# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction and Overview</td>
<td></td>
</tr>
<tr>
<td>Introduction to BBC BASIC</td>
<td>18</td>
</tr>
<tr>
<td>Overview</td>
<td>20</td>
</tr>
<tr>
<td>Running BBC BASIC</td>
<td>21</td>
</tr>
<tr>
<td>Minimum system requirements</td>
<td>22</td>
</tr>
<tr>
<td>Specification</td>
<td>23</td>
</tr>
<tr>
<td>Example programs</td>
<td>25</td>
</tr>
<tr>
<td>Differences from BBC BASIC (86)</td>
<td>30</td>
</tr>
<tr>
<td>Version 6 changes</td>
<td>32</td>
</tr>
<tr>
<td>Notices</td>
<td>36</td>
</tr>
<tr>
<td>Command and Editing Window</td>
<td></td>
</tr>
<tr>
<td>Introduction to the IDE</td>
<td>37</td>
</tr>
<tr>
<td>The title bar</td>
<td>38</td>
</tr>
<tr>
<td>The menu bar</td>
<td>39</td>
</tr>
<tr>
<td>The toolbar</td>
<td>40</td>
</tr>
<tr>
<td>The status bar</td>
<td>41</td>
</tr>
<tr>
<td>File menu</td>
<td>43</td>
</tr>
<tr>
<td>Edit menu</td>
<td>48</td>
</tr>
<tr>
<td>Utilities menu</td>
<td>53</td>
</tr>
<tr>
<td>Options menu</td>
<td>66</td>
</tr>
<tr>
<td>Run menu</td>
<td>72</td>
</tr>
<tr>
<td>Help menu</td>
<td>76</td>
</tr>
<tr>
<td>The editing pane</td>
<td>78</td>
</tr>
<tr>
<td>Registry settings</td>
<td>85</td>
</tr>
<tr>
<td>General Information</td>
<td></td>
</tr>
<tr>
<td>Line numbers</td>
<td>86</td>
</tr>
<tr>
<td>Statement separators</td>
<td>87</td>
</tr>
<tr>
<td>Line continuation</td>
<td>88</td>
</tr>
</tbody>
</table>
Expression priority 89
Array arithmetic 90
Initialising arrays 92
Variables 93
Structures 103
Program flow control 111
Indirection 115
Operators and special symbols 121
Keywords 124
Debugging 126
Error handling 130
Procedures and functions 139
Input editing 150
Copy key editing 152
Hardcopy output to a printer 154
Labelling program lines 155
Multilingual (Unicode) text output 156

Graphics and Colours
  Introduction to graphics 157
  Display modes 158
  Colours 162
  Drawing on the screen 168
  Text and graphics viewports 182
  Writing text to the screen 186
  Capturing the screen contents 189
  Pixel-perfect graphics 190
  Compatibility limitations 194

