin runs very slowly, you can try disabling this option to see if it helps. If it does, then you really need to get a decent soundcard. On-board soundcards are the main culprits in this case.

## Buffering options

BASin does not output sound directly to the soundcard as it is produced by your program. Instead, it builds a "buffer" of sound samples which, when full, is then sent out to the soundcard for playing. Smaller buffers result in quicker response, but the transport to the soundcard can consume a relatively large amount of CPU time. Larger buffers require less processing time, but there may be a small, but noticeable lag before the sound plays.

Users are encouraged to experiment with these settings to find the best for their tastes/tolerances and system hardware.

## Sound buffer size

This is the overall amount of time that will be buffered, in frames per second, of

sound before it is sent to the soundcard. Smaller values require more CPU time, but if this is available then the sound will be much more responsive and remain better synchronised to the display. This will be more noticeable with machine code sound routines, such as sound effects. A larger buffer will require less CPU time to process, but will conversely affect the response of rapid sound effects. This setting will also contribute in some ways to the speed and fluidity of the emulation and display update speed.

#### **Sound latency**

Some sound cards have trouble outputting sound at a rapid rate, and on some computers the CPU may have trouble keeping up with your buffer settings. This can be alleviated by setting a higher latency, but be warned that the higher the latency, the larger the "lag" between emulation and sound synchronisation. Lag experienced due to this option will be more noticeable than lag created by a larger sound buffer.

# LOAD DATA

Statement/Command

LOAD DATA is used to load arrays from files. The arrays are stored using SAVE DATA.

How to use LOAD DATA

LOAD DATA may be used to form a statement or as a direct command. LOAD is first followed by a filename, which is a string value, followed by DATA and a letter or a letter and \$, and finally by a pair of *empty* brackets, for example

# 270 LOAD "numbers" DATA n() 300 LOAD "names" DATA n\$()

The filename following LOAD is the name that is given to the array in the file, and it is subject to the same restrictions as program names used with LOAD. The letter or letter\$ following DATA is the name to be given to the array in the program when it is loaded and used.

On execution, the BASIC searches for the named array. When found, the message Number array: or Character array: followed by the name appears and the array is loaded. Any array currently in memory having the same letter name (n or n\$ above) is deleted, and a new array having this letter name and the values stored in the file is created. Note that with character arrays, any string variable currently in memory having the same letter name is also deleted.

Format

• LOAD string-expr DATA letter[\$]()

See also

Chapter 20.

# LOAD SCREEN\$

#### Statement/Command

LOAD SCREEN\$ enables a screen display to be loaded directly from a file. It sends information to the section of memory controlling the screen display in order to reproduce the picture.

How to use LOAD SCREEN\$

LOAD SCREEN\$ may be used to form a statement or as a direct command. LOAD is followed by a filename, which is a string value, and then SCREEN\$, for example

## LOAD "picture" SCREEN\$

The filename following LOAD is the name that is given to the screen information in the file, and it is subject to the same restrictions as program names used with LOAD. The BASIC then searches for the named information and when found, loads it first into the display file and then the attributes section of the memory. The picture slowly builds up in the current ink and paper colours and then the attributes (true colours and so on) are added.

For more details on storing screen information, see **SAVE SCREEN\$**.

Format

• LOAD string-expr SCREEN\$

See also

Chapter 20.

# SAVE SCREEN\$

#### Statement/Command

SAVE SCREEN\$ stores the screen display in a file. It can be loaded back into the computer at a later date using LOAD SCREEN\$.

How to use SAVE SCREEN\$

SAVE SCREEN\$ may be used as a direct command or to form a statement in a program. SAVE is followed by a filename which is a string value, and then SCREEN\$, for example

## SAVE "picture" SCREEN\$

The filename may have up to ten characters. The display is then sent to the file in the same way as a program is with SAVE.

Format

• **SAVE** string-expre **SCREEN\$** 

See also

Chapter 20.

# **Tape Creator window**

The Tape Creator window is available from the Tools menu.



BASin provides this window as a means to creating and manipulating Spectrum Tape images of your programs and code. These images can be "inserted" into any emulator that supports the .tap and .tzx format, for loading into the emulated Spectrum. Blocks can be created here in this tool, or by using SAVE commands from BASIC whilst Tape Save Streaming is active (see the Options dialog, Files Tab help and the Working with Tape files section for information on setting this up).

