bright basic computer

user manual & programmer's guide

http://brightcorporation.net/basic-computer

ABOUT THE BBC

The Bright Basic Computer (or "BBC") is a working microcomputer, which can be programmed using the BASIC programming language.

It is similar to the 8-bit computers of the late 70s and early 80s, such as the Apple I, the Commodore PET, and the Sinclair ZX81.

It runs slowly, interpreting Basic at around 5-10 lines per second, and (like the machines above) it runs in text mode, without drawing or pixel mapping: so it does not have the speed or graphics for real-time or arcade-style games.

However, there exists a huge library of non-real-time, text mode programs, originally written for early microcomputers in their various versions of BASIC, which can be adapted to "Bright Basic" and run on the BBC. We have included some, including the much loved text adventure "The Oregon Trail" played in many US schools, games from various 1970s issues of "Creative Computing" magazine, and a dozen new games we wrote ourselves for the BBC.

Bright Basic was designed to support the most common features of the various BASIC dialects popular at the time, and should feel immediately familiar to anyone who has used them.

The Bright Basic Computer comes with a 40 column x 25 row, cursor-addressable colour monitor with a 4:3 aspect ratio, typical to those of the period.

Both the computer and the monitor run entirely in-world. The monitor displays output using real, Second Life textured prims, and so does not require users to activate web-on-a-prim style "shared media" streams to view. And BASIC programs are executed by scripts inside the computer, with no dependency on off-world emulators.

You can write programs by clicking the keyboard, entering commands in chat, or writing programs on notecards and then dropping them into the computer. You are naturally also free to share these notecards with anyone else who owns a BBC.

And the system features a floppy disk drive, and is supplied with transferable, copyable floppy disks. You can save programs from your BBC onto a floppy: to keep a backup for yourself, or to share with other BBC users.

The system has highly configurable security control, so you can - if you wish - protect your computer against use by others, leave it rezzed in a public space for anyone to use, or restrict it to a "kiosk like" mode in which visitors may run the programs you install, but not break out of them or reprogram the machine.

The Bright Basic Computer is a loving recreation of the early days of popular computing. Use it to write and run your own programs, or to provide realistic, detailed, and interactive scene dressing for a public venue.

```
10 PRINT "HAVE FUN!"
20 GOTO 10
```

PS. The name "Bright Basic Computer" was chosen both because it was descriptive - it is a computer which runs Basic - and as a fond tribute to a British computer called the "Acorn BBC Micro", released in 1981 by Acorn Computers Limited. Acorn's machine was, in turn, named after the British Broadcasting Corporation, who commissioned it for use in their "BBC Computer Literacy" project, and the Acorn BBC was adopted in many schools to teach computing, making it culturally iconic in Britain. However, the Bright Basic Computer is not an exact emulation of the Acorn BBC micro, or of any single machine. It draws on many popular mainly US and UK microcomputers of the period, from Commodore, Sinclair, Apple, Acorn, and others.

CONTENTS

ABOUT THE BBC....1
1. GETTING STARTED....6
1.1 UNPACKING....6
1.2 USING THE CONTROLS....6
1.3 ENTERING COMMANDS....7
1.3.1 METHOD 1: CLICKING THE KEYBOARD....8
1.3.2 METHOD 2: USING THE "CHAT INPUT CHANNEL"....8
1.3.3 METHOD 3: USING "CHAT-TO-TYPE"....9
1.3.4 COPY AND PASTE....9
1.3.5 MULTICOMMAND LINES....10
1.4 WRITING PROGRAMS....10
1.5 USING PROGRAM NOTECARDS....12
1.6 INCLUDED NOTECARDS....17
1.7 USING FLOPPIES....19
2. SETTINGS....22
2.1 COMPUTER SETTINGS....22
2.2 MONITOR SETTINGS....24
3. BASIC....26
3.1 ABOUT BRIGHT BASIC....26
3.1.1 COMPARISON WITH EARLY BASICS....27
3.1.2 BRIGHT BASIC MODERNISATIONS....31
3.1.3 MEMORY HANDLING....33
3.2 DATA HANDLING....35
3.2.1 CHARACTER SET....35
3.2.2 LITERALS....35
3.2.3 LOGICAL VALUES....36
3.2.4 VARIABLES....37
3.2.5 ARRAYS....38
3.3 OPERATORS....40
3.3.1 PARENTHESES ()....41
3.3.2 EXPONENTIATION (^)....41
3.3.3 MULTIPLICATION AND DIVISION (*, /, \)....41
3.3.4 ADDITION AND SUBTRACTION (+, -)....41
3.3.5 CONCATENATION (~)....42
3.3.6 COMPARISON (>=, <=, >, <)....42

3.3.7 EQUALITY (=, <>, !=)....43
3.3.8 LOGICAL AND, OR, AND NOT (&, |, !)....43
3.3.9 ARRAY BUILDING (`)....44
3.3.10 ASSIGNMENT = += -= *= /= #= ^= &= |= `=....44
3.3.11 SINGLE CHARACTER ALTERNATIVES (#,], [)....45
3.4 FUNCTIONS....45
3.4.1 ABS(number)....46
3.4.2 ALEN(array)....46
3.4.3 AND(value1, value2)....46
3.4.4 ASC(string)....47
3.4.5 AT(row, column) / AT(column)....47
3.4.6 CASE(text, upcase)....47
3.4.7 CHR(asciicode)....48
3.4.8 COLOUR(number) / COLOR(number)....48
3.4.9 DEL(string, start, length)....49
3.4.10 EVAL(expression)....49
3.4.11 FREE(0)....50
3.4.12 IIF(condition, trueresult, falseresult)....50
3.4.13 INKEY(pressednow)....50
3.4.14 INS(string, start, substring)....51
3.4.15 INSTR(string, substring)....52
3.4.16 INT(number)....52
3.4.17 JUMP(columns)....52
3.4.18 LEFT(string, length)....53
3.4.19 LEN(string)....53
3.4.20 MID(string, start, length)....53
3.4.21 MOD(number, divisor)....53
3.4.22 NOT(value)....54
3.4.23 OR(value1, value2)....54
3.4.24 PAD(string, length, char, left, truncate)....54
3.4.25 REP(string, start, substring, length)....55
3.4.26 RIGHT(string, length)....55
3.4.27 REPEAT(string, count)....55
3.4.28 RND(0)....56
3.4.29 SPC(number)....56
3.4.40 TAB(column)....56
3.4.41 TIME(0)....57

3.4.42 TRIM/LTRIM/RTRIM(string).....57
3.5 COMMANDS.....57
3.5.1 BEEP tone.....58
3.5.2 CLEAR.....58
3.5.3 CLS.....58
3.5.4 DATA value, value... / READ variable / RESTORE.....58
3.5.5 DELETE linenumber / from, to.....59
3.5.6 DIM array(elementcount) = value.....60
3.5.7 END / STOP.....61
3.5.8 FOR variable = startvalue TO finalvalue STEP increment / EXIT condition / NEXT.....61
3.5.9 GET variable.....63
3.5.10 GOTO linenum.....64
3.5.11 GOSUB linenum / RETURN.....65
3.5.12 IF condition THEN / ELSEIF / ELSE / ENDIF.....65
3.5.13 INPUT prompt, variable[;] AS pattern.....66
3.5.14 LABEL/@ labelname.....69
3.5.15 LET variable = value.....69
3.5.16 LIST/LLIST linenumber / from, to.....70
3.5.17 LISTV from, to.....71
3.5.18 LOAD notecard.....72
3.5.19 LOOP / WHILE/UNTIL condition / REPEAT.....74
3.5.20 NEW.....75
3.5.21 ON number GOTO/GOSUB linenumber1, linenumber2.....75
3.5.22 PLAY/PLAYLOOP sounditem/uuid, volume / PLAYSTOP..76
3.5.23 PRINT/?/LPRINT textvalues.....77
3.5.24 REM/' comment.....79
3.5.25 RUN linenumber.....80
3.5.26 TRON/TROFF.....81
3.5.27 WAIT seconds.....82
3.6 APPENDICES.....82
3.6.1 SYMBOLS.....82
3.6.2 PRINT TOKENS.....83
3.6.3 INPUT PATTERNS.....84
3.6.4 KEY NAMES (INKEY/GET).....85

1. GETTING STARTED

This section explains how to rez and use your computer.

1.1 UNPACKING

Find the object "Bright Basic Computer boxed" in your inventory, and drag it onto the ground to rez it. Right-click the box for its context menu, select "Open" to display its contents window, and click the "Copy To Inventory" button at the bottom.

This will create a folder in your inventory, also called "Bright Basic Computer boxed". Inside this folder, you will find the object "Bright Basic Computer". Drag this to the ground to rez it. You will see the computer, and its monitor.

Note that the computer and the monitor are separate objects. You can keep them together, or place them up to 96m apart. They are designed so that if you use the Second Life object editor to move both objects to exactly the same X/Y/Z coordinates, the monitor will sit on top of the computer.

IMPORTANT: If you rez more than one computer, each will display its output on the monitor closest to it. If you replace a computer's monitor, switch the computer off and on again to force it to use the new monitor. Make sure that no monitor is the closest monitor to two computers, or they will both update it, and the result on the screen is likely to be confusing.

Also, please do not change the object names "Bright Basic Computer" or "Bright Computer Monitor". In order to communicate, the computer and monitor scan for each other by name. If either discovers that its name has been changed, it will automatically change it back - but a name change may still briefly interrupt their connection until it is autocorrected.

1.2 USING THE CONTROLS

The computer has a red “chat-to-type” button and red power switch on the front, and a keyboard with black and red keys.

Click the "chat-to-type" button to switch chat-to-type mode on and off. [SEE 1.3 ENTERING COMMANDS]

Click the power switch to switch the computer on or off.

Click the black keys to type text into the computer.

Click the red "help" key for brief advice on using the computer.

Click the red "load" key to see a menu of the programs stored inside the computer on notecards. Choose one of these programs, and the computer will load it into memory, and run it. (Below, you will learn how to write your own programs onto notecards, and add them to this menu. [SEE 1.5 USING PROGRAM NOTECARDS])

Click the red "settings" key to configure the computer. [SEE 2.1 COMPUTER SETTINGS]

Click the red "break" key while a program is running to stop it, and return to the ">" command prompt.

Click the floppy disk on the disk drive to the right of the computer to receive a blank floppy. [SEE 1.6 USING FLOPPIES]

The monitor has a single settings button (showing crossed tools) below its screen. Click this to access the monitor's settings. [SEE 2.2 MONITOR SETTINGS]

With factory settings, only you - as the computer's owner - may use any of these controls. But you can configure access, allowing other people to use none, some, or all of them, and optionally making access conditional on group membership. [SEE 2.1 COMPUTER SETTINGS]

1.3 ENTERING COMMANDS

When your computer is ready to receive a command, the screen will display a ">" symbol as a command prompt.

```
>
```

When you see this prompt, you can enter commands in the Basic programming language. This section uses some simple examples of Basic commands, but do not worry if you are not familiar with the language - it is described in great detail later in this manual.

Commands may be up to 256 characters long.

There are three ways to enter them.

1.3.1 METHOD 1: CLICKING THE KEYBOARD

Simply click the keys on the keyboard, and text will appear on the monitor.

Click ERASE to erase mistakes. Click a SHIFT key to make your next click select a lower case letter, or a shifted symbol on a key. Click SHIFT LOCK to make these shifted symbols the default - an indicator light on the key will glow to indicate SHIFT LOCK is enabled until you click it it again.

Finally, click ENTER to complete your command.

Clicking works, but it can be a laborious way to enter a lot of text. The other two entry methods allow you to use chat, which can be quicker.

1.3.2 METHOD 2: USING THE "CHAT INPUT CHANNEL"

You can type commands using the "chat input channel", which is shown on an LED display on the front of the computer. To begin with, the chat input channel is 9, so you might type the following into your chat window:

```
/9 PRINT "Hello World!"
```

The monitor will echo your command after the ">" prompt, and then execute it, displaying:

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

1.3.3 METHOD 3: USING "CHAT-TO-TYPE"

If you want to save yourself having to type /9 before every command, click the "chat-to-type" button to enable chat-to-type mode. Your computer will now listen to anything you say in ordinary chat - which can make it easier to use if you have a lot of text to enter.

Of course, this means that the computer will also listen to anything you say in chat to other avatars nearby, which can be inconvenient. Clicking the "chat-to-type" button again will disable chat-to-type mode - but still allow you to enter commands by clicking the keyboard, or using the /9 prefix. You can switch chat-to-type mode on and off as is convenient.

1.3.4 COPY AND PASTE

You can use Control+C to copy text from another source - say, a Basic program listing on the web - and then use Control+V to paste it into chat, using either the chat input channel prefix, or chat-to-type.

If you copy and paste multiline block of text, containing a series of commands, each one will be executed in turn. To try this, select and copy the following block of lines with one Control+C, and paste them into your chat window to enter them into your computer.

```
PRINT 4*8
PRINT 132/12
PRINT 3+7
```

The monitor will show:

```
>PRINT 4*8
32
>PRINT 132/12
11
>PRINT 3+7
10
>
```

This is a handy way of entering a few commands, and can be useful in trying out the short code examples in this manual. But to enter long

programs copied from other sources, it is better to paste them into a notecard. [SEE 1.5 USING PROGRAM NOTECARDS]

1.3.5 MULTICOMMAND LINES

You can enter multiple commands on a single line, separated by colons (:).

For instance, this line contains four commands. (1) Assign a value to the variable A, (2) print A, (3) double A, and (4) print A's new value.

```
A = 100 : PRINT A : A = A * 2 : PRINT A
```

Enter this line on your computer, and the monitor will display:

```
>A = 100 : PRINT A : A = A * 2 : PRINT A
100
200
>
```

1.4 WRITING PROGRAMS

So far, every command you have typed in has been executed immediately. In Basic, they are known as "direct mode" or "immediate mode" commands.

But if you enter a command prefixed by a number - called a "line number - the command is not executed immediately. Instead, it is saved in your computer's memory as part of a program.

Try entering the following commands.

```
10 PRINT "This is a program."
20 PRINT "It has numbered commands."
30 PRINT "They will run in sequence."
```

Note that none of these commands were carried out when they were entered: they've been stored as a program. To see the program, use the LIST command.

```
>LIST
10 PRINT "This is a program."
20 PRINT "It has numbered commands."
```

```
30 PRINT "They will run in sequence."
3 line(s) listed.
>
```

To run the program, enter the command RUN.

```
>RUN
This is a program.
It has numbered commands.
They will run in sequence.
>
```

Program lines are sequenced not by the order in which they were entered, but always by their line numbers.