Keywords
  Introduction to keywords 197
  ABS 199
  ACS 201
<table>
<thead>
<tr>
<th>Keyword</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENDPDRC</td>
<td>253</td>
</tr>
<tr>
<td>ENDWHILE</td>
<td>254</td>
</tr>
<tr>
<td>ENVELOPE</td>
<td>255</td>
</tr>
<tr>
<td>EOF#</td>
<td>257</td>
</tr>
<tr>
<td>EOR</td>
<td>258</td>
</tr>
<tr>
<td>ERL</td>
<td>260</td>
</tr>
<tr>
<td>ERR</td>
<td>261</td>
</tr>
<tr>
<td>ERROR</td>
<td>262</td>
</tr>
<tr>
<td>EVAL</td>
<td>264</td>
</tr>
<tr>
<td>EXIT</td>
<td>266</td>
</tr>
<tr>
<td>EXP</td>
<td>268</td>
</tr>
<tr>
<td>EXT#</td>
<td>269</td>
</tr>
<tr>
<td>FALSE</td>
<td>271</td>
</tr>
<tr>
<td>FILL</td>
<td>273</td>
</tr>
<tr>
<td>FN</td>
<td>274</td>
</tr>
<tr>
<td>FOR</td>
<td>276</td>
</tr>
<tr>
<td>GCOL</td>
<td>278</td>
</tr>
<tr>
<td>GET/GET$</td>
<td>280</td>
</tr>
<tr>
<td>GOSUB</td>
<td>285</td>
</tr>
<tr>
<td>GOTO</td>
<td>286</td>
</tr>
<tr>
<td>HIMEM</td>
<td>287</td>
</tr>
<tr>
<td>IF</td>
<td>289</td>
</tr>
<tr>
<td>INKEY/INKEY$</td>
<td>292</td>
</tr>
<tr>
<td>INPUT</td>
<td>295</td>
</tr>
<tr>
<td>INPUT LINE</td>
<td>297</td>
</tr>
<tr>
<td>INPUT#</td>
<td>298</td>
</tr>
<tr>
<td>INSTALL</td>
<td>300</td>
</tr>
<tr>
<td>INSTR</td>
<td>302</td>
</tr>
<tr>
<td>INT</td>
<td>303</td>
</tr>
<tr>
<td>LEFT$</td>
<td>305</td>
</tr>
<tr>
<td>LEN</td>
<td>307</td>
</tr>
<tr>
<td>Term</td>
<td>Page</td>
</tr>
<tr>
<td>------------</td>
<td>------</td>
</tr>
<tr>
<td>LET</td>
<td>308</td>
</tr>
<tr>
<td>LINE</td>
<td>309</td>
</tr>
<tr>
<td>LN</td>
<td>310</td>
</tr>
<tr>
<td>LOCAL</td>
<td>311</td>
</tr>
<tr>
<td>LOG</td>
<td>313</td>
</tr>
<tr>
<td>LOMEM</td>
<td>314</td>
</tr>
<tr>
<td>MID$</td>
<td>315</td>
</tr>
<tr>
<td>MOD</td>
<td>317</td>
</tr>
<tr>
<td>MODE</td>
<td>319</td>
</tr>
<tr>
<td>MOUSE</td>
<td>322</td>
</tr>
<tr>
<td>MOVE</td>
<td>325</td>
</tr>
<tr>
<td>NEXT</td>
<td>326</td>
</tr>
<tr>
<td>NOT</td>
<td>327</td>
</tr>
<tr>
<td>OF</td>
<td>329</td>
</tr>
<tr>
<td>OFF</td>
<td>330</td>
</tr>
<tr>
<td>ON</td>
<td>331</td>
</tr>
<tr>
<td>ON CLOSE</td>
<td>333</td>
</tr>
<tr>
<td>ON ERROR</td>
<td>335</td>
</tr>
<tr>
<td>ON ERROR LOCAL</td>
<td>336</td>
</tr>
<tr>
<td>ON MOUSE</td>
<td>338</td>
</tr>
<tr>
<td>ON MOVE</td>
<td>340</td>
</tr>
<tr>
<td>ON SYS</td>
<td>342</td>
</tr>
<tr>
<td>ON TIME</td>
<td>344</td>
</tr>
<tr>
<td>OPENIN</td>
<td>350</td>
</tr>
<tr>
<td>OPENOUT</td>
<td>352</td>
</tr>
<tr>
<td>OPENUP</td>
<td>354</td>
</tr>
<tr>
<td>OPT</td>
<td>355</td>
</tr>
<tr>
<td>OR</td>
<td>356</td>
</tr>
<tr>
<td>ORIGIN</td>
<td>358</td>
</tr>
<tr>
<td>OSCLI</td>
<td>359</td>
</tr>
<tr>
<td>OTHERWISE</td>
<td>360</td>
</tr>
<tr>
<td>Keyword</td>
<td>Page</td>
</tr>
<tr>
<td>----------</td>
<td>------</td>
</tr>
<tr>
<td>PAGE</td>
<td>361</td>
</tr>
<tr>
<td>PI</td>
<td>362</td>
</tr>
<tr>
<td>PLOT</td>
<td>363</td>
</tr>
<tr>
<td>POINT</td>
<td>364</td>
</tr>
<tr>
<td>POS</td>
<td>365</td>
</tr>
<tr>
<td>PRINT</td>
<td>366</td>
</tr>
<tr>
<td>PRINT#</td>
<td>374</td>
</tr>
<tr>
<td>PRIVATE</td>
<td>376</td>
</tr>
<tr>
<td>PROC</td>
<td>378</td>
</tr>
<tr>
<td>PTR#</td>
<td>380</td>
</tr>
<tr>
<td>QUIT</td>
<td>381</td>
</tr>
<tr>
<td>RAD</td>
<td>382</td>
</tr>
<tr>
<td>READ</td>
<td>383</td>
</tr>
<tr>
<td>READ#</td>
<td>384</td>
</tr>
<tr>
<td>RECTANGLE</td>
<td>385</td>
</tr>
<tr>
<td>REM</td>
<td>387</td>
</tr>
<tr>
<td>REPEAT</td>
<td>388</td>
</tr>
<tr>
<td>REPORT/REPORT$</td>
<td>389</td>
</tr>
<tr>
<td>RESTORE</td>
<td>390</td>
</tr>
<tr>
<td>RETURN</td>
<td>392</td>
</tr>
<tr>
<td>RIGHT$</td>
<td>394</td>
</tr>
<tr>
<td>RND</td>
<td>396</td>
</tr>
<tr>
<td>RUN</td>
<td>397</td>
</tr>
<tr>
<td>SGN</td>
<td>398</td>
</tr>
<tr>
<td>SIN</td>
<td>399</td>
</tr>
<tr>
<td>SOUND</td>
<td>400</td>
</tr>
<tr>
<td>SPC</td>
<td>405</td>
</tr>
<tr>
<td>SQR</td>
<td>406</td>
</tr>
<tr>
<td>STEP</td>
<td>407</td>
</tr>
<tr>
<td>STOP</td>
<td>408</td>
</tr>
<tr>
<td>STR$</td>
<td>409</td>
</tr>
</tbody>
</table>
VDU Emulation