The main list in the window contains the blocks that will make up the tape, in the order that they will appear. You should start by either:

- Adding a program block, then adding other blocks for that program to load, or
- Using the File Menu to open an existing tape image.

Adding blocks to a tape image

To add a block, you need to click the "Add" button, or use the "Block" menu item, and choose from the menu that appears:



#### • From File...

Opens the Windows file requester, from where you can load any one of either .bas files, .bsc files, .bsc files, Snapshots (both .Sna and .Z80), .scr files, .tzx/.tap files, and raw binary files which will be added as CODE blocks. When you open a .tzx or .tap file, all the compatible blocks found within will be added to the end of the list.

## • From Current...

This menu has a set of sub-items which allow you to import from your current program:

#### • Program

This option will send the current program area (with variables) to the tape creator as a **PROGRAM** block.

#### • Screen

This option will send the current screen as a .scr file to the tape creator list. This will be a 6912 byte CODE block, and is a straight memory dump of the emulated screen display.

## • Memory Block

This option will open the Add Memory Block window, where you can specify the size and location of the memory block you want to add to the tape as a CODE block.

As you add blocks, they will appear in the main list. You can get the properties of these blocks by using the Block Menu item, or by clicking the "properties" button below the list.
As blocks are added, you can delete them with the "Delete" button, or move them up/down through the list with the up/down arrow buttons. You can cut, copy and paste the blocks by utilising the Edit Menu.

You can use the **Block Menu** item to save these blocks out as their corresponding **File Formats** using the Save As... option.

When you have added all the blocks your program needs, then you can hit the "Build" button to create a tape image. The Windows file requester will appear, prompting you for a filename. The extension you supply dictates the type of file created - a ".tap" filename will create a file of the older "TAP" format, which although less capable than the TZX format, will load into a greater number of emulators. The TZX format is much more flexible, but in BASin has no advantages over the TAP format. You can save files of this type by specifying a ".tzx" extension. The TZX file format is used as a default if you fail to specify an extension, and this format is to be preferred in the interests of phasing out the obsolete TAP format.

The menu strip

## The File Menu

The file menu gives you options for loading and saving the tape images you use.

#### • Open Tape Image...

This will open the standard windows file requester and prompt you to load a new .tzx or .tap file. This will clear any tape image held in memory.

#### • Add From Image...

Like the Open Tape Image... option, this option will prompt you to load a .tzx or .tap file, but will not clear the current tape image - it will add the blocks from the new file to the end of the current tape file.

#### • Save Image

Saves any changes you may have made to the current tape image. If this image has not been saved previously, you will be prompted for a filename.

## • Save Image As...

Allows you to specify the name that the image will be saved under, which can be useful when editing tapes which you want to make copies of.

• Exit

Closes the Tape Creator tool, but leaves the current tape image intact within memory, for tape streaming purposes.

## The Edit Menu

The edit menu gives functionality for copying and pasting blocks, to enable you to manage the list order and content.

## • Cut Block

Removes the currently selected block from the image, but takes a copy and places it on the internal clipboard first.

## • Copy Block

Takes a copy of the currently selected block and places that copy on the internal clipboard.

## • Paste Block

If a copied tape block is currently residing on the internal clipboard, it will be copied and inserted into the tape image at the currently selected tape block position. The block on the clipboard will be left on the clipboard.

## • Delete Block

Removes the currently selected block from the tape image, and does not take any copies of that block before it does so. Handle with care.

## The Block Menu

The block menu gives you methods of working with the blocks themselves, from adding them through to converting them to files external to the tape image. This

can be useful for "ripping" a block out (say for example, a 768 byte font image) for later use in a BASIC program.

#### • Add From File

Opens the Windows file requester, from where you can load any one of either .bas files, .bsc files, .bsd files, Snapshots (both .sna and .z80), .scr files, .tzx/.tap files, and raw binary files which will be added as CODE blocks. When you open a .tzx or .tap file, all the compatible blocks found within will be added to the end of the list.

#### • Add From Current...

This menu has a set of sub-items which allow you to import from your current program:

#### • **Program**

This option will send the current program area (with variables) to the tape creator as a **PROGRAM** block.

#### • Screen

This option will send the current screen as a .scr file to the tape creator list. This will be 6912 bytes long, and is a straight memory dump of the emulated screen display.

#### • Memory Block

This option will open the Add Memory Block window, where you can specify the size and location of the memory block you want to add to the tape as a CODE block.