You may use any line number between 1 and 99999. Note that the line numbers in this example run 10, 20, 30, rather than 1, 2, 3. Most programs for Basic-powered microcomputers were written this way, because leaving gaps in the line number sequence allows new lines to be added to a program, between two existing lines.

```
>15 PRINT "(This line was inserted.)"
>LIST
10 PRINT "This is a program."
15 PRINT "(This line was inserted.)"
20 PRINT "It has numbered commands."
30 PRINT "They will run in sequence."
4 line(s) listed.
>
```

If you enter a line with the same line number as an existing line, the old program line is simply replaced.

```
>10 PRINT "This is a BASIC program."
>LIST
10 PRINT "This is a BASIC program."
15 PRINT "(This line was inserted.)"
20 PRINT "It has numbered commands."
30 PRINT "They will run in sequence."
4 line(s) listed.
>
```

And entering a line number on its own - without a command - deletes the corresponding line from the program.

```
>30
>LIST
10 PRINT "This is a BASIC program."
15 PRINT "(This line was inserted.)"
20 PRINT "It has numbered commands."
3 line(s) listed.
>
```

You can clear a program from the computer's memory using the NEW command.

```
>NEW
>LIST
0 line(s) listed.
>
```

Bear in mind that it is possible to write a program which will, in theory, run forever...

```
>10 PRINT "Hello world!"
>20 GOTO 10
>RUN
Hello world!
Hello world!
Hello world!
...
```

...but you can click the "break" key on the BBC's keyboard to stop a program while it is running.

```
...
Hello world!
Hello world!
*BREAK*
>
```

1.5 USING PROGRAM NOTECARDS

As an alternative to writing programs by typing numbered commands directly into the computer, you can write them into a notecard, drop this

into your computer's inventory, and then load the programs they contain into the computer's memory.

Not only does this allow you to use an ordinary text editor to write your Basic, it allows you to keep a copy of your program to load in the future - or to give to someone else to run on their own BBC.

Begin by creating a notecard in your inventory. Give it a simple name, with up to 12 characters - like (say) "TESTPROG". Then enter your program.

```
10 REM Simple demo program
20 PRINT "This is a short program."
30 PRINT "It was written in a notecard."
```

Right-click your computer for its context menu, select "Open" to display its "Contents" window, and drag the "TESTPROG" notecard from your inventory into the window. A copy of the notecard is now stored in your computer's inventory.

Use the "LOAD" command to load your program from the notecard into your computer's memory.

```
>LOAD "TESTPROG"
Loading "TESTPROG", 3 lines...
Autorunning program...
This is a short program.
It was written in a notecard.
>
```

Now that the notecard is in the computer, you can modify the program using the notecard editor, and LOAD it to test your changes as you go.

This is the easiest way to develop programs.

It works like this. First, right-click your computer for its context menu, select "Open" to display its "Contents" window, double-click the "TESTPROG" notecard to open it, and finally close the "Contents" window.

Now the "TESTPROG" notecard inside your computer is visible in an editing window on your screen. You can modify it, and periodically click the "Save" button at the bottom of the editor to save your changes - all

while keeping the edit window open. And you can reload your modified program to test on your computer at any time using the command LOAD "TESTPROG".

If you are an LSL programmer, you will find this process very similar to editing and testing LSL scripts in Second Life. The only difference is that instead of using an LSL script editor, you are using a notecard editor.

Note that when the LOAD command has loaded a program, it will start running it automatically. You can click the "break" button to stop it, but if your program is half-finished, and you are still working on it, you might not want it to run automatically on each load.

To indicate that a program is incomplete, just include a question mark in the notecard's name - like "TESTPROG?". You can still LOAD it, and still manually use the RUN command to start it when loaded, but it will not start running automatically.

Also, notecards with "?" in their names will not appear on the menu of programs listed when someone clicks the "load" key - so if you allow other people to click this button to load and run programs, you needn't worry that they'll accidentally choose to load a half-finished program.

There are three special "tokens" you may add to a notecard: {NOTE}, {!}, and {AUTONUM}.

Any line in the notecard which begins with {NOTE} or {!} is simply ignored. This allows you to add comments to the code stored in the notecard, without having them loaded as part of the Basic program. Basic REM statements also allow you to write comments, but though they do nothing, they are still Basic statements, so they take up memory in the computer, and when a program is executed, it takes a small but still finite amount of time to read and fall through REM statements. If you spend most of your time editing programs in the notecard, rather than on the computer, you may prefer to use {NOTE} or {!} comments. [SEE 3.5.24 REM]

You can also put an {AUTONUM} command on the first line of your notecard.

{AUTONUM} allows you to write a Basic program without using line numbers. As the lines are loaded, they will automatically be assigned line numbers at intervals of 10: so the first basic line will become line 10, the second 20, and so on.

In traditional Basic, lines had to be numbered to provide a destination for commands like GOTO. But in Bright Basic, you can use the LABEL command (or its abbreviation @) to act as a destination for a GOTO, making line numbers optional. [SEE 3.5.14 LABEL]

Remember also that, as explained above [SEE 1.3 ENTERING COMMANDS], the computer will automatically upcase Basic commands and keywords, so you can also write them into your notecard in lower case if you prefer.

You can also include blank lines, which LOAD will ignore.

So if you chose to use the {autonum}, automatic casing, blank line support, and labels features, the program in your notecard might look like this:

```
{autonum}

@start
print "The first five square numbers are..."
counter = 1
loop
    print counter^2
    counter += 1
while counter <= 5
repeat

input "Again? (Y/N): ", again
if again = "Y" then
    print "Ok, here they are again."
    goto start
endif
```

At first glance, this might look very different from the examples of BASIC above. The command and variable names are in lower case, there are no line numbers, and there is even indentation to show the lines within the

LOOP/REPEAT and IF/ENDIF blocks. It looks like a more modern implementation of Basic.

But all these features are optional. Keywords and variable names are automatically upcased on entry. Line numbers are automatically added because {AUTONUM} has been specified at the top of the program. And the indentation will be ignored by the computer when the code is loaded.

Bear in mind that this program will look a little different once it has been loaded. LIST will show:

```
10 @START
20 PRINT "The first five square numbers are"
30 COUNTER = 1
40 LOOP
50 PRINT COUNTER^2
60 COUNTER += 1
70 WHILE COUNTER <= 5
80 REPEAT
90 INPUT "Again? (Y/N): ", AGAIN
100 IF AGAIN = "Y" THEN
110 PRINT "Ok, here they are again."
120 GOTO START
130 ENDIF
```

Bright Basic is capable of interpreting old programs written for early Basic computers, with their line numbers and all-caps, or it can load more modern looking Basic from a notecard like the example above. Which you ask it to do is entirely up to you.

One sometimes useful technique is to use the LLIST command - a version of the LIST command [SEE 3.5.16 LIST] which sends output to local chat, rather than your computer monitor - and then cut and paste text from your chat window (or saved "chat.txt" Second Life log life) into a notecard. Chat is noisy, and you may wish to delete any lines beginning [hh:mm]... to tidy up the code, but in case any are accidentally left in, the LOAD command [SEE 3.5.18 LOAD] will ignore any notecard lines beginning with an opening bracket, "[".

1.6 INCLUDED NOTECARDS

Your computer was delivered with 22 notecards in its inventory, each storing a Basic program.

Click "load" to load and run programs from these notecards, and see what they do. Feel free to edit the notecards themselves, see how they work, modify them, and delete any you do not need.

The first program is called "!WELCOME!". This displays introductory information about the BBC. It is designed to help people who may be seeing the computer for the first time, explaining what it is, and how to use it and load programs.

The next 13 program notecards were all written specifically for the BBC:

BLACKJACK - Casino style Blackjack game
BOXART - Ambient music & coloured box display
HANGMAN - Hangman word-guessing game
HANOI - "Towers of Hanoi" puzzle game
HORSES - Horse race betting game
MASTMIND - Mastermind positional puzzle game
MATRIX - Screen display inspired by "The Matrix"
QUIZ-GEO - Geography quiz game
QUIZ-PRES - US Presidents quiz game
QUIZ-SL - Second Life quiz game
ROCKPAPER - Rock Paper Scissors game
ROULETTE - Casino roulette game
XCALC - Expression calculator

The program notecard "{OREGON}" contains the source code of the original version game "The Oregon Trail" - a text adventure in which you play as a 19th century pioneer settler, trekking from Independence, Missouri to Oregon City, Oregon.

The game was written in 1971 by Don Rawitsch, Bill Heinemann, and Paul Dillenberger, and produced and distributed by MECC (Minnesota Educational Computing Consortium) to US schools. It became a cultural icon, with new versions being developed for new computers as they were

released over decades. The game sold over 65 million copies, and was inducted to the World Video Game Hall Of Fame.

An introduction has been added to the game, and minor edits have been made to suit Bright Basic, but otherwise the source code is as it was written in 1971 - a milestone in the early history of popular computing.

The last 7 notecards contain Basic programs originally published in the magazine "Creative Computing" in the 1970s. This was one of many "listings magazines", which printed Basic source code listings for hobbyists to type into their microcomputers, and many thousands of such programs have been archived on retro-computing websites.

They were originally published for various machines, running various dialects of Basic, and so will all require some editing to work with Bright Basic, and some use machine specific features which make them difficult to convert, but many text mode games, educational, and even scientific programs can be converted easily to run on the BBC.

We looked at many, and chose these 7 as a representative sample, varying in length and quality. The archives from which they came are not only a source of some good programs, but offer a glimpse into the technology, and even the social attitudes, of the 1970s.

Each is prefixed with ~CC~ for "Creative Computing":

~CC~BAGELS - Guess the computer's chosen 3 digit number
~CC~MATCHES - Play against computer: avoid taking the last match
~CC~MPH - Maths quiz based on times, distances, and speeds
~CC~MUGWUMP - Geometric puzzle: find the "mugwumps"
~CC~REVERSE - Sort numbers by reversing sequences
~CC~TICTAC - Play Tic-Tac-Toe, or Noughts and Crosses
~CC~WEEKDAY - Calculate birth weekday, age, and other stats

Note that all of the programs on these notecards are designed to restart when complete. This means that if run by people to whom you have granted "use" access, but not "command" access, the programs will not end and drop them at a command prompt they cannot use. [SEE 2.1 COMPUTER SETTINGS]

1.7 USING FLOPPIES

As an alternative to storing programs on notecards [SEE 1.5 USING PROGRAM NOTECARDS], you can store them on floppies.

A floppy is a one-prim, copyable and transferable virtual version of a 70s/80s "floppy disk". Each floppy can store one program. A floppy can "save" a program from your computer's memory, and "load" it back there.

To receive a blank floppy, click the floppy lying on the disk drive to the right of the computer. An object called "BBC FLOPPY" will be added to your inventory's "Objects" folder.

Rez it next to your computer. (Or at least, within 5m. The floppy must have a computer to save from or load onto, and will work with whichever computer is closest, and within its 5m range.)

The floppy has floating text showing its name, how much free space it has, and the number of Basic statements it stores.

(Note: this is not quite the same as the number of lines of Basic you'll see in a listing, because in Bright Basic multiple statements can be separated on one line using colons ":". Floppies save and load one statement at a time, so this is the more useful metric.)

Click the floppy to display this menu:

Please click...
[FORMAT] to erase this floppy's program
[NAME] to change the floppy's name
[LIST] to list the stored Basic
[SAVE] to save a program FROM the computer
[LOAD] to load a program ONTO the computer

[FORMAT] [NAME] [LIST]
[SAVE] [LOAD] [LOAD & RUN]

Click [FORMAT] to erase the program stored on a floppy and leave it blank.

Click [NAME] to rename it. A text entry box will appear, and you can enter a name up to 12 characters long, and click Submit.

Click [LIST] to have the Basic program stored on the floppy listed in chat.

Click [SAVE] to save a copy of the program currently in your computer's memory onto the floppy. You can do this even while the program is running - it will not disrupt it (though clicking "break" while the floppy is saving will abort the save). Saving to the floppy will erase whatever program the floppy previously stored - each floppy can store one program at a time, though you can create and keep as many floppies as you wish.

Click [LOAD] to load the program stored on the floppy back onto the computer. If the computer has a program already in memory, that program will be stopped and erased, and the monitor will display "PLEASE WAIT - LOADING FLOPPY". The program from the floppy will be loaded in its place.

Clicking [LOAD & RUN] does much the same as [LOAD], except that once the program has been loaded, it will automatically start to run.

While you are loading and saving, you will see the red LED on the computer's floppy disk drive flash, and the floppy's floating text will be continuously updated to show progress. Programs are loaded and saved at about 5-10 statements per second, depending on the performance of the region.

Once you have stored a program on a floppy, use the [NAME] option to rename it to remind you which program it stores, and right-click and take it back into your inventory. You can then rez it, and load its stored program back onto the computer, at any time.

Remember that the floppy is also copyable and transferable, so you can give a copy of it to someone else, and they can load it onto their own Bright Basic Computer.

It might seem confusing to offer two ways to store programs - notecards, and floppies - but you can use whichever you prefer. The advantage of notecards is that they make it easy to edit a program. But floppies may be useful if you have been typing your program lines directly into the

computer, and want to make a copy of your work, without having to use #LIST to list it into your chat log, and then cut and paste it into a notecard.

Floppies are also rather retro and a little fun, and a cool way to distribute a program. Whether you use notecards, or floppies, or both, is entirely up to you.

2. SETTINGS

The computer and the monitor are separate objects and are configured separately.

2.1 COMPUTER SETTINGS

Click the red "settings" key to display the computer's settings menu:

Input Channel: 9
Startup Program: (NONE)
Autoreset Range: (DISABLED)
Surface: BLACK TP
Use Access: OWNER
Load Access: OWNER
Command Access: OWNER

[CHANNEL] [STARTUP]
[AUTORESET] [SURFACE] [MEMORY]
[USE ACCESS] [LOAD ACCESS] [COM ACCESS]

Click [CHANNEL] to set the chat input channel to between 1 and 9. [SEE 1.3 ENTERING COMMANDS]

Click [STARTUP] to choose a program notecard to load automatically each time the computer is reset. The computer will reset each time it is switched off and back on, "break" is clicked, or its host region restarts. (Choose "(NONE)" in the menu of programs displayed to disable the startup program, so that when reset, the computer simply displays the command prompt ">".)

Click [AUTORESET] to enable or disable automatic reset. When enabled, this setting will cause the computer to automatically reset after anyone uses it, but only once they have left, and no avatars are within the specified range of the computer. This is useful if you wish to allow other people to use the computer, but have it reset itself when they have finished with it, to be ready for the next person.

Click [SURFACE] to choose the computer's surface texture. Various retro colours of textured plastic are available - such Acorn BBC beige, Commodore 64 brown, Tatung Einstein white - plus smooth, scratched, or rusted steel, pale and dark wood, and smooth black.

Click [MEMORY] to see how much RAM the computer has left for storing Basic, and how much LSL memory each of its six chip-emulating scripts has free. [SEE 3.1.3 MEMORY HANDLING]

Click the [ACCESS] buttons to control who can use the computer.

There are three levels of access:

USE ACCESS: Allows the user to interact with a running program. In other words, this access allows a user to enter a value when the program uses the "INPUT" command, [SEE 3.5.13 INPUT], the "GET" command [3.5.9 GET], or the "INKEY" function [SEE 3.4.13 INKEY].