Introduction to VDU commands

VDU code summary

VDU 0

VDU 1

VDU 2

VDU 3

VDU 4
Operating System interface

Introduction to star commands

Star command summary
*BYE 487
*CHDIR (*CD) 488
*COPY 489
*DELETE (*DEL) 491
*DIR 492
*DISPLAY 493
*EGA 495
*ERASE (*ERA) 496
*ESC 497
*EXEC 498
*FLOAT 499
*FONT 501
*FX 503
*GSAVE 504
*HARDCOPY 505
*HELP 506
*HEX 507
*INPUT 508
*KEY 509
*LIST 511
*LOAD 512
*LOCK 513
*LOWERCASE 514
*MARGINS 515
*MKDIR (*MD) 516
*MDISPLAY 517
*NOEGA 519
*OUTPUT (*OPT) 520
*PLAY 521
*PRINTER 522
*PRINTERFONT 523
*QUIT 524
*REFRESH 525
*RENAME (*REN) 526
*RMDIR (*RD) 527
*RUN 528
*SAVE 529
*SCREENSAVE 530
*SPOOL 531
*SPOOLON 532
*SYS 533
*TEMPO 535
*TIMER 536
*TV 537
*TYPE 538
*UNLOCK 539
|
Additional OS Interfaces 541
Assembler access to OS routines 543

Disk Files
Introduction to disk files 544
The structure of files 545
Files in BBC BASIC 548
BBC BASIC file access 550
Disk file commands 551
Serial files 558
Random (relative) files 569
Indexed data files 585
The Binary Chop 595

The Assembler
Introduction to the assembler 598
Differences from Intel syntax 600
Numeric and string constants 605
Opcodes 607
Floating-point opcodes 609
MMX opcodes 610
Using BASIC input/output 611
Calling the Windows API 612
Reserving memory 613
The assembly process 616
Conditional assembly and macros 620

Reference information
Table of ASCII codes 622
Mathematical functions 626
Keyword tokens 628