#### • Save As...

Allows you to save the currently selected block as one of the BASin supported types- for instance, you can save Program blocks as .bas files, Code blocks as .bsc files, and variables as .bsd files. Certain types can also be saved as raw bytes. You will be prompted for a filename to save the

block under, and in these circumstances your choice of file extensions will determine what you can save the blocks as.

In particular it is worth mentioning that although every effort to make BASin as stable as possible has been undertaken for these procedures, a lot of the more modern games played fast and loose with the rules of the ROM where memory structure was concerned, and inserted all sorts of fiendish tricks like invalid lines and misleading system variables into their programs. If BASin encounters such a program block, it may well hang or crash BASin when it tries to extract the BASIC from them. You have been warned!

## • Move Up

Moves the selected memory block up one place in the list.

## • Move Down

Moves the selected memory block down one place in the list.

## • Properties

Opens the Block properties dialog where you can set the parameters of each block that appears in your tape image. The options present here are explained in more detail elsewhere in the Tape Block Properties Window help section.

## The Streaming Menu

One of BASin's other tape manipulation features is the ability to stream data to your program from tape image files. These options enable or disable this behaviour.

## • LOAD/MERGE/VERIFY Commands

When checked, this option instructs BASin that all LOAD, MERGE and VERIFY commands executed by the user are to gather their data not from files on your hard disk, but from the current tape image held in the Tape Creator Window.

#### • SAVE Commands

When checked, this option instructs BASin that all SAVE commands executed by the user are to send their data not to files on your hard disk, but to the end of the current tape image held in the Tape Creator Window.

## **Memory Viewer window**

The Memory Viewer window is available from the View menu and the System Variables window.



**Note:** Editing the memory incorrectly can corrupt your BASIC program or cause the emulated Spectrum to crash.

The memory viewer is reminiscent of a file hex-editor, but operates on the memory of the emulated spectrum. It is primarily intended for close inspection/editing of various regions of memory, and to that end it supports instant location of those areas.

At the top of the window, you will notice that there is a drop-down list attached to an edit field. From the drop-down list, you can choose from any one of the System Variables, and the display area will move to the address pointed to by that SysVar. For example, you could choose the **PROG** system variable, and the display will show the start of the area of memory that contains your program. You can also choose from a list of any variables that are stored in the VARS area of memory, and inspect the structure of those. It may help to refer to Chapter 24 of the Sinclair BASIC Manual, which lists how variables and programs are stored in memory in much more detail. An interesting feature that you may use often is the program line locator - simply type in a line number followed by the statement number (in *Line:Statement* format, i.e. "**10:1**") and the display will move to that line's tokenised representation in memory.

Having chosen a region to jump to, you must click the "Go" button to the right

of the edit field to actually move to that location.

You can also use the "Find" button to search memory for whatever string of characters you have typed into the edit field.

You can edit the memory by either typing in the hexadecimal codes into the lefthand pane of the window, or by typing in regular ASCII codes into the righthand pane. All changed memory locations are coloured in **dark red**.

At the bottom of the window is a small status area, which shows the current address that you are examining, and the contents of that address in both byte and word format.

# **Emulation tab**

The **Emulation** tab can be found in the Options window.

Options			
BASin	Editor fonts	Error	Reporting
Emulation	Display	Sound	Files

Z80 core emulation

## Allow IN for keyboard

If this box is checked, the **IN** keyword can be used to read the emulated Spectrum keyboard at hardware level - for example, to detect multiple keys being pressed at once.

## **Speed tuning**

Determine the speed of the emulated Z80 processor (approximately 3.5 MHz on a true Spectrum, or 69888 TStates per frame). Beware that excessibley speeding BASin up can cause the display to stop responding, and for the sound to break up if your PC is not fast enough.

ZX printer emulation

BASin emulates a ZX Printer as a device which operates independently of the main Z80 emulation. The Z80 core can communicate with the device through port commands, to set the stylus and to turn the motor on and off.

#### Save output between sessions

To simulate a real printer, BASin can save the state of the "paper" that the ZX Printer uses, as a continuous sheet, or roll of paper. This option will save the current printer output when you quit BASin, and re-load it when you start a new BASin session. The file is saved in BASin's folder, as "Printer.bin".

## **Accelerate printing**

All ROM Commands for printing such as LPRINT, LLIST, COPY and the printer streams pass through a particular point in the ROM which sends a complete line of 32 characters to the printer. BASin can trap this and send all the characters in one go, which considerably accelerates printing. This option will turn this behaviour on and off. Machine code routines which utilise this ROM entry point will also benefit from this behaviour, but it is not guaranteed to work with all machine code routines, especially if they don't supply the right values.