LOAD ACCESS: Allows the user to click the red "load" key, and select a program to load from those on notecards inside the machine. [SEE 1.5 USING PROGRAM NOTECARDS]

COMMAND ACCESS: Allows the user to click the red "break" key to stop a program running, and to enter BASIC commands at the ">" prompt.

You can grant these access levels to yourself as OWNER alone, only to residents who have the same active GROUP as the computer object, or to ALL.

Note that, regardless of these settings, only you as the owner may click the "settings" button to display the settings menu, use power switch to turn the computer on and off, or request or use floppies. [SEE 1.2 USING THE CONTROLS]

When you change a setting, a new button will appear on the menu labelled [APPLY]. You can change multiple settings, and the changes will only take effect when you click [APPLY]. When you apply settings, the computer will clear its screen, but you will not lose the Basic program you had in memory.

2.2 MONITOR SETTINGS

Click the settings button under the screen - which shows a crossed tools icon - to display the monitor's settings menu:

Default colour: WHITE
Typeface: ANKA
Glow Level: 10%
Glass Opacity: 20%
Surface: BLACK TP
Screen Width: 60cm
(Memory free: 23044+18644 bytes)

Please choose a setting to change.

[COLOUR] [TYPEFACE] [GLOW]
[OPACITY] [SURFACE] [WIDTH]

Click [COLOUR] to set the default text colour. Where programs specify that particular sections of text must be shown in particular colours, the default colour will be ignored. But when a program displays text without specifying a colour, or when entering basic commands, text will be shown in this default colour. Options are WHITE, RED, GREEN, BLUE, CYAN, MAGENTA, and YELLOW.

Click [TYPEFACE] to choose the typeface in which text should be shown. Options are ANKA, COURIER, EFFECTS80, NK57, OVERPASS, UBUNTU, and UNISPACE.

Click [GLOW] to choose how strongly displayed text should glow, reminicent of text on 70s CRT screens. Options run between 0% and 100%.

Click [OPACITY] to choose the opacity of the glass screen in front of the text, which is textured and can be slightly opaque to recreate the effect of a 70s CRT screen. Options run between 0% and 100%.

Click [SURFACE] to choose the main surface texture of the monitor. The options are the same as those available for the surface of the computer. [SEE 2.1 COMPUTER SETTINGS]

Click [WIDTH] to choose the width of the screen's text display area. Note that changing this will cause the entire monitor to scale, from a tiny 30cm across, to a huge 5m.

You can change the monitor's settings at any time, even while a program is running, but doing so will clear the screen and display a page of sample text to demonstrate any new settings, disrupting the program's output.

So it is probably best only to change your monitor's settings when a computer is not running a program, and then to switch your computer off and on again to reset the screen.

3. BASIC

The Bright Basic Computer is so called because it runs a command and programming language called Bright Basic, which is an implementation of the BASIC programming language, or "Beginner's All-purpose Symbolic Instruction Code".

This section provides a detailed description of the language, including complete lists of its operators, functions, and commands.

3.1 ABOUT BRIGHT BASIC

Bright Basic draws features from the various BASIC dialects implemented on computers like the Apple I, the Commodore PET, and the Sinclair ZX81 in the late 1970s and early 1980s.

The language will feel familiar to anyone who has programmed such machines.

But each version of Basic is different. Unlike a language like C, for which relatively well-defined standards have been established, each microcomputer manufacturer created their own version of Basic.

Bright Basic aims to be a "common Basic", supporting the most commonly agreed parts of these early dialects, and sometimes supporting multiple ways of doing a single thing to increase compatibility.

The aim was to make it as easy as possible to convert any of the huge number of text-mode Basic programs which were created by hobbyist programmers, and published in computing magazines, when these machines were popular. Many such programs can still be found on the World Wide Web, and below you will find advice on adapting them for the BBC.

But it is also possible to write your own, entirely new programs, and Bright Basic also allows some more modern, structured programming techniques, which should make it easier to do so. These extensions, too, are discussed below.

3.1.1 COMPARISON WITH EARLY BASICS

Every Bright Basic operator, function, and command is described in detail below, but if you are familiar with early dialects of Basic, here are a few points of comparison.

(NOTE: If you are new to Basic programming, these notes may not mean anything to you. Feel free to skip ahead to the description of Bright Basic itself below.)

DATA TYPES: Most Basic dialects were typed languages, but the types available, and the range of values each could store, varied from system to system. All supported integer values, and most also supported reals and strings. Variables usually required suffixes to indicate their type - $ for strings, and sometimes % for integers. For the greatest possible compatibility, Bright Basic is "typeless": any variable may hold any value, and $ and % suffixes are allowed at the end of any variable name. Note that A, A$, and A% are distinct variable names, as was the convention in most Basics. [SEE 3.2.4 VARIABLES]

ARRAYS: Most Basic dialects supported arrays, though with differing restrictions. Some allowed only single letter array names, and some did not allow string arrays. Some also allowed redimensioning, and some did not. Bright Basic arrays may take any name, hold any type of value, and allow redimensioning. [SEE 3.2.5 ARRAYS]

TYPED FUNCTIONS: Some Basic dialects used suffixes to denote the type of value a function yielded (such as the $ to indicate string values yielded by CHR$ and LEFT$), and some did not. For greatest compatibility, Bright Basic allows a $ or % to be added to any function name, and simply ignores it. For instance, Bright Basic's CHR function may also be called CHR$. [SEE 3.4 FUNCTIONS]

GRAPHICS: The BBC is a text-only system, as were many of the systems it was based on. Basic programs which use pixel-level drawing and graphics functions are not supported by Bright Basic.

CURSOR ADDRESSING: Some early Basics could not position text on the screen and worked only in "scroll mode". Others allowed cursor addressing, but using varying functions and with varying text resolutions.

When converting a program which uses cursor-addressing, map coordinates to the BBC's 40 row x 25 column screen, and convert the code to use Bright Basic's AT(row, column) and TAB(column) functions. [SEE 3.4.5 AT]

COLOUR: Where they supported colour at all, basic microcomputers varied widely in how many they offered, and how they were invoked. Map colour use to Bright Basic's COLOUR/COLOUR function, which offers 8 colours plus two text styles: "strong" (or bold) and "inverse". [SEE 3.4.8 COLOUR]

PEEK and POKE: The PEEK function and the POKE statement in many early Basics allowed a programmer to read or write individually addressed bytes in memory. But their effect depended entirely on the hardware and memory map of the machine on which they ran, so Bright Basic cannot support them. If you are trying to convert a program which uses PEEK and POKE, the only way is to find out what was achieved by doing so, and replicate the effect with other Basic commands.

IF/THEN: Some Basic dialects supported only line numbers in THEN and ELSE clauses (IF A=1 THEN 100), while others required complete Basic statements (IF A=1 THEN PRINT "A is 1"). For best compatibility, Bright Basic allows both these syntaxes. Bright Basic also allows multiline conditions using IF/ELSEIF/ELSE/ENDIF. [SEE 3.4.12 IF]

LPRINT/LLIST: These commands were supported in a number of Basic dialects, working like PRINT [SEE 3.5.23 PRINT] and LIST [SEE 3.5.16 LIST], but sending their output to a printer - or "line printer", hence the "L" prefix. Bright Basic supports both these commands, but as it has no access to any real world printer, both these statements output in Second Life chat, "whispering" so that they can be heard within 10m. The chat window essentially acts as a printer analogue.

PRINT: Many Basic dialects allow lists of string values to be printed. The separators between items varied, but Bright Basic uses what seems to be the most common convention: a semicolon (;) between two values to indicate that they should print end to end, a comma (,) to print from the next tab position, and a semicolon at the end of the line to suppress a final linefeed, which was otherwise printed by default. [SEE 3.5.23 PRINT]

INPUT: Some Basics allow a prompt string to be specified in an INPUT command, eg. INPUT "What is your name? ", NAME. However, some use a semicolon (;) between the prompt and variable, some use a comma (,), and some allow a semicolon to be placed after the variable name to suppress any linefeed after the entered value is echoed on the screen. Bright Basic uses a comma between prompt and variable, and allows a trailing semicolon, so INPUT statements from some Basics may require editing. [SEE 3.5.13 INPUT]

INKEY: In some Basic dialects, the INKEY function returned an identifier for the key currently being held down - if any. In others, it returned an identifier for the last key pressed, and removed it from the input buffer. The Bright Basic INKEY function accepts a "pressed now" parameter enabling it to do either, so use INKEY(0) or INKEY(1) depending on the behaviour you need. [SEE 3.4.13 INKEY]

COMPUTED GOTOS: As some Basic dialects allowed computed GOTOs, Bright Basic also allows them. A = 10 : GOTO A + 10 will go to line 20. Note that the target line number need not exist in the program: if line 20 doesn't exist, execution will fall through to the first line number after 20. [SEE 3.5.10 GOTO]

ON GOTO/GOSUB: Some Basic dialects supported ON ... GOTO and ON ... GOSUB, choosing the destination to which to transfer program execution on the basic of an expression yielding a number. A = 2 : ON A GOTO 10, 20, 30 will go to 20. Bright Basic supports these commands. [SEE 3.5.21 ON]

MULTICOMMAND LINES: Some Basics allowed multiple commands to be placed on a single line, usually separated by colons (:), and so Bright Basic also allows this. However, some dialects ignored colons within comments (or REM statements), while others treated these too as command separators. Bright Basic always treats them as command separators, to minimise the danger that a command which meant to be executed is silently ignored, as the PRINT might be in 10 REM Start game : PRINT "START!". This means that a comment containing a colon is likely to be rejected by Bright Basic, eg. 20 REM Programmer: Shan Bright - as SHAN is not a recognised command. This is easily fixed by

replacing the colons in a comment with a dash or semicolon. [SEE 3.5.24 REM]

EVAL: Bright Basic supports the EVAL function found in some but not all Basics, which allows the value of an expression to be interpreted as if it were itself an expression. A = 1 : PRINT EVAL("A+1") prints 2. [SEE 3.4.10 EVAL]

FOR/NEXT: Some Basics require the variable used in a FOR to be repeated after the corresponding NEXT, and some do not allow it. For widest compatibility, Bright Basic allows it, but does not require it: if a variable name is used after a NEXT, it is simply ignored. [SEE 3.5.8 FOR]

TRUTH AND FALSITY: In early versions of Basic, the IF statement had to be followed by an explicit comparison, such as IF X>4 THEN... Other Basics allowed any integer expression to be used, treating them as "true" if they yielded a non-zero result, or "false" if zero. Bright Basic allows any expression after an IF, treating them as "false" if 0 or an empty string, and true otherwise. Comparison operators = <> > < >= and <= are evaluated as ordinary operators which yield 1 or 0. This syntax is backwardly compatible with other Basics, as in IF X>4 THEN..., the expression X>4 will yield 1 or 0. [SEE 3.2.3 LOGICAL VALUES]

BOOLEAN ALGEBRA: Some Basics offered AND, OR, and NOT functions, while others offered these keywords as operators, and yet others used symbols as operators. For compatibility, Bright Basic supports AND, OR, and NOT as functions [SEE 3.4 FUNCTIONS], and &, |, and ! as corresponding symbolic operators [SEE 3.3.8 LOGICAL AND, OR, AND NOT].

ASSIGNMENT IN EXPRESSIONS: Some (few) Basics treated = as an assignment operator even when it was used in expressions, returning the value assigned, so that A=B=7 would assign 7 to a both variables A and B. They would only allow = as an equality operator in IF statements. In Bright Basic, only the first = in a assignment is an assignment operator [SEE 3.3.10 ASSIGNMENT]: all others are equality operators [SEE 3.3.7 EQUALITY]. So A=B=7 first evaluates B=7, yielding 1 (truth) if B is 7, and 0 otherwise. The 1 or the 0 is then assigned to A. This can cause subtle bugs when converting Basic from one of the few dialects which uses the

former system: look out for A=B=C type statements when converting from such a language.

SINGLE CHARACTER OPERATORS: A handful of Basics, including a popular Basic to machine code compiler developed for the Sinclair ZX81, used a heavily restricted syntax which allowed only single character operators, and so used [and] for <= and >=, and # for <>. Bright Basic supports <= and >=, and [and], and <> and # (and also !=). [SEE 3.3.11 SINGLE CHARACTER ALTERNATIVES]

3.1.2 BRIGHT BASIC MODERNISATIONS

Some features have been adopted into Bright Basic from more modern versions of Basic, and from other languages.

The syntax of their implementation was designed to be backwardly compatible with early Basic, so that older programs could still be easily adapted for the BBC. But these extensions should make life easier for those wishing to write new programs.

The most important modernisations are listed below, with references to where each is more fully documented.

1. Multiline IF/ELSEIF/ELSE/ENDIF blocks: Rather than being restricted to using single line IFs and GOTOs to branch execution, as early Basics often were, Bright Basic supports block IF constructs for more structured and readable condition handling. [SEE 3.5.12 IF]

2. Named line labels: In place of inherently arbitrary line numbers, lines in a Bright Basic program can be given meaningful label names, so that instead of GOSUB 1470, you can GOSUB SHOWHELPSCREEN. [SEE 3.5.14 LABEL]

3. Optional line numbers, lower case, and code indentation: Early Basic was made more difficult to read by being written in ALL CAPS, with arbitrary line numbers, and without indentation to show structure. These features of the language all had to be maintained to allow compatability with the early Basic computers the BBC is based on. But if you wish, you can write your programs in a more modern style in a notecard, and have

the computer automatically upcase, remove indents, and add line numbers only when it is loaded. [SEE 1.5 USING PROGRAM NOTECARDS]

4. Improved structured loop support. Many early Basic programs relied on GOTO to loop. Though the language always supported FOR/NEXT loops, programs frequently jumped out of these, littering memory with unused return addresses: or worse, jumped into them. A new EXIT statement allows a program to leave a FOR/NEXT loop cleanly should it be necessary, and generalised LOOP/REPEAT construct allows genuine, condition based looping. [SEE 3.5.19 LOOP]

5. Flexible arrays: Though Bright Basic array syntax is backwardly compatible with early Basic, Bright Basic arrays are far more flexibly implemented, avoiding the need to rigidly dimension them in advance, and even allowing them to be used as indefinite length "lists". [SEE 3.2.5 ARRAYS]

6. Strong multicommand line support. Though many Basics allowed single lines of code to contain multiple commands separated by colons, there were many restrictions on their use, particular in direct or immediate mode. In Bright Basic, you can effectively write an entire program on one line, with commands separated by colons, including internal loops, labels, even DATA statements. For instance:

```
@START : RESTORE : PRINT "The Beatles: "; : LOOP : READ NAME
: WHILE NAME : PRINT NAME; " "; : REPEAT : PRINT : INPUT
"Want to see their names again (Y/N)? ", AGAIN AS "U,T,Y|
N" : IF AGAIN = "Y" THEN GOTO START : DATA "John", "George",
"Paul", "Ringo"
```

This can be a surprisingly useful facility while writing a program, as it allows entire routines to be tested before they are ever added to a program.