Error Messages and Codes
Editor and compiler errors 635
Run-time errors 642
Access denied 644
Accuracy lost 645
Address out of range 646
Bad call 647
Bad command 648
Bad DIM statement 649
Bad directory 650
Bad device 651
Bad FOR variable 652
Bad hex or binary 653
Bad key 654
Bad library 655
Bad MODE 656
Bad MOUSE variable 657
Bad name 658
Bad program 659
Bad string 660
Bad subscript 661
Bad use of array 662
Bad use of structure 663
Can't match FOR 664
DATA not LOCAL 665
Device fault 666
Device unavailable 667
DIM space 668
Disk fault 669
Disk full 670
Disk read only 671
Division by zero 672
Escape 673
Exponent range 674
File exists 675
File or path not found 676
Incorrect arguments 677
Invalid channel 678
Jump out of range 679
Logarithm range 680
Missing , 681
Missing \\#34; 682
Missing ) 683
Missing # 684
Missing \ 685
Missing ENDCASE 686
Missing ENDIF 687
Missing OF 688
Missing TO 689
Mistake 690
Multiple label 691
Negative root 692
No coprocessor 693
No room 694
No such FN/PROC 695
No such font 696
No such line 697
No such printer 698
No such system call 699
No such variable 700
Not in a FN or PROC 701
Not in a FOR loop 702
Not in a function 703
Not in a procedure 704
Not in a REPEAT loop 705
Not in a subroutine 706
Not in a WHILE loop 707
Number too big 708
OF not last 709
ON ERROR not LOCAL 710
ON range 711
ON syntax 712
Out of DATA 713
Size mismatch 714
Size needed 715
STEP cannot be zero 716
String too long 717
Syntax error 718
Too many open files 719
Type mismatch 720
Unknown error 721
WHEN/OTHERWISE not first 722

Format of Data in Memory

Memory map 723
Memory management 725
Program storage in memory 727
Variable storage in memory 728

Accessing the Windows API

Introduction to the Windows API 740
Changing the window title 742
Flashing the title bar 743
Finding the display size 744
Displaying a message box 745
Updating the screen 747
Producing a warning sound 748
Playing WAV files 749
Checking for a sound card 751
Timing program execution 752
Pausing a program 753
Reading the command line 754
Finding the filename 755
Discovering an 'unknown error' 756
Repositioning the window 757
Fixing the window size 759
Minimising or maximising the window 760
Forcing the window to stay on top 761
Removing the title bar 762
Using the entire screen 763
Drawing graphics to a printer 764
Printing coloured text 766
Printing in landscape format 767
Adding a menu bar
Adding popup and sub-menus
Ticking a menu item
Disabling a menu item
Deleting and inserting menu items
Creating a toolbar
Creating a status bar
Using the clipboard
Using dialogue boxes
Using system colours
Loading or saving part of a file
Aligning proportional-spaced text
Displaying enhanced metafiles
Using multimedia timers
Changing the process priority
Using scroll bars
Displaying GIF and JPEG images
Listing the disk directory
Opening a document file
Discovering the Windows version
Finding special folders
Finding the current font name
Distributing the BBCWIN font
Checking for input focus
Displaying icon files
Putting an icon in the SysTray
Using the system registry
Creating a screen saver
Drawing angled text
Downloading files from the internet
Serial and Parallel I/O
Serial input/output
Opening a serial port
Writing to a serial port
Reading from a serial port
Closing a serial port
Transferring files from a BBC Micro
Parallel input/output
Direct port input/output
Modem control input/output