# **Display tab**

The **Display** tab can be found in the **Options window**.

Options			
BASin	Editor fonts	Error	Reporting
Emulation	Display	Sound	Files

Scaling options

Here you can configure how BASin scales the Display Window when the user resizes. The options affect display update performance in various ways, so experiment to find the settings that best suit your display and PC hardware.

#### Maintain aspect ratio

If this box is checked, the width and height of the display window always keep their standard PAL ratio of 4:3, and the border expands to fill any remaining space. To set this option temporarily, choose **Display Window > Force 1:1 Aspect** on the View menu.

#### Use integer scaling

This, when enabled, will only allow scaling to occur at exactly 1x, 2x, 3x etc of the display. Any scaling in-between will not occur, for example at 150% (1.5x), and borders will be added to make up the extra space in the window. This option is enabled when using some scalers - the Hq2x, Super 2xSAI, Super Eagle, and Scale2x types, as they can in most cases only scale at 2 or 3 times normal size. These types make educated guesses as to what the scaling should look like, so some distortion can occur in dithered graphics.

## Use 8-bit stretch for GDI mode

This option will, particularly on NT based systems, allocate an extra in-memory bitmap set to 256 colours which will be used for scaling. This can increase scaling performance considerably when using the GDI scaling method, but does

require more memory to use.

## Scaling method

This option dictates how BASin will scale the Display Window when it is enlarged beyond 100% of the original (320x240) size. BASin does not employ DirectDraw or OpenGL acceleration. The options are:

Scaling method	Available sizes	Notes
None (GDI)	Any	Uses the standard Windows scaling routines. Relatively slow, but guaranteed to work on any system. To improve performance, also select <b>Use 8-bit</b> <b>stretch in GDI mode</b> .
Bilinear	Any	Uses a software bilinear interpolation method to scale the display smoothly. This may be very slow at larger sizes, and causes pixels to appear blurred where blending occurs between them.
Scale2x	2×, 3×, 4×	Uses a fast and simple algorithm to guess the intended shape of a set of pixels. This is restricted to 256 colours, so no blending occurs.
HQ2x	2×, 3×, 4×	Combines a method similar to Scale2x and some blending to achieve the highest quality display. Relatively slow, but optimised for Intel processors that support MMX (Multimedia Extensions).

Super 2xSAI	2× only	Uses a relatively slow and complex algorithm to guess the intended shape of a set of pixels. The result is of higher quality than Scale2x.
Super Eagle	2× only	Similar to Super 2xSAI.

Frameskip

Frameskipping is a useful system which, rather than render all 50 frames of display every second, instead renders only a set portion - every other frame, every third frame and so on. This has the advantage that BASin is able to emulate the Spectrum more fluidly, as it has to spend less time per second rendering the display (converting the Spectrum's Display memory area into a Windows bitmap image is quite an intensive operation). Frameskip can be quite useful, mainly because Sinclair BASIC is not the swiftest language, and does not require a constant frame update every 50th of a second.

## Update every *n* frames

Determines how often BASin updates the display. Setting this to 1 always updates the display immediately; setting it to 2 only renders every other frame; and so on. A larger value increases performance but reduces the smoothness of the display.

## Automatically skip if necessary

Selecting this option will set BASin to update only when the emulation is waiting for synchronisation to the 50hz frequency - i.e. when the engine is idling. This will allow a smooth emulation with, depending on the speed of your PC hardware, a reasonable display update speed. This is managed by sending all display functions to a separate thread, which waits until Windows gives it a slice of CPU time to work with. As the new thread is set to a lower priority than the emulation, then only when the emulation is idling does the display get priority.

Note: On slower hardware with large scaling applied, this option can freeze the

display or cause a rather jerky update.

## Embellishments

These extra options are cosmetic only, and will not affect the speed or the quality of the display. They can be enabled or disabled according to the tastes of the user.

## Scanlines simulation

If this box is checked, BASin adds dark lines to the display that simulate the scan lines on a portable TV set. (The original Spectrum was generally used with a TV set, not a computer monitor.)

## **Rounded display corners**

If this box is checked, the display is given a rounded rectangle shape, which more closely resembles an old 14" television set.