7. Improved string handling. Four functions have been added. INS, DEL, and REP allow substrings to be easily inserted, deleted, or replaced within a string, and PAD allows strings to be padded (or optionally truncated) to a fixed width. Bright Basic also supports the common Basic MID, LEFT, and RIGHT functions, and the rarer TRIM, LTRIM, and RTRIM. [SEE 3.4 FUNCTIONS]

8. Combined operation assignments. Most operators may be combined with the assignment operator to simplify an assignment which updates the value of a variable. A = A + 1 may be written A += 1. [SEE 3.3.10 ASSIGNMENT]

9. PRINT tokens: As an alternative to concatenating AT and COLOUR functions into print statements, it is possible to embed simple "tokens" into a string containing cursor positioning and colour information, eg. PRINT "This is {R}red!...{W} now back to white." This can considerably simplify display routines. [SEE 3.5.23 PRINT]

10. INPUT conversion and validation. An optional AS clause can be added to an input statement, which can apply basic, commonly needed conversion and validation to user input, and automatically ask a user to reenter invalid input without additional code. Most Basic programs contain a lot of code simply to validate input, which the AS clause can often replace, eg. INPUT "Enter 1-20, H for help, or Q to quit: " AS "I,1:20,U, Q|H" will let through only the values 1-20, Q, and H. It will even automatically update q and h to Q and H. [SEE 3.5.13 INPUT]

3.1.3 MEMORY HANDLING

(NOTE: this section is rather technical, and you do not need to understand it to use your computer. It is provided purely for those who are interested in how old computers work, and how the Bright Basic Computer emulates them.)

The 70s/80s microcomputers, which the Bright Basic Computer seeks to emulate, each had a small, fixed amount of RAM or "Random Access Memory" in which to store BASIC programs and variable values. These RAM capacities were quoted in "kilobytes" (KB) a unit which was used to represent 1024 bytes.

(The IEC now recommends "kilobyte" be used to mean 1000 bytes, and be styled kB, to conform with the meaning of "kilo" in the SI system, and "kibibyte", KiB, be used for 1024 bytes. As the Bright Basic Computer is modelled on computers which used the older kilobyte=KB=1024 bytes convention, it is retained below.)

The original Commodore Pet, for instance, offered 4KB of RAM, while the Sinclair ZX81 had only 1KB, though many users attached an additional 16KB "RAM pack".

The Bright Basic Computer is an object in Second Life, without dedicated physical RAM chips. Instead, it uses Second Life "Linkset Data" (https://wiki.secondlife.com/wiki/Category:LSL_LinksetData) to store lines of Basic code, variable values, and a few other data items. The amount of available Linkset Data is therefore the closest analogue the BBC has to these memory capacities, and it is limited to 128KB, making the BBC effectively an 128KB microcomputer.

The Bright Basic FREE(0) function will return the number of bytes of memory (ie. LSD storage) the BBC still has available at any point. This is provided as an analogue for the FREE function in some versions of Sinclair Basic, and the FRE(0) function in some Commodore Basics. As the total LSD storage available is 128KB, the FREE(0) function will always return a number between 0 and 131072. [SEE 3.4.11 FREE]

Note that none of this relates to Second LSL script memory. The BBC uses six LSL scripts, each of which emulates a different chip:

BIOS: Basic Input/Output System
ROM: Read Only Memory, with Basic interpreter
CPU: Central Processing Unit
APU: Ancillary Processing Unit, or Coprocessor
FDC: Floppy Drive Controller
NVRAM: Non-volatile RAM, settings storage

The amount of memory each of these chips uses - or has available - is not really dependent on the number of lines of Basic code, or number or size of stored variables. It is affected more by the depth of logic and loop nesting within programs, and the number of recursive levels required to evaluate individual expressions.

Though it is not easily done, it is possible to write code sufficiently complicated to cause one of the BBC's scripts to run out of memory and report a Second Life "Stack/Heap Collision" error. Should this happen,

clicking the "BREAK" key on the keyboard will reset the computers scripts.

To see both how much RAM (ie. linkset data) the computer has available, and how much memory each chip emulating script has available, click the red "settings" button, and choose the "MEMORY" option. [SEE 2.1 COMPUTER SETTINGS]

3.2 DATA HANDLING

This section describes how Bright Basic stores and manipulates data: both literal values embedded in commands, and changeable values stored in variables.

3.2.1 CHARACTER SET

The Bright Basic Computer is an 8-bit system, which works purely in printable ASCII. The supported character set is as follows:

```
ASCII CODES: CHARACTERS
32: SPACE
33-47: !"#$%&'()*+,-./
48-57: 0123456789
58-64: :;<=>?@
65-90: ABCDEFGHIJKLMNOPQRSTUVWXYZ
91-96: [\]^_`
97-122: abcdefghijklmnopqrstuvwxyz
123-126: {|}~
```

Characters other than these 95 - such as modern Unicode characters beyond the first Unicode block - are stripped from text entered.

3.2.2 LITERALS

Literal strings must be enclosed in quotation marks when embedded in a Basic statement.

```
10 PRINT "Hello World"
```

Quotation marks may be included within literal strings by repeating them.

```
20 PRINT "Tony ""Scarface"" Montana"
```

Numeric values may be written without quotation marks, and may be prefixed by a minus, and contain a decimal point.

```
30 A = 2
40 PI = 3.14159
50 NEGATIVEPI = -3.14159
```

Sometimes, in some very specific cases, an operator or a function which would normally return a number will return the string "NaN" (meaning "Not a Number"), or "Infinity".

```
60 PRINT -1^0.5 :' Square root of -1, prints NaN
70 PRINT 99^99 :' Too large, prints Infinity
80 PRINT 1/0 :' Division by zero, prints Infinity
```

These values are not numbers, but are mentioned here as they can be returned by functions which normally return numbers. Note that neither "NaN" nor "Infinity" are quite mathematically correct: the square root of -1 is a number, it is just an imaginary number, and 99^99 is not infinite, just too big to handle at 198 digits long. Both simply refer to particular kinds of out-of-range results.

3.2.3 LOGICAL VALUES

Some commands need to evaluate an expression as either "true" or "false" - or in other words, as a logical or boolean value.

```
100 IF SCORE > 100 THEN PRINT "Well done!"
```

This IF statement must evaluate the expression SCORE > 100 as a boolean value. If it yields "true", if must execute the THEN clause.

In Bright Basic, any expression can be treated as a boolean. If is considered "false" if it is 0 or an empty string, and it is "true" if it is anything else.

In this example, the operator > yields 1 if the value before it is greater than the value after, and 0 otherwise. The 1 yielded by the > operator causes the IF statement to trigger the THEN clause.

Note that you can store such values in variables, or use them in expressions, like any other: they need not only appear in IF statements.

```
100 HASHIGHSCORE = SCORE > 100
```

If you now wished to give the player an extra game life for having a high score, you could do it like this:

```
110 LIFE += HASHIGHSCORE
```

And you could then use the stored logical value in an IF condition.

```
120 IF HASHIGHSCORE THEN PRINT "Well done!"
```

3.2.4 VARIABLES

Variable names must begin with a letter, and may contain only letters or numbers, except for the last character which may be a $ or a %. The following are all valid variable names:

```
X
PLAYER1
A$
VALUE%
```

The $ and % suffixes are supported purely for compatibility with other Basics: they are never required. In some Basic dialects, variables which store strings must have names ending in $, and those which store integers must have names ending in %. Additionally, some Basics impose various limits on the length of variable names.

None of these restrictions apply in Bright Basic: variable names are not limited in length, and any variable may store any value. The suffixes are supported simply to make it easier to adapt programs written for other computers to run on the BBC, without having to change their variable names.

Note that, as is usual in other Basics, A, A$, and A% are three different and distinct variable names.

3.2.5 ARRAYS

An array variable may be dimensioned with a DIM statement, the length of the array being specified in brackets. All elements of the array are initialised to empty strings.

```
10 DIM LETTERS(26)
```

Elements of an array may be individual referenced by specifying a subscript between parentheses. An subscript value of 1 refers to the first element, 2 to the second, and so on.

```
20 LETTERS(1) = "A"
30 PRINT LETTERS(1) :' Prints A
```

Reading out-of-range elements does not cause an error or program halt: the value yielded is simply an empty string.

```
40 PRINT LETTERS(30) :' Prints nothing, outside 1-26
50 PRINT LETTERS(-4) :' Prints nothing, outside 1-26
```

The elements of an array may all be given a single initial value by adding an assignment to the DIM statement.

```
60 DIM PLAYERSCORES(4) = 99
70 PRINT PLAYERSCORES(1) :' Prints 99
```

Array variables are actually simply ordinary variables, which store values in a string delimited by backtick (`) symbols.

```
80 PRINT PLAYERSCORES :' Prints 99`99`99`99
```

The DIM statement is therefore optional, though useful for compatibility, and when you wish to assign initial values to all the elements of an array at once. But even without using DIM, any variable which has been initialised by giving it a value may be referenced as an array using subscripts.

```
90 MYVAR = "Some value"
100 MYVAR(2) = "Another value"
110 PRINT MYVAR :' Prints Some value`Another value
```

However, attempting to access a subscripted element of a variable which has never been initialised at all will cause an error.

```
120 PRINT UNINITIALISEDVAR(2) :' Causes error
```

Note that assigning an element beyond the number of elements an array contains does not cause an error, as ` markers are added to expand the number of elements.

```
130 NEWVAR = "Element 1"
140 NEWVAR(5) = "Element 5"
150 PRINT NEWVAR :' Prints Element 1````Element 5
```

The backtick ` operator concatenates two values around a literal backtick. It can therefore be used to create an entire array with individually assigned elements in a single expression.

```
160 X = 3
170 TRIPLES = X`2*X`3*X`4*X`5*X
180 PRINT TRIPLES :' Prints 3`6`9`12`15
190 PRINT TRIPLES(4) :' Prints 12
```

The `= operator combines an assignment with the ` operator, effectively adding an element to the end of an array.

```
200 TRIPLES `= 6*X
210 PRINT TRIPLES(6) :' Prints 18
```

If the initial values are fixed, arrays can also be created using a literal string.

```
220 VOWELS = "A`E`I`O`U"
230 PRINT VOWELS(2) :' Prints E
```

As the number of elements in an array can be varied as a program runs, elements can be counted using the ALEN function. In the same way that LEN measures the number of characters in value, ALEN measures the number of array elements. [SEE 3.4.2 ALEN]

```
240 PRINT ALEN(VOWELS) :' Prints 5
```

As can be seen, Bright Basic arrays are more flexible and forgiving than those in most Basics, allowing them to take any name, store any type of data, and even allow arrays to be used as "lists" of indefinite length.

This liberal syntax is designed to be backwardly compatible with Basic programs apply more rigid, but different, restrictions on array use.

A wrinkle in this approach is that as elements are separated by backticks `, an element cannot actually contain a backtick. Assigning a value which contains a backtick to an element in an array effectively splits that element into two elements. This can be a useful way to insert elements into the middle of an array.

3.3 OPERATORS

Bright Basic's operators are listed here, in order of precedence.

Parentheses ()
Exponentiation ^
Multiplication and division * / \
Addition and subtraction + -
Concatenation ~
Comparison >= <= > <
Equality = <>
Logical AND &
Logical OR |
Logical NOT !
Array building `
Assignment = += -= *= /= ~= ^= &= |=

Operators listed on the same line have equal precedence, and are evaluated from left to right.

```
>PRINT 20/5*3
12
```

In this example, multiplication and division have equal precedence, and so these operations were applied from left to right. The division 20/5 yields 4, which is then multiplied by 3 to yield 12.

Note that the result would have been different if these operations were done in a different order. Using parentheses not only ensures the operations are done in the order you intended, they make that order of operations obvious to anyone who reads your code in the future.

```
>PRINT 20/(5*3) :' Reverses usual order of operations
1.333333
>PRINT (20/5)*3 :' Usual order, but made obvious
12
```

3.3.1 PARENTHESES ()

The contents of parentheses are evaluated before the rest of an expression, allowing them to be used to change the order in which operators are applied.

```
10 PRINT 1+2*3 :' Prints 7
20 PRINT (1+2)*3 :' Prints 9
```

3.3.2 EXPONENTIATION (^)

The ^ operator raises one number to the power of another.

```
10 PRINT 3^2 :' Prints 9 (3 squared)
20 PRINT 2^3 :' Prints 8 (2 cubed)
30 PRINT 10^0.5 :' Prints 3.162278 (square root of 10)
```

Expressions yielding imaginary or complex numbers, like the square root of -1, return "NaN" ("Not a Number"). [SEE 3.2.2 LITERALS]

```
40 PRINT -1^0.5 :' Prints NaN
```

3.3.3 MULTIPLICATION AND DIVISION (*, /, \)

The operators * and / will multiply and divide. The operator \ does integer division, in which the fractional part of the result is discarded.

```
10 PRINT 3*2 :' Prints 6
20 PRINT 3/2 :' Prints 1.5
30 PRINT 3\2 :' Prints 1
```

Division by zero yields "Infinity". [SEE 3.2.2 LITERALS]

```
40 PRINT 1/0 :' Prints Infinity
```

3.3.4 ADDITION AND SUBTRACTION (+, -)

The operators + and - will add or subtract values.

```
10 PRINT 1+2 :' Prints 3
20 PRINT 1-2 :' Prints -1
```

Note that while numeric literals may begin with a minus (-) to indicate that they are negative, a unary minus may not be placed before other expressions to reverse their sign, as the - operator requires two operands. But the solution is simple. If you wish to negate the value of an expression, use "0-expression".

```
30 A=30
40 PRINT 0-A :' Prints -30
```

3.3.5 CONCATENATION (~)

The operator ~ joins two strings into one.

```
10 A = 1
20 B = 2
30 PRINT A~B :' Prints 12
```

3.3.6 COMPARISON (>=, <=, >, <)

These operators will compare two values to see if the first is greater than or equal (>=), less than or equal (<=) greater than (>), or less than (<), the second.

Each operator yields 1 if the result is true, or 0 if false.

```
10 PRINT 99 >= 100 :' Prints 0
20 PRINT 99 <= 100 :' Prints 1
30 PRINT 99 > 100 :' Prints 0
40 PRINT 99 < 100 :' Prints 1
```

If both values are numeric, the comparison will test their numeric values - but if either or both are non-numeric, the comparison will compare them alphabetically, as strings of characters.

```
50 PRINT "99A" >= 100 :' Prints 1
60 PRINT "99A" <= 100 :' Prints 0
70 PRINT "99A" > 100 :' Prints 1
80 PRINT "99A" < 100 :' Prints 0
```

3.3.7 EQUALITY (=, <>, !=)

These operators will compare two values to see if the first is equal to (=), or not equal to (<>), the second.

They return 1 if the result is true, or 0 if false.