Library Routines
Introduction to libraries
Array and matrix functions
Toolbars and status bars
Dialogue boxes
Trackbars and progress bars
Property sheets and wizards
Boxes and buttons
Sprites
Formatting and conversion
Multiple Document Interface
Calendar functions
Direct3D graphics
Plotting angled ellipses
Sorting data arrays
Socket (network) connections
Antialiased graphics
COM automation
String manipulation
Multiple output windows
No-wait function replacements
Callback routines
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Window subclassing</td>
<td>939</td>
</tr>
<tr>
<td>High speed timers</td>
<td>941</td>
</tr>
<tr>
<td>Parsing XML files</td>
<td>943</td>
</tr>
<tr>
<td>Extending the assembler</td>
<td>946</td>
</tr>
<tr>
<td>More assembler extensions</td>
<td>948</td>
</tr>
<tr>
<td>High quality sound patch</td>
<td>949</td>
</tr>
<tr>
<td><strong>MODE 7 - Teletext</strong></td>
<td></td>
</tr>
<tr>
<td>Introduction to MODE 7</td>
<td>951</td>
</tr>
<tr>
<td>Implementation</td>
<td>952</td>
</tr>
<tr>
<td>Character set (MODE 7)</td>
<td>953</td>
</tr>
<tr>
<td>Control codes</td>
<td>954</td>
</tr>
<tr>
<td>Coloured text (MODE 7)</td>
<td>956</td>
</tr>
<tr>
<td>Background colour (MODE 7)</td>
<td>957</td>
</tr>
<tr>
<td>Block graphics</td>
<td>958</td>
</tr>
<tr>
<td>Flashing</td>
<td>960</td>
</tr>
<tr>
<td>Double height</td>
<td>961</td>
</tr>
<tr>
<td>Reveal/conceal</td>
<td>962</td>
</tr>
<tr>
<td><strong>Hints and Tips</strong></td>
<td></td>
</tr>
<tr>
<td>Trapped OPENIN, OPENOUT or OPENUP</td>
<td>963</td>
</tr>
<tr>
<td>Using windows larger than 1920 x 1440 pixels</td>
<td>964</td>
</tr>
<tr>
<td>TrackPopupMenu</td>
<td>965</td>
</tr>
<tr>
<td>Passing floats to assembler code or DLLs</td>
<td>966</td>
</tr>
<tr>
<td>Precautions when using LOCAL arrays</td>
<td>967</td>
</tr>
<tr>
<td>Using status bars and toolbars with the Multiple Document Interface</td>
<td>968</td>
</tr>
<tr>
<td>Foreign language equivalent of TIME$</td>
<td>970</td>
</tr>
<tr>
<td>Loading a sprite (or icon) from memory</td>
<td>971</td>
</tr>
<tr>
<td>Setting the File Open dialogue's initial directory</td>
<td>972</td>
</tr>
<tr>
<td>Centering a hardcopy printout on the page</td>
<td>973</td>
</tr>
<tr>
<td>Using DATA in installed modules</td>
<td>974</td>
</tr>
<tr>
<td>Re-entrant ON MOVE interrupts</td>
<td>975</td>
</tr>
<tr>
<td>Cunning use of the CASE statement</td>
<td>976</td>
</tr>
</tbody>
</table>
Introduction to BBC BASIC

BBC BASIC is the programming language originally specified and adopted by the British Broadcasting Corporation for its groundbreaking Computer Literacy Project of the early 1980s. It was designed to be simple enough to be used by a complete beginner yet powerful enough to cope with the demands of the very largest and most complex programs, and that remains true today. During the intervening years BBC BASIC has been extended and ported onto at least seven different CPUs and more than thirty different platforms.

BBC BASIC for Windows is an advanced version of BBC BASIC which is fully integrated into the Microsoft Windows™ operating environment (Windows 95, 98, Me, NT4, 2000, XP, Vista, Windows 7, 8/8.1 or 10), providing the programmer with a familiar language but with an up-to-date user interface. It combines the simplicity of BASIC with the sophistication of a modern structured language, allowing you to write utilities and games, use sound and graphics, perform calculations and create complete Windows™ applications.

Getting Started

If you have never programmed in BASIC before, or you're so rusty that you feel the need for a refresher course, consider working through the Beginners' Tutorial which you can find either via the Help Menu or online.

If you've programmed in BASIC before, but aren't familiar with BBC BASIC, then you might like to dip into some of the tutorial chapters which cover differences between BBC BASIC and other dialects. You can also browse this manual or look up particular features which don't seem to work the way you're used to. The General Information chapter may be of particular interest.