## Working with Tape image files

**Note:** Detailed information about the two types of tape file format that BASin understands can be found here.

There are two methods which BASin provides for managing Sinclair Spectrum tape image files, which are predominantly used by emulators. These files are intended to be a compressed representation of the sounds which the Spectrum produces when saving files, and to this end can be very complex to use. BASin, through the use of modified emulation, employs these tape images as storage areas where you can store files created from BASIC programs and memory data blocks.

The simplest method of creating and working with tape files is to use the Tape Creator tool, which is found in BASin's Tools Menu. You should read the documentation for this tool if you are at all unsure as to how to use tape image files. It allows you to add, remove and create new tape blocks from programs either in-memory or from .bas files, and to add blocks from memory chunks, and to save certain types of variables.

The other method for working with tape files within BASin is to use "streaming" where the actions of LOAD, SAVE, MERGE and VERIFY commands are redirected to a tape image rather than files on your hard drive. The Files Tab in the Options dialog has two switches, which allow you to turn this behaviour on or off. When active, for example, a LOAD "" command will (instead of bringing up a file requester asking for a file to load) attempt to load a block from the current tape image. If no tape image has been created and "attached", then the file requester will be used. If tape SAVE streaming is active, then any SAVE command issued by the user will send the saved data directly to the tape, adding it on to the end of the tape image. This behaviour is very close to that of Spectrum Emulators in general.

The tape image formats that BASin supports are the .tzx and the .tap formats. The .tap format is supported completely - but only the .tzx format's ROM block type (ID type 16) is supported by BASin. This is because BASin intercepts ROM routines for loading and saving, rather than monitoring the EAR bit of port \$FE (which is where tape-sounds would normally be arriving in a real Spectrum). As such, complex loaders that employ the more exotic blocks of the .tzx format will fail to be recognised, or indeed even acknowledged as existing at all. Support for headerless tape blocks is provided, however.

The original tapes stored files sequentially, and the position of the tape head in the tape drive, along the tape, dictated which file would be loaded next. In BASin, this is indicated in the Tape Creator tool by the small cassette-tape icon that appears to the left of the block list. This icon indicates which block will be loaded next by a streamed tape operation.

## **Tape Block Properties window**

The Tape Block Properties window is available from the Tape Creator tool.

This window allows you to view the properties of any tape blocks found in the Tape Creator window. Some block types can be modified - the information contained in the tape header is customisable.

For more information about tape file headers, see the Tape image file formats section.

Tape block types explained

#### **Program Blocks**

🛚 Tape Block Properties 🛛 🛛 🔀	
Program: Block	
Program Name	
Untitled	
Auto-start	
	C Enabled
	Line number
	이
	Okay Cancel

Program blocks are listed with the prefix "Program:" and contain images of the program as it exists in the PROG memory area, with any variables that have been declared. Options available to set are the filename that appears as the block loads, and the Autostart line number - when the program is loaded, then that line will be executed automatically as though a GO TO command had been issued.

#### **CODE Blocks**

🗷 Tape Block Properties 🛛 🛛 🚺		
Bytes: Block		
Block Name		
Unnamed		
Parameters		
Start Address	16384	
Length (Read Only)	6912	
Okay	Cancel	

CODE blocks are listed with the prefix "Bytes:" and are an image of an area of memory. They must typically be loaded using the LOAD "" CODE direct command, and the only optional parameter you can set (aside from the filename) is the start address that the block will be loaded into.

## **Array Blocks**

🗷 Tape Block Properties 🛛 🛛 🔀	
Number array:	Block
Block Name	
a	
	Okay Cancel

Array Blocks are listed with the prefix "Number Array:" or "String Array:" and hold the contents of either a string, a string array, or a numeric array. You can set the filename for these blocks, but there are no other options that you can set.

**Note:** The original ROM file loader could only accept up to ten characters for a filename, and so the tape images that BASin produces inherit this limitation.

# **Add Memory Block window**

The Add Memory Block window is accessible from the Tape Creator tool.

🛚 Add Memory Block 🛛 🛛		
StartAddress:	32768	
Length:	768	
	Okay Cancel	

This window is used to specify a memory block to add to the Tape Creator Window. Here you can specify the location and size (using the two edit fields) of the memory block you wish to copy onto your tape image.

#### Start Address

This is the location in the emulated memory model that you wish to copy your block from. This value can range from 0 to 65535.

#### Length

This value specifies the length of the memory block you want to copy. Bear in mind that there is a maximum size of 65535 bytes, but only enough bytes to copy from the **Start Address** up to this limit will be copied.