If both values are numeric, they are compared by their numeric value:

```
10 PRINT 99 = 99 :' Prints 1, exactly the same
20 PRINT 99 <> 100 :' Prints 1, different
30 PRINT 99 = 099 :' Prints 1, same number value
```

But if either or both are non-numeric, they are compared as strings of characters.

```
40 PRINT "A" = "B" :' Prints 0, different strings
50 PRINT 99 <> "99A" :' Prints 1, different strings
```

Though most Basic dialects use <> for inequality, some use !=, so both are supported for compatibility.

```
60 PRINT "A" != "B" :' Prints 1, different strings
```

3.3.8 LOGICAL AND, OR, AND NOT (&, |, !)

These operators will test and combine the logical, boolean value of operands.

A value is "false" if it is 0 or an empty string, and "true" otherwise. [SEE 3.2.3 LOGICAL VALUES]

The logical AND operator, &, will yield 1 (representing "true") if both its operands are "true", and 0 (representing "false") otherwise.

The logical OR operator, |, will yield 1 (represeting "true") if either or both of its operands are "true", and 0 (representing "false") otherwise.

The logical NOT operator precedes a single operand, and returns 1 (representing "true") if it is "false", and 0 (representing "false") if it is "true".

```
10 PRINT "A" & 1 :' Prints 1 as "A" and 1 both "true"
20 PRINT "" | 0 :' Prints 0 as "" and 0 both "false"
30 PRINT !"A" :' Prints 0 as "A" is "true"
```

3.3.9 ARRAY BUILDING (`)

Two or more values separated by backticks (`) are joined into an array. [SEE 3.2.5 ARRAYS]

```
10 CUBES = 1`8`27`64`125`216`343`512`729`1000
20 PRINT CUBES(7) :' Prints 343
```

3.3.10 ASSIGNMENT = += -= *= /= #= ^= &= |= `=

The = operator assigns a value to a variable, or variable element.

```
10 SCORE = 100
20 PLAYERNAMES(1) = "Jane"
```

It may be used after an optional LET statement, or a DIM statement, in which it assigns an initial value to each element in an array.

```
30 LET TARGETSCORE = 5000
40 DIM PLAYERHEALTHS(4) = 100
```

But aside from LET or DIM, nothing may appear to the left of the variable or variable element being assigned.

The operators += -= *= /= ~= ^= &= |= `= each combine an operation with an assignment.

```
50 A = 1
60 A += 9 :' Equivalent to A = A + 9
70 PRINT A :' Prints 10
80 A -= 2 :' Equivalent to A = A - 2
90 PRINT A :' Prints 8
100 A *= 5 :' Equivalent to A = A * 5
110 PRINT A :' Prints 40
120 A /= 4 :' Equivalent to A = A / 4
130 PRINT A :' Prints 10
140 A ~= "0" :' Equivalent to A = A ~ "0"
150 PRINT A :' Prints 100
160 A ^= 0.5 :' Equivalent to A = A ^ 0.5
```

```
170 PRINT A :' Prints 10
180 A &= 0 :' Equivalent to A = A & 0
190 PRINT A :' Prints 0
200 A |= 1 :' Equivalent to A = A | 1
210 PRINT A :' Prints 1
220 A `= 1 :' Equivalent to A = A ` 1
230 PRINT A :' Prints 1`1
```

3.3.11 SINGLE CHARACTER ALTERNATIVES (#,], [)

Three operators - <> (not equal to), >= (greater than or equal to), and <= (less than or equal to), are two characters long.

A few (rather early and obscure) dialects of Basic used only single character operators, to speed and simplify interpretation, and save precious bytes of memory, on early, underpowered computers.

Therefore they used # instead of <>, [instead of <=, and] instead of >=.

For compatibility, Bright Basic also supports these operators as alternatives.

```
10 PRINT 1#2 :' Same as 1<>2, prints 1 (true)
20 PRINT 1[2 :' Same as 1<=2, prints 1 (true)
30 PRINT 1]2 :' Same as 1>=2, prints 0 (false)
```

They have the same meaning and precedence as the operators for which they may be substituted.

3.4 FUNCTIONS

Bright Basic supports most common Basic functions.

Note that some (but not all) Basic dialects added $ or % suffixes to function names to indicate that the functions returned strings or integers. For instance, both Commodore Pet Basic and Sinclair Basic supported a CHR$ function, which returned the character corresponding to a particular character code.

For compatability, Bright Basic allows, and then ignores, a $ or % suffix on any function name. So it will evaluate both CHR(88) and CHR$(88) identically - as "X".

3.4.1 ABS(number)

Returns the absolute value of {number}.

```
10 PRINT ABS(7) :' Prints 7
20 PRINT ABS(-7) :' Prints 7
```

3.4.2 ALEN(array)

Returns the length of an array.

As any variable in Bright Basic may be treated as an array, which is simply a string in which elements are separated by ` symbols, this function essentially returns the number of ` symbols in a value, plus 1. [SEE 3.2.5 ARRAYS]

```
10 DIM LETTERS(26)
20 PRINT ALEN(LETTERS) :' Prints 26
30 VOWELS = "A`E`I`O`U"
40 PRINT ALEN(VOWELS) :' Prints 5
50 EMPTYSTRING = ""
60 PRINT ALEN(EMPTYSTRING) :' Prints 1
```

In the examples above, ALEN was passed a single variable, and this is a common use for the function. But it is an ordinary function, which can be passed any expression.

```
10 PRINT ALEN(1`2`3`4`5) :' Prints 5
20 PRINT ALEN("THIS`HAS`SEPARATORS") :' Prints 3
```

3.4.3 AND(value1, value2)

Returns 1 (indicating "true") if both {value1} and {value2} are "true", or 0 (indicating "false") otherwise. All values except 0 and the empty string "" are "true". [SEE 3.2.3 LOGICAL VALUES]

```
10 PRINT AND(1, "A") :' Prints 1
20 PRINT AND(1, 0) :' Prints 0
30 PRINT AND(1, "") :' Prints 0
```

This function is provided for compatibility with Basics which offered functions in place of logical operators. Bright Basic also supports the and operator &. [SEE 3.3.8 LOGICAL AND, OR, AND NOT]

3.4.4 ASC(string)

Returns the ASCII character code of the first character in the string passed, or 0 if the string is empty.

```
10 PRINT ASC("M") :' Prints 77
20 PRINT ASC("Multicharacter string") :' Prints 77
30 PRINT ASC("") :' Prints 0
```

3.4.5 AT(row, column) / AT(column)

AT returns a string which, when sent by PRINT to the monitor, will set the printing position to {row} and {column}.

```
10 REM Print "Hello!" at row 3, column 12
20 PRINT AT(3,12); "Hello!"
```

If AT used with a single parameter, it too sets only the {column} position.

```
10 REM Print "Hello!" at columns 20
20 PRINT AT(20); "Hello!"
```

The monitor has 25 rows, each 40 columns wide, so {row} and {column} coordinates outside these ranges are ignored.

3.4.6 CASE(text, upcase)

Returns {text} upcased if {upcase} is true, or downcased if {upcase} is false.

```
10 LANGUAGE = "Bright Basic"
20 PRINT CASE(LANGUAGE, 1) :' Prints BRIGHT BASIC
30 PRINT CASE(LANGUAGE, 0) :' Prints bright basic
```

3.4.7 CHR(asciicode)

Returns the ASCII character corresponding to the numeric {asciicode} passed. The code must correspond to a printable ASCII character, so it must be between 32 and 126: an empty string will be returned otherwise. [SEE 3.2.1 CHARACTER SET]

```
10 PRINT CHR(77) :' Prints M
20 PRINT CHR(200) : ' Prints nothing
```

3.4.8 COLOUR(number) / COLOR(number)

Returns a string which, when sent by PRINT to the monitor, will set the colour in which text should be printed.

The <number> parameter is calculated by adding together +1 for red, +2 for green, +4 for blue, +8 for bold text, and +16 for inverse text.

```
10 PRINT COLOUR(1); "Red (1=red)"
20 PRINT COLOUR(3); "Yellow (1=red+2=green)"
30 PRINT COLOUR(12); "Bold Blue (4=blue+8=bold)"
40 PRINT COLOUR(18); "Inverse Green (2=green+16=inv.)"
```

Red, green, and blue combine as follows: red + green = yellow (3), red + blue = magenta (5), green + blue = cyan (6), and red + blue + green = white (7).

If <number> is 0, text will be printed in the monitor's default text colour. With factory settings, the monitor's text colour is white, but you can change this. [SEE 2.2 MONITOR SETTINGS]

```
10 PRINT COLOR(0); "Default colour"
20 PRINT COLOR(8); "Bold default"
30 PRINT COLOR(16); "Inverse default"
40 PRINT COLOR(24); "Inverse bold default"
50 PRINT COLOR(0); "Back to default"
```

For compatibility with Basic dialects used by British computers (such as the Acorn BBC Micro) and US computers (such as the Commodore 64), both COLOUR and COLOR keywords are supported.

3.4.9 DEL(string, start, length)

The DEL ("delete") function returns {string}, with {length} characters deleted, beginning with {start}.

```
10 PRINT DEL("abcde", 2, 2) :' Prints ade
```

If {start} is less than 1, or greater than the length of {string}, the function simply returns {string} unmodified.

```
20 PRINT DEL("abcde", 10, 2) :' Prints abcde
```

3.4.10 EVAL(expression)

Returns the value of (expression), evaluated as a Bright Basic expression.

This is best demonstrated by example.

```
10 PRINT EVAL("2+2") :' Prints 4
20 X=-101
30 PRINT EVAL("X*9") :' Prints -909
40 PRINT EVAL("ABS(X)") :' Prints 101
```

The use of EVAL may not be immediately obvious, as it simply seems to evaluate expressions which you could simply write into the body of the program.

But the key point is that, using EVAL, a program can create its own expressions in memory at runtime, and evaluate them - expressions which were never written into the program.

For instance, the following program allows a user to perform arithmetic calculations:

```
10 INPUT "Enter first number: ", NUM1 AS "N"
20 INPUT "Enter operator (+,-,*,/): ", OP AS "+|-|*|/"
30 INPUT "Enter second number: ", NUM2 AS "N"
40 PRINT "The result is: "; EVAL(NUM1~OP~NUM2)
```

While you could write a calculator program without EVAL, it would take many more lines of code, including a string of IF statements to determine which OP (operation) had been chosen, and separate lines of calculation for

each. EVAL allows the expression solving power of Basic to be applied without having to reprogram them by hand.

3.4.11 FREE(0)

Returns the number of bytes of virtual RAM free for the storage of Basic and variable values. [SEE 3.1.3 MEMORY HANDLING]

(The argument is a dummy. It must be valid, but is ignored. Use 0.)

```
10 PRINT FREE(0)
```

The function is provided partly for compatibility with various Basic dialects (like the version of Sinclair Basic used on the Timex computer in the US which offers FREE, and Commodore 64 Basic which supports FRE(0)).

3.4.12 IIF(condition, trueresult, falseresult)

The "immediate if" function evaluates {condition}, returning {trueresult} if it is a number , or {falseresult} if false.

```
10 PRINT IIF(1<2, "TRUE", "FALSE") :' Prints TRUE
```

The {condition} is false if it is a number equal to zero, or an empty string, and true otherwise. [3.2.3 LOGICAL VALUES]

3.4.13 INKEY(pressednow)

The INKEY function returns the name of a clicked key.

If {pressednow} is "true" - as in INKEY(1) - the function will return the name of the key currently being clicked, while the user clicking it is actually holding down the left mouse button. Repeated calls to INKEY(1) during a keypress will continue to return the name of this key until the user releases the mouse button. When no key is being pressed, INKEY(1) will return an empty string.

If {pressednow} is "false" - as in INKEY(0) - the function will return the name of the last key which was pressed, whether or not the mouse button is still being held down. But it will only return a key name once for each

keypress - repeated calls to INKEY(0) during a keypress will return an empty string until the key mouse button has been released, and a new keypress has begun.

Run this program, and try clicking keys, holding down the mouse button, and then releasing them, to see the effect. CURRENTKEY will always show the key currently being clicked. LASTKEY will show a key's name only once, at the beginning of each click.

```
10 CURRENTKEY = INKEY(1)
20 LASTKEY = INKEY(0)
30 PRINT CURRENTKEY;"|";LASTKEY;" ";
40 GOTO 10
```

The computer has 58 keys. The four red keys - "help", "load", "settings", "break" - have special functions, and are ignored by INKEY. The values returned by INKEY when one of the remaining 54 keys was the last pressed are as follows:

26 alphabetic keys: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

10 numeric keys: 0 1 2 3 4 5 6 7 8 9

11 punctuation keys: ~ - = ' [ ] \ ; . , /

6 editing keys: ERASE, TAB, SHIFTLOCK, ENTER, LEFTSHIFT, and RIGHTSHIFT

Space bar: a space.

Note that INKEY ignores the use of SHIFT keys. When the key bearing the legend "2 @" is clicked, INKEY returns 2, never @.

Also, INKEY only returns the names of keys clicked: it ignores chat input. To allow a user to enter entire strings using chat input (or multiple clicks ending in ENTER), use the INPUT statement. [SEE 3.5.13 INPUT]

3.4.14 INS(string, start, substring)

The INS ("insert") function returns {string}, with {substring} inserted from character {start}.

```
10 PRINT INS("abcde", 2, "XX") :' Prints aXXbcde
```

If {start} is less than 1, or greater than the length of {string}, the function simply returns {string} unmodified.

```
20 PRINT INS("abcde", 10, "XX") :' Prints abcde
```

3.4.15 INSTR(string, substring)

Returns the position of {substring} within {string}, returning 0 if it does not appear there.

```
10 PRINT INSTR("ONE TWO", "ONE") :' Prints 1
20 PRINT INSTR("ONE TWO", "TWO") :' Prints 5
30 PRINT INSTR("ONE TWO", "THREE") :' Prints 0
```

If substring is an empty string, INSTR will always return 1.

```
40 PRINT INSTR("ONE TWO", "") :' Prints 1
```

3.4.16 INT(number)

Returns the integer part of a number.

```
10 PRINT INT(1) :' Prints 1
20 PRINT INT(1.5) :' Prints 1
30 PRINT INT(-1.5) :' Prints -1
```

3.4.17 JUMP(columns)

Returns a string which, when sent by PRINT to the monitor, will "jump" the printing position on the monitor forwards or backwards by {columns}, which may be between -40 and 40.

```
10 PRINT "abc"; JUMP(-1); "X" :' Prints abX
```

If {columns} is less than -40, or more than 40, this function returns nothing.

Jumping forwards past the last column will move the printing position onto the beginning of the next row. If done on the bottom row, this will cause the screen to scroll.

Jumping back past the beginning of a row will move the printing position onto the end of the row above. It is not possible to jump back past the beginning of the first row.