If you are used to earlier versions of BBC BASIC (on the BBC Micro, on the Acorn Archimedes or BBC BASIC (86) on a PC) you may find the user interface somewhat unfamiliar. When you start BBC BASIC for Windows you are presented with a blank editing window into which you can type your BASIC program. Commands such as Load, Save, Chain, New and Renumber are activated by clicking with the mouse on drop-down menus or (in some cases) toolbar buttons, as is usual for Windows programs. There are also keyboard shortcuts for all the commands.
Once you have typed in, or loaded, your BASIC program you can run it by selecting the Run command from the Run menu or by clicking on the button in the toolbar. A second window will open in which appears any output from the BASIC program; error messages also appear here.

If you don't want to run a BASIC program, but simply want to enter commands for immediate execution (rather like a calculator) then select the Immediate Mode command from the Run menu, or click on the button. A second window will open into which you can type BASIC statements, in the same way as you could with previous versions of BBC BASIC.
Overview

*BBC BASIC for Windows* has a Graphical User Interface, including a BASIC program editor with syntax colouring, live indentation, search & replace and many other features. Built-in diagnostic aids such as run-time trace, listing of variables and single-step execution make debugging even the most complex programs straightforward.

A unique feature is the incorporation of an assembler which generates 32-bit code and accepts all 80486 instructions and some Pentium-specific instructions (except special-register and privileged instructions), including floating-point and MMX instructions. You can access the Windows™ Application Program Interface both from BASIC and from assembler code, allowing an experienced programmer to produce sophisticated applications.

The full version provides up to 512 Mbytes of user memory and allows you to create compact (typically less than 100K) stand-alone executable (.EXE) files from your BASIC programs, which will run without BBC BASIC having to be installed and without the need for any special DLLs. You can distribute such executables freely without any royalty being due.

*BBC BASIC for Windows* is largely compatible with Version 5 of BBC BASIC resident in the Acorn Archimedes and Iyonix computers. It also retains a high degree of compatibility with the BBC Microcomputer, including emulation of the SOUND and ENVELOPE statements, and the MODE 7 (teletext) screen mode.

In addition there are a number of major enhancements over these early versions, including data structures, an EXIT statement, PRIVATE variables, long strings, NUL-terminated strings, event interrupts, an address of operator, byte variables and arrays, a line continuation character, indirect procedure and function calls and improved numeric accuracy.
Running BBC BASIC

*BBC BASIC for Windows* can be run in a number of ways:

- Select it from the Start Menu (usually Start ... All Programs ... BBC BASIC for Windows).
- Double-click on its desktop icon.
- Double-click on BBCWIN.EXE or BBCWIN6.EXE (usually in C:\Program Files\BBC BASIC for Windows).
- At a command prompt issue the command **BBCWIN** or **BBCWIN6** (assuming that BBCWIN.EXE or BBCWIN6.EXE, respectively, is in the current directory or on the *PATH*).

Each of these methods starts *BBC BASIC for Windows* with an empty editing window. Alternatively you can start it and at the same time load a specified BASIC program for editing:

- Drag the BASIC program and drop it on the *BBC BASIC for Windows* icon.
- Right-click on the BASIC program and select **edit** from the list.
- At a command prompt enter the command **BBCWIN programe** or **BBCWIN6 programe** where *programe* is the name of the BASIC program you want to load (assuming that BBCWIN.EXE or BBCWIN6.EXE, respectively, is in the current directory or on the *PATH*).

If you only have the evaluation version substitute **BBCWDEM** for **BBCWIN** in the above instructions.

BBCWIN.EXE and BBCWIN6.EXE have one optional command-line switch, **-c** , which automatically invokes the Compile utility:

```
bbcwin6 -c programe.bbc
```

Once the compile dialogue has been dismissed by the user BBCWIN terminates.
Minimum system requirements

In order to be able to install and run *BBC BASIC for Windows* your system must meet these minimum requirements:

- IBM-compatible PC with 80486 or later processor.
- 16 Megabytes or more of Random Access Memory.
- VGA or higher resolution display.
- 6 Megabytes or more of free hard disk space.
- Microsoft Windows™ 95, 98, Me, NT4, 2000, XP, Vista, Windows 7, 8/8.1 or 10.

For correct operation under Windows™ 95, 98 or NT4 it may be necessary to install Microsoft Internet Explorer version 5.5 or later.