3.4.18 LEFT(string, length)

Returns the first {length} characters of {string}.

```
10 PRINT LEFT("abcde", 2) :' Prints ab
```

3.4.19 LEN(string)

Returns the length of (or number of characters in) {string}.

```
10 PRINT LEN("abcde") :' Prints 5
```

3.4.20 MID(string, start, length)

Starting from character {start}, returns {length} characters from {string}.

```
10 PRINT MID("abcde", 3, 2) :' Prints cd
```

If {length} is zero, or is omitted altogether, then MID returns everything from character {start} to the end of the string.

```
20 PRINT MID("abcde", 3) :' Prints cde
```

3.4.21 MOD(number, divisor)

Performs a modulo operation, returning the remainder left when {number} is divided by {divisor}.

```
10 PRINT MOD(47, 10) :' Prints 7
```

Note that if the divisor is zero, this function will return 0.

```
20 PRINT MOD(10, 0) :' Prints 0
```

3.4.22 NOT(value)

Returns 1 (indicating "truth") if {value} is "false", and 0 (indicating "false") if it is "true". All values except 0 and the empty string "" are true. [SEE 3.2.3 LOGICAL VALUES]

```
10 PRINT NOT(0) :' Prints 1
20 PRINT NOT("") :' Prints 1
30 PRINT NOT(1) :' Prints 0
```

This function is provided for compatibility with Basics which offered functions in place of logical operators. Bright Basic also supports the not operator !. [SEE 3.3.8 LOGICAL AND, OR, AND NOT]

3.4.23 OR(value1, value2)

Returns 1 (indicating "true") if either {value1} or {value2} are "true", or 0 (indicating "false") otherwise. All values except 0 and the empty string "" are "true". [SEE 3.2.3 LOGICAL VALUES]

```
10 PRINT OR(1, 0) :' Prints 1
20 PRINT OR("A", "") :' Prints 1
30 PRINT OR(0, "") :' Prints 0
```

This function is provided for compatibility with Basics which offered functions in place of logical operators. Bright Basic also supports the or operator |. [SEE 3.3.8 LOGICAL AND, OR, AND NOT]

3.4.24 PAD(string, length, char, left, truncate)

When given just {string} and {length} parameters, PAD returns a string with spaces added to bring it up to {length}. If {string} is already equal to or greater than {length}, it is returned unmodified.

```
10 PRINT "["; PAD("A", 3); "]" :' Prints [A  ]
20 PRINT PAD("ABCD", 3) :' Prints ABCD
```

If specified, the first character of {char} is used to pad {string}.

```
30 PRINT PAD("A", 3, "*") :' Prints A**
```

If {left} is specified and "true", {string} is padded on the left rather than the right.

```
40 PRINT PAD("A", 3, "*", 1) :' Prints **A
```

Finally, if {truncate} is specified and "true", then if {string} is longer than {length}, it will be truncated.

```
50 PRINT PAD("ABCD", 3, "*", 1, 1) :' Prints ABC
```

3.4.25 REP(string, start, substring, length)

The REP ("replace") function returns {string}, with {length} characters removed from character {start}, and {substring} inserted in their place.

```
10 PRINT REP("abcde", 2, "XXXXX", 1) :' Prints aXXXXXcde
```

If the final {length} parameter is omitted, then the number of characters replaced is equal to the length of {substring}.

```
20 PRINT REP("12345", 3, "--") :' Prints 12--5
```

If {start} is less than 1, or greater than the length of {string}, the function simply returns {string} unmodified.

```
30 PRINT REP("abcde", 10, "XXXXX", 1) :' Prints abcde
```

3.4.26 RIGHT(string, length)

Returns the last {length} characters of {string}.

```
10 PRINT RIGHT("abcde", 2) :' Prints de
```

3.4.27 REPEAT(string, count)

Returns {string} repeated {count} times.

```
10 PRINT REPEAT("Ho!", 3) :' Prints Ho!Ho!Ho!
```

If {string} is a single space, you can alternatively use the SPC function. [SEE 3.4.29 SPC]

```
20 REM The line below prints "Same!"
30 IF REPEAT(" ", 3) = SPC(3) THEN PRINT "Same!"
```

3.4.28 RND(0)

The RND function returns a random number greater than or equal to 0, and less than 1.

The parameter passed to the function is ignored. In some dialects of Basic, this parameter was used to specify how the random number sequence should be seeded, and their values and meaning was specific to the hardware on which they ran. For compatibility, Bright Basic accepts, and ignores, any value passed. Seeding is done by Second Life's servers.

RND can be combined with INT to randomly return an integer in a fixed range. [SEE 3.4.16 INT]

```
10 PRINT 1+INT(RND(0)*6) :' Prints 1-6, like rolling a die
```

3.4.29 SPC(number)

Returns a string of spaces of length {number}.

```
10 PRINT "["; SPC(3); "]" :' Prints [   ]
```

A string of spaces may also be created using the REPEAT function, but though SPC is less flexible than REPEAT, in the specific case where spaces are needed SPC is marginally faster, and as this function is supported by a number of dialects of Basic, it is supported for compatibility. [SEE 3.4.27 REPEAT]

3.4.40 TAB(column)

TAB returns a string which, when sent by PRINT to the monitor, will set the printing position to {column}.

```
10 REM Print "Hello!" at columns 20
20 PRINT TAB(20); "Hello!"
```

The monitor is 40 columns wide, so if {column} is less than 1, or more than 40, the function returns an empty string.

TAB is equivalent to the AT function when AT is used with a single coordinate, and is supported for compatibility with other Basic dialects.

3.4.41 TIME(0)

The TIME function returns the amount of time, in seconds, that the program has been running.

The parameter passed to the function is ignored. In some dialects of Basic which support this or similar functions, the parameter determined the unit in which time was measured, often related to a hardware specific "tick rate". Bright Basic TIME(0) always returns seconds.

```
10 PRINT "Welcome to the typing test!"
20 STARTTIME = TIME(0)
30 INPUT "Type in the alphabet in caps! - ", LETTERS
40 IF LETTERS = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" THEN
50 PRINT "Your time was "; TIME(0) - STARTTIME; "s"
60 ELSE
70 PRINT "Sorry, incorrect."
80 ENDIF
```

3.4.42 TRIM/LTRIM/RTRIM(string)

These functions will return {string} with spaces removed from one or both ends: LTRIM removing them from the beginning, RTRIM from the end, and TRIM from both.

```
10 X = " X "
20 PRINT "["; LTRIM(X); "]" :' Prints [X ]
30 PRINT "["; RTRIM(X); "]" :' Prints [ X]
40 PRINT "["; TRIM(X); "]" :' Prints [X]
```

3.5 COMMANDS

This section lists the commands supported by Bright Basic.

(Basic commands are often referred to as "statements". This is a well-established usage, but the term command is used in this manual as it is more descriptive of their purpose. Do not be confused: a "PRINT statement" and a "PRINT command" are the same.)

3.5.1 BEEP tone

Produces a beep sound. The {tone} is a number between 1 and 4, in descending order of pitch. 1 is a high pitch beep, 2 a neutral or generic beep, 3 a low beep, and 4 a very low beep or error tone. (The fourth tone is also produced when an the user enters a value which does not match the AS clause of an INPUT statement. [SEE 3.5.13 INPUT])

If {tone} is not in the range 1-4, or is omitted altogether, a value of 2 is assumed, and a generic beep is sounded.

```
10 BEEP :' Plays a generic beep
20 WAIT 0.5 : BEEP 1 :' Plays high beep
30 WAIT 0.5 : BEEP 2 :' Plays generic beep
40 WAIT 0.5 : BEEP 3 :' Plays low beep
50 WAIT 0.5 : BEEP 4 :' Plays v. low beep
```

This command is supported as a convenient way to create simple beep sounds, and for compatibility with some dialects of Basic. Compare the "PLAY" command, which can play any Second Life sound item for which you know the UUID, or have an item you can put in the computer's inventory. [SEE 3.5.22 PLAY]

3.5.2 CLEAR

Clear all variables from memory.

```
10 A=10
20 PRINT "["; A; "]" :' Prints [10]
30 CLEAR
40 PRINT "["; A; "]" :' Prints []
```

3.5.3 CLS

Clears the screen.

```
10 CLS
20 PRINT "<- Top-left corner of clear screen"
```

3.5.4 DATA value, value... / READ variable / RESTORE

The DATA statement stores a series of data values, separated by commas.

The READ statement takes a single value stored using a DATA statement, and assigns it to a variable.

The program maintains a "pointer" telling it which data value to READ next. Starting from the first value in the first DATA statement, each READ statement reads a value, and sets the pointer to the next.

When a READ runs out of data values to read, it assigns the empty string ("") to a variable.

The RESTORE command resets the pointer to the first data value.

```
10 @LISTBEATLES
20 RESTORE
30 PRINT "The Beatles:";
40 @NEXTBEATLE
50 READ NAME
60 IF NAME = "" THEN GOTO ENDBEATLES
70 PRINT " ";NAME;
80 GOTO NEXTBEATLE
90 @ENDBEATLES
100 PRINT
110 INPUT "Want to see them again?", REPEAT
120 IF REPEAT = "Y" THEN GOTO LISTBEATLES
130 DATA "John", "Paul"
140 DATA "George", "Ringo"
```

This program will print...

```
The Beatles: John Paul George Ringo
```

You can add as many DATA statements as you wish to a program, and place them anywhere in the listing.

A DATA statement may contain only literal values - fixed numbers and strings - and not expressions.

3.5.5 DELETE linenumber / from, to

Delete lines of the BASIC program in memory.

DELETE alone deletes the entire program. (Compare NEW, which also deletes all the lines, but also clears all variable values. [SEE 3.5.20 NEW])

If a single {linenumber} is specified, just that line (if it exists) is deleted. So DELETE 20 will delete only line 20. (You can also delete a single line simply by entering its line number without a statement - just enter 20 to delete line 20.)

If a comma is used, the line to delete {from} appears before it, and the line to delete to {to} appears after. Note that neither {from} nor {to} need refer to existing lines - any lines in the range will be deleted.

This means DELETE 10;20 will delete lines between 10 and 20, DELETE ;20 will delete all lines up to 20, and DELETE 20; will delete all lines from 20 to the end of the program.

```
>LIST
10 PRINT "First line"
20 PRINT "Second line"
30 PRINT "Third line"
3 line(s) listed.
>DELETE 20
1 line(s) deleted.
>LIST
10 PRINT "First line"
30 PRINT "Third line"
2 line(s) listed.
>
```

3.5.6 DIM array(elementcount) = value

DIM initialises an array with {elementcount} elements, setting each element to {value}.

```
10 DIM PLAYERNAMES(4) = "New Player"
20 PRINT ALEN(PLAYERNAMES) :' Prints 4
30 PRINT PLAYERNAMES(1) :' Prints New Player
```

If no value is specified, all elements are set to empty strings.

```
40 DIM SCORES(4)
50 PRINT ALEN(SCORES) :' Prints 4
60 PRINT SCORES(1) :' Prints nothing
```

DIM is optional: it is possible to treat any initialised variable as an array even if it has not been dimensioned, as arrays are actually simply strings split into elements by backtick (`) separators. [SEE 3.2.5 ARRAYS]

```
70 X = "Value 1"
80 X(5) = "Value 5"
90 PRINT X :' Prints Value 1````Value 5
```

DIM is supported because it is a useful away to initialise all the elements of an array in a single statement, and for compatibility with other dialects of Basic.

3.5.7 END / STOP

Both END and STOP commands stop program execution. Both are supported for compatibility with other dialects of Basic.

```
10 INPUT "Want to end the program?", ANS AS "U,Y|N"
20 IF ANS = "Y" THEN END
30 INPUT "How about now?", ANS AS "U,Y|N"
40 IF ANS = "Y" THEN STOP
50 PRINT "Ok, still running!"
```

3.5.8 FOR variable = startvalue TO finalvalue STEP increment / EXIT condition / NEXT

A FOR statement introduces a block of statements which must be executed repeatedly.

```
10 PRINT "About to print 123... GO!"
20 FOR X = 1 TO 3
30 PRINT X;
40 NEXT
50 PRINT "... GO!"
```

In the example above, the FOR statement on line 20 assigns the {startvalue} of 1 to the {variable} X.

X is then checked to see if it has exceeded the {finalvalue} of 3. It hasn't, and execution passes to line 30, where X is printed. Finally execution reaches line 40 - the NEXT statement - which returns control to line 20 - the FOR statement.

X is then incremented by 1, raising it to 2, and X is printed again. Then X is incremented to 3, and printed again. Finally, the FOR statement increments X to 4. This exceeds the {finalvalue} of 3, and so execution jumps to line 50 - the line after the NEXT statement.

You can specify a STEP clause to add a different {increment} to the value of the {variable} for each loop. If the increment is negative, the value will fall during each loop.

```
10 PRINT "About to print 6420!"
20 FOR Y = 6 TO 0 STEP -2
30 PRINT Y;
40 NEXT
50 PRINT "!"
```

FOR loops can also be nested.

```
10 PRINT "All possible 2x4-sided dice rolls"
20 FOR DIE1 = 1 TO 4
30 FOR DIE2 = 1 TO 4
40 PRINT DIE1; "+"; DIE2; "="; DIE1+DIE2,
50 NEXT
60 PRINT
70 NEXT
```

This example prints:

```
1+1=2   1+2=3   1+3=4   1+4=5
2+1=3   2+2=4   2+3=5   2+4=6
3+1=4   3+2=5   3+3=6   3+4=7
4+1=5   4+2=6   4+3=7   4+4=8
```

It is a bad idea to jump into the middle of a FOR/NEXT loop (using a statement like GOTO). If the computer encounters a NEXT statement without having previously passed through a FOR statement, it will stop the program with an error message of "NEXT without FOR".

Also, it is better to avoid jumping out of a FOR/NEXT loop before it has finished. It is less likely to cause an immediate error, but the computer is left "waiting" for another "NEXT" statement. So if you need to exit a loop before it finishes naturally, use EXIT.

EXIT immediately ends the loop, and continues execution from the line after NEXT.

```
10 FOR X = 1 TO 10
20 PRINT "X is "; X
30 INPUT "Continue? (Y/N): ", CONT AS "U,T,Y|N"
40 IF CONT # "Y" THEN EXIT
50 PRINT "Ok, continuing."
60 NEXT
70 PRINT "DONE"
```

Sometimes, a FOR/NEXT loop with an EXIT may be more elegantly written as a LOOP/REPEAT loop. [SEE 3.5.19 LOOP]

Bear in mind that if {startvalue} is greater than {finalvalue}, and the {increment} is not negative, the statements inside the loop will not be executed at all: program will be transferred immediately to the statement following the NEXT statement. The same will happen if the {incremenent} is negative, but the {startvalue} is less than the {finalvalue}.

```
10 FOR COUNTER = 1 TO 3 STEP -1
20 PRINT "This will never print."
30 NEXT
40 PRINT "The PRINT above won't execute."
```

Note that many dialects of Basic required the name of the FOR variable to be repeated after word NEXT. Bright Basic does not require this, but - for the sake of compatibility with existing Basic programs - it is allowed, and ignored.

```
10 FOR POTATO = 1 TO 3
20 PRINT POTATO; " potato"
30 NEXT APPLE :' Variable name ignored, even if wrong
```

3.5.9 GET variable

The GET statement pauses the program until a key on the keyboard has been clicked, and then assigns the name of the key clicked to a {variable}.