For the SOUND statement to work correctly, your processor clock speed should be 133 MHz or higher.
## Specification

*BBC BASIC for Windows* meets or exceeds the following specification:

<table>
<thead>
<tr>
<th></th>
<th><strong>Version 5</strong></th>
<th><strong>Version 6</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Memory available for user's program, data and stack</strong></td>
<td>Default approximately 1 Mbyte, maximum 256 Mbytes (depending on amount of RAM installed)</td>
<td>Default approximately 2 Mbytes, maximum 512 Mbytes (depending on amount of RAM installed)</td>
</tr>
<tr>
<td><strong>Size of BASIC program</strong></td>
<td>Limited only by available memory (maximum 65535 numbered lines)</td>
<td>Limited only by available memory (maximum 65535 numbered lines)</td>
</tr>
<tr>
<td><strong>Length of program lines</strong></td>
<td>Maximum 251 characters (tokenised) excluding line number</td>
<td>Maximum 251 characters (tokenised) excluding line number</td>
</tr>
<tr>
<td><strong>Integer variables</strong></td>
<td>32 bits, signed (-2147483648 to +2147483647)</td>
<td>32 bits, signed (-2147483648 to +2147483647)</td>
</tr>
<tr>
<td></td>
<td>64 bits, signed (-9223372036854775808 to +9223372036854775807)</td>
<td>64 bits, signed (-2.23370368547758E308 to +2.23370368547758E308)</td>
</tr>
<tr>
<td><strong>Floating-point (real) variables</strong></td>
<td>Default 32-bit mantissa plus 8-bit exponent (±5.9E-39 to ±3.4E38); precision approximately 9 significant figures</td>
<td>Default 64-bit mantissa plus 16-bit exponent (±3.4E-4932 to ±1.1E4932); precision approximately 19 significant figures</td>
</tr>
<tr>
<td></td>
<td>Optionally 53-bit mantissa plus 11-bit exponent (±2.3E-308 to ±1.7E308); precision approximately 15 significant figures</td>
<td>Optionally 53-bit mantissa plus 11-bit exponent (±2.3E-308 to ±1.7E308); precision approximately 15 significant figures</td>
</tr>
<tr>
<td><strong>String variables</strong></td>
<td>Maximum length 65535 8-bit characters</td>
<td>Length limited only by available memory</td>
</tr>
<tr>
<td><strong>Array variables</strong></td>
<td>Maximum number of dimensions 255, maximum size of each dimension limited only by available memory</td>
<td>Maximum number of dimensions 255, maximum size of each dimension limited only by available memory</td>
</tr>
<tr>
<td>Number of variables</td>
<td>Limited only by available memory</td>
<td>Limited only by available memory</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>Length of variable names</td>
<td>Limited only by available memory and maximum line length</td>
<td>Limited only by available memory and maximum line length</td>
</tr>
<tr>
<td>Levels of nesting</td>
<td>Limited only by available memory</td>
<td>Limited only by available memory</td>
</tr>
</tbody>
</table>
Example programs

A number of example programs are supplied with *BBC BASIC for Windows*. They can be found in the EXAMPLES folder (this is the default file folder following installation). If you have selected the default location this will usually be C:\Program Files\BBC BASIC for Windows\EXAMPLES\.

Most of these programs will run using the trial version of *BBC BASIC for Windows*, although (depending on which programs you have run beforehand) you may need to restart the application first to free up memory. A few of them (notably ASMTEST.BBC, DLGEDIT.BBC, SEARCHBBC.BBC, SORTTEST.BBC and the file-handling demonstrations) require more than the 32K of user memory available with that version.