The key names assigned by GET are the same as those returned by the INKEY function. [SEE 3.4.13 INKEY]

```
10 PRINT "The program has started."
20 PRINT "Press any key to continue."
30 GET KEY
40 PRINT "You pressed ["; KEY; "]"
50 PRINT "Pausing output is a common use of GET."
```

Note that GET responds only to single clicks, and does not echo the key pressed on the monitor.

To collect an entire string entered using multiple keyboard clicks or via through chat, and echo the string to the monitor, use INPUT. [SEE 3.5.13 INPUT]

3.5.10 GOTO linenum

The GOTO statement starts or continues running a program from a specified line.

```
10 PRINT "Hello World!"
20 GOTO 10 :' Program loops endlessly
```

If GOTO references a line number which doesn't exist, program execution will continue from the first line number with a greater value than the one specified. (Except for GOTO 0, which is simply ignored.)

```
10 REM Note there is no line 15
20 PRINT "LINE TWENTY"
30 GOTO 15 :' No line 15, so goes to 20 & loops
```

The target may be specified using a line number (as above), or a "label", established using the LABEL command, or its abbreviated form, @.

```
10 @START :' Equivalent to LABEL START
20 PRINT "Hello World!"
30 GOTO START :' Loops forever
```

Bright Basic supports "computed GOTO", which means that the target line number may be specified using an expression.

```
10 INPUT "Enter a number from 1 to 3: ", NUMBER AS "I,1:3"
20 GOTO 20 + NUMBER * 10 :' Computed numeric goto
30 PRINT "YOU ENTERED 1" : STOP
```

```
40 PRINT "YOU ENTERED 2" : STOP
50 PRINT "YOU ENTERED 3" : STOP
```

Expressions can also return label names.

```
10 INPUT "Enter a number from 1 to 3: ", NUMBER AS "I,1:3"
20 GOTO "LABEL" ~ NUMBER :' Computed label goto
30 @LABEL1 : PRINT "YOU ENTERED 1" : STOP
40 @LABEL2 : PRINT "YOU ENTERED 2" : STOP
50 @LABEL3 : PRINT "YOU ENTERED 3" : STOP
```

3.5.11 GOSUB linenum / RETURN

GOSUB branches execution to the specified {linenum} exactly as GOTO does. [SEE 3.5.10 GOTO]

The only difference between the two is that GOSUB is designed to execute a "subroutine" (hence "GOSUB" or "go to subroutine") which begins at {linenum}, and is executed until a RETURN statement is reached, at which point control transfers back to the line immediately following the GOSUB.

```
10 X = 1 : GOSUB 50
20 X = 32 : GOSUB 50
30 X = 97 : GOSUB 50
40 END
50 PRINT "The number after "; X; " is "; X+1
60 RETURN
```

GOSUB therefore allows you to write subroutines, or blocks of code which are reused at different points throughout your program.

3.5.12 IF condition THEN / ELSEIF / ELSE / ENDIF

The IF statement branches the execution of a program depending on a {condition}, executing the "THEN clause" if the {condition} is "true", or the "ELSE clause" if it is "false". (ELSE is optional: if the condition is "false", and there is no ELSE, the IF statement does nothing.)

Different dialects of Basic implement the THEN and ELSE clauses in different ways. Bright Basic supports the three most common syntaxes.

1. A THEN or ELSE may simply be followed by a line number. The clause then acts like a GOTO statement [SEE 3.5.10 GOTO], branching execution to the specified line.

```
10 IF 1=1 THEN 20 ELSE 30
20 PRINT "True!" : STOP
30 PRINT "False!" : STOP
```

2. A THEN or ELSE may be followed by a complete Basic statement.

```
10 IF 1=1 THEN PRINT "True" ELSE PRINT "False"
```

3. The IF statement may have nothing after THEN. In this case, the block of lines following the IF are made conditional: if the {condition} is false, execution jumps to the next ELSEIF, ELSE, or ENDIF statement. ELSEIF statements have the same syntax as IF statements, but their blocks are only executed if the IF and ELSEIF blocks above them were not. If none of the IF or ELSEIF blocks are executed, the ELSE block - if there is one - is executed.

```
10 INPUT "Enter a number: ", NUM
20 IF NUM = 1 THEN
30 PRINT "You entered 1."
40 PRINT "That's an odd number."
50 ELSEIF NUM = 2 THEN
60 PRINT "You entered 2."
70 PRINT "That's an even number."
80 ELSE
90 PRINT "You didn't enter 1 or 2."
100 ENDIF
```

3.5.13 INPUT prompt, variable[;] AS pattern

The INPUT statement displays a {prompt} inviting user to enter a value, waits for them to do so, converts and validates the value entered according to the input {pattern}, displays it on the screen after the prompt, and assigns it to {variable}.

All the parameters except {variable} are optional. In its simplest form, INPUT simply pauses the program, collects input, and assigns it {variable}.

```
10 INPUT X :' Waits for input & assigns it to X
20 PRINT "X is: "; X :' Prints X is (value input)
```

If a {prompt} is specified, it is displayed to let the user know what they should enter.

```
10 INPUT "Enter your name: ", NAME
20 PRINT "Hello "; NAME; "!"
```

When run, and a name entered, the monitor displays:

```
>RUN
Enter your name: JOHN
Hello JOHN!
>
```

As happened in the example above, when the entered value is echoed on the monitor, subsequent output will normally begin at the beginning of the next line. To prevent this, and allow output to continue immediately after the echoed input and on the same line, add a semicolon, ";", after the variable name.

```
10 INPUT "What is your name? ", NAME;
20 PRINT " <-- Nice name!"
```

Note the semicolon after NAME. Running this program will display this output.

```
>RUN
What is your name? JOHN <-- Nice name!
>
```

The {pattern} is a list of codes which will apply conversion and validation to the value entered. The conversion codes are:

U - Upcase entry
D - Downcase entry
T - Trim entry
/n - Cut entry to n characters

The conversion codes above are applied before the validation codes are checked. The validation codes are:

I - Must be integer
N - Must be number (integer or real)
min:max - Must be min <= entry <= max
a|b|c... - May be any of these values

This example will accept only integers between 1 and 100.

```
10 INPUT "Enter your age (1-100): ", AGE AS "I,1:100"
```

This statement will accept any number, integer or real.

```
20 INPUT "Enter your height in metres: ", HEIGHT AS "N"
```

This example will upcase the entry made, trim it, cut it to one character long, and then ensure to see if it matches Y or N.

```
30 INPUT "Continue? (Y/N): ", CONTINUE AS "U,T,/1,Y|N"
```

If you combine I, N, or min:max with a list a|b|c..., the values a, b, and c are "whitelisted" - they will be valid even if they don't meet the other critera. Conversely, input matching the I, N, and min:max criteria need not appear in the whitelist.

```
40 INPUT "Enter 1-20 (or Q to quit): ", CHOICE AS
"I,1:20,U,Q|Q"
```

This will accept any integer between 1 and 20, or Q. Note the use of Q|Q: this trick allows a single value to be whitelisted.

If a user enters a value which doesn't meet the specified criteria, the entry will be erased from the monitor, they will hear a low beep, and they will receive the message in chat: "Sorry, invalid input. Please try again."

The AS clause was not taken from any existing Basic. It was added to Bright Basic to make input easier to manage, as so much code in traditional Basic programs was dedicated to validating input. But because it is an optional clause, it doesn't break backwards compatibility with old code: rather than adding an AS clause to an input statement, you may do the validation using separate IF statements, displaying error messages on the screen.

```
10 INPUT "Option 1 or 2? ", OPTION
20 IF OPTION = 1 | OPTION = 2 THEN 60
```

```
30 BEEP 4 :' Low beep, same as INPUT error beep
40 PRINT "Please enter 1 or 2."
50 GOTO 10
60 PRINT "You chose "; OPTION
```

BEEP 4 produces the same tone as an input error. [See 3.5.1 BEEP]

3.5.14 LABEL/@ labelname

The LABEL statement establishes a target to which GOTO, GOSUB, and IF statements may branch, as an alternative to branching to a line numnber. It may be abbreviated to an at symbol (@).

```
10 @START :' Equivalent to LABEL START
20 PRINT "Hello World!"
30 GOTO START :' Loops forever
```

3.5.15 LET variable = value

Assign a value to a variable.

```
10 LET COMP = "Bright Basic Computer"
20 LET COMPVERSION = 1
30 LET OTHERCOMPS(1) = "Commodore PET"
40 LET OTHERCOMPS(2) = "Sinclair ZX81"
```

As in most Basic dialects, the LET keyword is optional. A variable (or indexed array element) may simply be named before an = sign to be assigned the value which follows.

```
10 COMP = "Bright Basic Computer"
20 COMPVERSION = 1
30 OTHERCOMPS(1) = "Commodore PET"
40 OTHERCOMPS(2) = "Sinclair ZX81"
```

The operators +=. -=, *=, /=, #=, ^=, &=, |= and `= may be used in place of = to allow an operation to be combined with an assignment.

```
10 A = 1
20 A += 9 :' Equivalent to A = A + 9
30 PRINT A :' Prints 10
40 A -= 2 :' Equivalent to A = A - 2
50 PRINT A :' Prints 8
```

```
60 A *= 5 :' Equivalent to A = A * 5
70 PRINT A :' Prints 40
80 A /= 4 :' Equivalent to A = A / 4
90 PRINT A :' Prints 10
100 A ~= "0" :' Equivalent to A = A ~ "0"
110 PRINT A :' Prints 100
120 A ^= 0.5 :' Equivalent to A = A ^ 0.5
130 PRINT A :' Prints 10
140 A &= 0 :' Equivalent to A = A & 0
150 PRINT A :' Prints 0
160 A |= 1 :' Equivalent to A = A | 1
170 PRINT A :' Prints 1
180 A `= 1 :' Equivalent to A = A ` 1
190 PRINT A :' Prints 1`1
```

3.5.16 LIST/LLIST linenumber / from, to

List lines of the BASIC program in memory.

The LIST command lists the lines on your monitor, while LLIST lists them in chat, allowing you to cut and paste them from your chat window, or chat.txt log. (In early Basic dialects, the extra initial "L" stood for "line printer", so LLIST meant "list on the line printer". BBC Basic has no access to a printer, so it sends this output to your chat window instead.)

LIST alone lists the entire program.

```
>LIST
10 REM Simple program
20 PRINT "Hello World!"
30 GOTO 20
3 line(s) listed.
>
```

If a single {linenumber} is specified, just that line (if it exists) is listed.

```
>LIST 20
20 PRINT "Hello World!"
1 line(s) listed.
>
```

If a comma is used, the line to list {from} appears before it, and the line to list {to} appears after. Note that neither {from} nor {to} need refer to existing lines - any lines in the range will be listed.

If a comma is used, and {from} is omitted, the program is listed from the first line. Similarly, if {to} is omitted, it is listed to the end.

When you use LLIST, your monitor will display a "*" as each line is listed.

```
>LLIST
***
3 line(s) listed.
>
```

You will hear the following in chat:

```
[11:54] Bright Basic Computer whispers:
10 PRINT "First line"
20 PRINT "Second line"
30 PRINT "Third line"
```

LLIST is particularly useful if you wish to save a program which you have entered or changed on the computer back into a notecard. Use LLIST to list the program into your chat log, and then cut and paste the text either from your viewer's chat window, or from the chat.txt log file Second Life saves on your computer.

As Second Life limits the length of chat text messages to 1024 bytes, you will find the listing is broken up into sections, and every forty lines or so will contain a line like "[11:54] Bright Basic Computer whispers:". You may naturally wish to cut these out of your listing, but if left in the notecard they will do no harm, as for precisely this reason, the LOAD command ignores any line beginning with an opening bracket "[". [SEE 3.5.18 LOAD]

LLIST chats in a Second Life "whisper", meaning that it can only be heard within 10m of the computer.

3.5.17 LISTV from, to

List the values of variables in memory, in alphabetical order.

LISTV on its own lists all variables in memory.

```
>LISTV
COMPUTER = "BBC"
LANG = "Bright Basic"
VERSION = 1
3 variable(s) listed.
>
```

Specifying LISTV {from}, {to} restricts the list to variable names which fall alphabetically between {from} and {to}. Omitting "from" (before the comma) lists from the first variable, omitting "to" (after the comma) lists to the last, and omitting both (and the comma) lists the all variables.

```
>LISTV M,
VERSION = 1
1 variable(s) listed.
>
```

3.5.18 LOAD notecard

The LOAD command first clears the computer's memory (like NEW [SEE 3.5.20 NEW]), loads into memory a Basic program from the specified {notecard} in the computer's inventory, and runs it.

```
>LOAD "MYPROGRAM"
Loading "MYPROGRAM", 2 lines...
Autorunning program...
*** This is my program. ***
It was saved in a notecard.
>
```

You can allow the program to complete, or click "break" to stop it running, and use LIST to see the lines of Basic.

```
>LIST
10 PRINT "*** This is my program. ***"
20 PRINT "It was saved in a notecard."
>
```

The name of the {notecard} may be up to 12 characters long.

Include a question mark ("?") in the name of a notecard if you do not wish the program it contains to be run automatically when loaded. This can be useful for storing half-finished programs which you are editing and reloading frequently after small changes, and don't necessarily wish to restart them each time.

```
>LOAD "NEWPROG?"
Loading "NEWPROG?", 3 lines...
Loading complete.
>
```

(Notecards with a question mark in their name will also be excluded from the menu of programs displayed when the "load" key is clicked.)

If a line at the top of the notecard says {AUTONUM}, then the lines of Basic in the notecard need not have line numbers: they will be numbered automatically as they are loaded.

Blank lines are ignored. Lines which begin with an opening bracket "[" are also ignored, in case the text in the notecard has been cut from a program echoed in the chat log produced by the LLIST command [SEE 3.5.16 LIST], and lines of chat ("[10:27] Jane Doe: Hi there!") have been accidentally included.

Normally, you will manually type in LOAD commands to load a notecard, but you can use a LOAD command within a program to load a different program.

```
10 PRINT "Which game do you wish to run?"
20 PRINT "1. Hangman"
30 PRINT "2. Noughts & Crosses/Tic-Tac-Toe"
40 PRINT "3. Global Thermonuclear War"
50 INPUT "Please enter (1-3): ", CHOICE AS "I,1:3"
60 IF CHOICE = 1 THEN LOAD "HANGMAN"
70 IF CHOICE = 2 THEN LOAD "OXO"
80 IF CHOICE = 3 THEN LOAD "WAR"
```

Remember that as soon as the LOAD command is executed, everything in the computer's memory will be cleared, including this program, before the chosen program is loaded. It is possible to use this technique to create menus, as done above, or even to break a very long program into separate

parts. This was a particularly useful technique on 70s/80s computers with little RAM, and was called "chaining".

[SEE 1.5 USING PROGRAM NOTECARDS] for a detailed discussion of the use of notecards to write programs, including the LOAD command.

3.5.19 LOOP / WHILE/UNTIL condition / REPEAT

Statements between a LOOP and a REPEAT will be executed, repeatedly, forever.

```
10 LOOP
20 PRINT "Hello World!"
30 REPEAT
```

Usually, you will wish to end your loop under specific conditions. You can specify a condition using a WHILE or an UNTIL statement anywhere between the LOOP and the REPEAT.

```
10 LOOP
20 PRINT "Die roll: "; 1 + INT(RND(0) * 6)
30 INPUT "Roll again? (Y/N) ", AGAIN AS "U,T,Y|N"
40 UNTIL AGAIN = "N"
50 PRINT "Ok, rolling again..."
60 REPEAT
70 PRINT "Finished rolling dice."
```

In this example, the loop will continue until the user enters N. The UNTIL condition at line 40 will then end the loop, and jump execution to line 70 - the line immediately after the REPEAT.

"UNTIL condition" ends a loop when the condition is true. "WHILE condition" ends it when the condition is false. They are otherwise identical. Use whichever makes the condition easier to read and understand.

Other programming languages, including some modern Basics, offer "WHILE...WEND" loops, which check a condition at the top, and "DO UNTIL" loops, which check a condition at the bottom. A LOOP/REPEAT loop can be used to recreate the effect of either, with a WHILE condition immediately after LOOP for the former, or an UNTIL immediately before REPEAT for the latter. It can also allow you to exit loops from the middle.

Note that, depending on the condition you set, the statements within a LOOP/REPEAT may not be executed at all. In the following loop, program execution skips the loop entirely and jumps to the line after REPEAT.

```
10 LOOP
20 WHILE 1=2
30 PRINT "1 doesn't equal 2, so"
40 PRINT "this will not be printed."
50 REPEAT
60 PRINT "This will be printed."
```

LOOP/REPEAT loops may be nested.

```
10 LOOP
20 COUNTER = 10
30 LOOP
40 WHILE COUNTER > 0
50 PRINT COUNTER; "-";
60 COUNTER -= 1
70 REPEAT
80 PRINT "Lift Off!"
90 INPUT "Count down again?", CONTINUE AS "U,T,Y|N"
100 UNTIL CONTINUE = "N"
110 PRINT "Ok, next count down..."
120 REPEAT
```

3.5.20 NEW

Clears the computer memory, erasing all numbered Basic command lines, and clearing all variables.

To delete the Basic program without clearing the variables, use DELETE. [SEE 3.5.5 DELETE]

To clear the programs without deleting the program, use CLEAR. [SEE 3.5.2 CLEAR]

3.5.21 ON number GOTO/GOSUB linenumber1, linenumber2...

This command works similarly to a GOTO or a GOSUB, except that rather than always transferring execution to a single line number, it chooses from

a list of line numbers on the basis of the specified {number} parameter. If {number} is 1, it chooses the first line number, if 2, the second, and so on.

```
10 INPUT "Enter a number (1-3): ", X
20 ON X GOTO 40, 50, 60
30 PRINT "You didn't enter 1-3." : GOTO 10
40 PRINT "You entered 1." : STOP
50 PRINT "You entered 2." : STOP
60 PRINT "You entered 3." : STOP
```

Note that if the {number} value is less than 1, or greater than the number of line numbers specified, the ON command will be ignored and execution will pass normally to the next line.

The ON command may also be used to transfer execution to a labelled line.

```
10 INPUT "Enter a number (1-3): ", X
20 ON X GOTO LABEL1, LABEL2, LABEL3
30 PRINT "You didn't enter 1-3." : GOTO 10
40 @LABEL1 : PRINT "You entered 1." : STOP
50 @LABEL2 : PRINT "You entered 2." : STOP
60 @LABEL3 : PRINT "You entered 3." : STOP
```

3.5.22 PLAY/PLAYLOOP sounditem/uuid, volume / PLAYSTOP

The PLAY command plays a Second Life sound, identified by the name of a {sounditem} in the computer's inventory, or a sound {uuid}.

The sound is played at a {volume} between 0 (silent) and 1 (loud). If {volume} is omitted, 1 is assumed.

```
10 REM Play PACMAN sound by UUID & by name
20 PLAY "fe9ee872-c6a5-9172-9a6d-356ad8fbd474" : WAIT 2
30 PLAY "PACMAN", 0.5 :' Only 50% volume : WAIT 2
```

The sound will play until it is complete, or the next PLAY command, or - if it is being played by a program - the program ends.

PLAYLOOP will play the sound on a loop, which will run until the next PLAY command, or the program ends.

PLAYSTOP will stop any sound currently playing.

Specifying a {sounditem} which is not in your computer's inventory, or an invalid sound {uuid}, will not cause an error - but no sound will be heard.

3.5.23 PRINT/?/LPRINT textvalues

PRINT (or its abbreviation "?") displays a series of values on the monitor screen.

LPRINT accepts the same text values, but instead of displaying them on your monitor, it prints them in chat. (In early Basic dialects, the extra initial "L" stood for "line printer", so LPRINT meant "print on the line printer". BBC Basic sends this output to your chat window instead.)

As implemented on the BBC, LPRINT can be particularly useful while debugging a program, allowing you to LPRINT the values of variables while a program runs, without interrupting the programs usual monitor output.

The values to be displayed by PRINT or LPRINT may be separated using semicolons (;), commas (,), or underscores (_).

If separated by a semicolon, a value is printed immediately after the last. If separated by a comma, it will appear at the next tab position: tab positions are 8 characters apart. If separated by an underscore, it will appear on the next row.

```
>PRINT 0;1,2;3,4;5_6;7,8;9
01      23      45
67      89
```

If a PRINT statement ends in a semicolon, the next PRINT statement will begin printing immediately after the value printed. If it ends in a comma, the next PRINT statement will begin at the next tab position. Otherwise, the next PRINT statement will begin at the beginning of the next row.

```
10 PRINT "A";
20 PRINT "a" :' Prints Aa
30 PRINT "B",
40 PRINT "b" :' Prints B       b
50 PRINT "C"
60 PRINT "c" :' Prints C (next row) c
```

As in many BASIC dialects, the PRINT command can be abbreviated to a ?, and even the space between ? and the expression printed may be omitted. This can be handy in direct mode to do a quick calculation.

```
>?2^16
65536
```

Various special character sequences, called "tokens", may be embedded in the values printed to control where they will appear on the monitor screen, and in what colour and style.

The colour tokens are: {W} White, {R} Red, {G} Green, {B} Blue, {C} Cyan, {M} Magenta, and {Y}ellow.

Tokens {S+} and {S-} enable and disable "strong" or bold text, and {I+} and {I-} enable and display inverse text - black text on a coloured background.

The token {O} selects the monitors "ordinary" or default text colour, chosen through the monitor's setting menu. [SEE 2.2 MONITOR SETTINGS]

And the token {P} sets the ordinary or default text colour (like {O}), and also cancels bold and inverse text.

```
10 REM Print different colours...
20 PRINT "{R}red {C}cyan {O}ordinary"
30 REM ...and different styles...
50 PRINT "{I+}inverse {I-} not inverse"
60 PRINT "{S+}strong {S-} not strong"
```

Rather than embedding these colour sequences in literal strings, they can also be generated by the COLOUR function. [SEE 3.4.8 COLOUR]

Tokens can also used to position text.

The token {Dn} will position text in the first column of the nth rows down, and {An} will position it at the nth column across.

```
10 CLS
20 PRINT "EMBEDDED COORDINATE EXAMPLES"
30 PRINT "{A7}X marks column 10 on current row"
```

```
40 PRINT "{D4}X marks beginning of fourth row down"
50 PRINT "{D12}{A5}X marks column 5, row 12"
```

As an alternative to embedding these tokens, they can be generated using the AT function. [SEE 3.4.5 AT]

A few other tokens are available:

{E} clears or "erases" the screen.

{N} embeds a "newline", to start text on the next row down.

{T} prints from the next tab position. Tab positions are 8 characters apart, in columns 1, 9, 17, 25, and 33 of the 40 column screen.

{Jn} "jumps" n (between -40 and 40) columns from the current column: back if negative, forward if positive. (If n is greater than 40, or less than 40, this token is ignored.)

```
10 PRINT "{E}Cleared or 'erased' screen"
20 PRINT "This line...{N}...the next line"
30 PRINT "Before tab{T}After tab"
40 PRINT "TOP SECRET{J-10}XXXXXXXXXX"
```

If you wish to print an actual brace symbol without having it interpreted as part of one of these tokens, use {<} for {, or {>} for }.

```
>PRINT "These are just {<}braces{>}."
These are just {braces}.
>
```

3.5.24 REM/' comment

Does nothing, except allowing a remark to be recorded within a Basic program.

```
10 REM *** REM DEMO PROGRAM ***
20 REM Remark lines are ignored.
30 PRINT "1st line which does anything."
```

The apostrophe, ', may be used instead of REM to introduce a remark.

```
40 ' Also a remark, also ignored.
```

Because a single line of Bright Basic can be divded into separate commands using colons (:), it is possible to add comments to the end of a line.

```
50 PRINT "Hello!" :' Be friendly!
```

However, bear in mind that you cannot put a : in a comment, as it will be treated as a command separator.

```
60 REM Programmer: Shan Bright
```

This line will be treated as an error, generating the message "Unknown command", because Bright Basic will treat "Shan Bright" as a separate command, and there is no "SHAN" command in the language.

If this happens, just change the : to another symbol - a ; or a -, for instance:

```
70 REM Programmer - Shan Bright
```

3.5.25 RUN linenumber

RUN will:

1. Erase all variable values (like CLEAR) [SEE 3.5.2 CLEAR]

2. Reset the DATA pointer (like RESTORE) [SEE 3.5.4 DATA]

3. Branch to {linenumber} (like GOTO) [SEE 3.5.10 GOTO]

If {linenumber} is not specified, RUN will start from the first line of Basic.

RUN is most commonly typed in at the command prompt to start a program, but it can be used within a program, to CLEAR variables, RESTORE the data pointer, and GOTO a line (or the beginning of a program), all in one line.

```
10 A = "THIS IS A!"
20 PRINT "About to print A."
30 PRINT A
40 PRINT "Just printed A."
50 IF A = "" THEN STOP
60 RUN 20
```

This program will product the following output:

```
>RUN
About to print A.
THIS IS A!
Just printed A.
About to print A.

Just printed A.
>
```

Note that the second time A was printed, it had no value, as RUN had cleared it.

3.5.26 TRON/TROFF

TRON and TROFF switch tracing on and off. While tracing is on, the line number of each Basic command executed is "whispered" into local chat. This can be useful while developing and debugging programs, as it allows a programmer to follow the path of the code being executed in real time.

Sometimes, you may simply wish to enter the TRON command to enable tracing before you use the RUN command to test the program, and every single line number will be traced.

However, as the computer may execute many lines a second, the output can be overwhelming. If you are trying to debug a particular part of your code, you can add TRON and TROFF statements to the code itself, so that it is enabled only while the lines you are particularly intered in are running.

```
10 TRON :' Switches on tracing
20 PRINT "Line 20 will be traced."
30 PRINT "Line 30 will be traced."
40 TROFF
50 PRINT "Line 50 will not be traced."
```

You can then delete the TRON and TROFF lines once you have finished working on the code in question.

3.5.27 WAIT seconds

The WAIT command will pause execution of the program for the specified number of seconds.

```
10 PRINT "Wait 3 seconds..."
20 WAIT 3
30 PRINT "...and relax"
```

Many Basic dialects offer variants of this command, such as PAUSE or SLEEP. More awkwardly, some offer WAIT, but measured in various different units, such as Acorn's Basic WAIT statement, which was measured in hundredths of a second. WAIT therefore waits for seconds, but allows fractions of a second to be specified by using a real number {seconds} parameter.

```
10 WAIT 0.5 :' Equivalent to WAIT 50 in Acorn Basic
```

3.6 APPENDICES

This section provides a few reference tables which may be useful while to Basic programmers.

3.6.1 SYMBOLS

Bright Basic uses the 95 characters of the printable ASCII character set.

The upper and lower case alphabets form 52 of these characters, and the digits a number 10. These leave 33 characters:

space -> Basic keywords separator
! -> Logical NOT operator
" -> String delimiter ("Hello World")
-> Alternative to <>
$ -> Optional variable name ending (X$)
% -> Optional variable name ending (X%)
& -> Logical AND operator
' -> REM abbreviation
(-> Precedence/function/array syntax
) -> Precedence/function/array syntax

* -> Multiplication operator
+ -> Addition operator
, -> Values separator (tab in PRINT)
- -> Negation/subtraction operator
. -> Decimal numbers
/ -> Division operator
: -> Statement separator
; -> PRINT item continuation separator
< -> Less than operator
= -> Equality operator
> -> Greater than operator
? -> PRINT abbreviation
@ -> Label prefix
[-> Alternative to <=
\ -> Integer division operator
] -> Alternative to >=
^ -> Exponential operator
_ -> PRINT item newline separator
` -> Array operator/separator
{ -> PRINT/notecard tokens
| -> Logical OR operator
} -> PRINT/notecard tokens
~ -> Concatenation operator

3.6.2 PRINT TOKENS

The following character sequences or "tokens" may be embedded into strings displayed on the computer monitor using the PRINT statement:

{W} White
{R} Red
{G} Green
{B} Blue
{C} Cyan
{M} Magenta
{Y} Yellow
{S+} "Strong" or bold text
{S-} Strong off

{I+} Inverse text
{I-} Inverse off
{O} "Ordinary" text colour (monitor default)
{P} "Plain" - ordinary colour, not bold/inverse
{Dn} "Down" to row n
{An} "Across" to column n
{Jn} "Jump" n (<=40) characters forward or back
{N} Newline
{T} Tab
{E} "Erase" or clear screen
{<} Literal { character
{>} Literal } character

3.6.3 INPUT PATTERNS

The following codes may be included in the AS clause of an INPUT statement to convert the value entered.

U -> Upcase entry
D -> Downcase entry
T -> Trim whitespace around entry
/n -> Cut entry to n characters

Once any of these conversion codes have been applied, the resulting input string can be validated using these codes:

I ->Must be an integer
N -> Must be a number (integer or real)
min:max -> Must be min <= entry <= max
a|b|c... -> Must be a or b or c...

Note that if a|b|c validation is combined with I, N, or min:max, then inputs a, b, and c are accepted even if they do not match the other criteria, and conversely inputs matching the other criteria are accepted even if they are not a, b, or c.

3.6.4 KEY NAMES (INKEY/GET)

The keyboard has 58 keys. The four red keys "help", "load", "settings", and "break" have special functions independent of the program running, and are ignored by both the INKEY function and the GET statement.

For the remaining 54 keys, the names returned by INKEY, or assigned to a variable by GET, are as follows:

26 alphabetic keys: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

10 numeric keys: 0 1 2 3 4 5 6 7 8 9

11 punctuation keys: ~ - = ' [ ] \ ; . , /

Space: a single space character

6 editing keys: ERASE, TAB, SHIFTLOCK, ENTER, LEFTSHIFT, and RIGHTSHIFT