The example programs are located in sub-folders under the following categories:

**FILING**

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F-?????.BBC</td>
<td>A set of 12 programs to demonstrate file handling.</td>
</tr>
</tbody>
</table>

**GAMES**

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2048.BBC</td>
<td>Slide the numbered tiles to combine them in pairs until you reach a total of 2048.</td>
</tr>
<tr>
<td>ANIMAL.BBC</td>
<td>A 'guess the animal' game which learns from experience (well, almost!).</td>
</tr>
<tr>
<td>BUGGY.BBC</td>
<td>A 'driving' game in which you need to avoid the verges and potholes.</td>
</tr>
<tr>
<td>DIBLEY.BBC</td>
<td>A puzzle in which you have to work out how to move obstacles to reach the exit.</td>
</tr>
<tr>
<td>HANOI.BBC</td>
<td>Solves the 'Towers of Hanoi' problem; not really a game as the computer does all the work!</td>
</tr>
<tr>
<td>RHEOLISM.BBC</td>
<td>A Tetris-like game in one (highly convoluted) line of BBC BASIC!</td>
</tr>
<tr>
<td>SUDOKU.BBC</td>
<td>Plays and solves the popular number puzzle.</td>
</tr>
</tbody>
</table>
TRIPLES.BBC  Move the wooden objects to create a row of three or more that are identical.

GENERAL

ASMTEST.BBC  A test program for the built-in assembler.
ASMTEST2.BBC A program to test the assembler with floating-point instructions.
ASMTEST3.BBC A program to test the assembler with MMX instructions.
ASMTEST4.BBC A program to test the extensions supported by the ASMLIB library.
ASMTEST5.BBC A program to test the extensions supported by the ASMLIB2 library.
BANNER.BBC  Displays horizontally-scrolling news headlines; uses the XMLLIB library.
CALENDAR.BBC A perpetual calendar, to demonstrate the DATELIB library.
MODE7DEM.BBC A program to test and demonstrate MODE 7.
PROMPTER.BBC Smoothly scrolls text up the screen; uses the NOWAIT library.
SCROLL.BBC  A demonstration of scrolling over a large canvas, using @ox% and @oy%.
SHEET.BBC  A simple spreadsheet program. See the file SHEET.TXT for details.
SORTTEST.BBC Tests and demonstrates the SORTLIB library.
UNICODE.BBC Demonstrates BBC BASIC for Windows' multilingual text capability.
UNICODEP.BBC Demonstrates multilingual text and right-to-left output to the hardcopy printer.

GRAPHICS

BOUNCE.BBC  Coloured balls bounce across the screen; the program illustrates the use of data structures.
CHAIN.BBC  A graphics demonstration using icons with transparent backgrounds.
CLOCK.BBC  A very realistic analogue clock display.
DOODLE.BBC  A simple 'etch-a-sketch' drawing program.
ELLIPSES.BBC  Multi-coloured patterns created from rotated ellipses.
FERN.BBC  A lifelike fractal image produced by a simple formula.
FLASHING.BBC  Emulates the BBC Micro's flashing colours.
FLIER.BBC  A spacecraft animation.
JCLOCK.BBC  An unusual clock that follows mouse movement. Loosely based on a Javascript program.
MANDEL.BBC  Plots the famous fractal very quickly, using SSE instructions.
OPENGL.BBC  A translation to BBC BASIC of an OpenGL 3D graphics demonstration program.
PERSIAN.BBC  A 'Persian carpet' graphics demonstration.
POINTER.BBC  A demo version of a LOGO like 'turtle graphics' language.
POLYGON.BBC  A graphics demonstration program.
PYRAMID.BBC  A demonstration of 3D graphics using the Direct3D library.
SAUCER.BBC  A simple but effective 3D drawing.
SCARAB.BBC  A scarab beetle-like pattern created from the behaviour of sines and cosines.
SINE.BBC  Illustrates the 'text at graphics cursor' (VDU 5) mode.
SLIDERULE.BBC  A demonstration of large sprites created using FNcreatespritefrombmp.
SPECTRUM.BBC  Proves you can display more than 16 colours at once!
SPRITES.BBC  A demonstration of BBC BASIC for Windows sprites.
SNOWSCENE.BBC  A winter scene featuring a realistic fractal tree which is different every time.
TEAPOT.BBC  A realistic 'tumbling teapot' animation, using D3DX8BBC.DLL.
WHEEL.BBC  A rotating colour wheel, showing the use of *REFRESH to control animation.
WORLD.BBC  A 'rotating globe' animation, demonstrating mapping a texture onto a 'curved' surface.
**SOUNDS**

CDPLAYER.BBC  Turns your CD-ROM drive into a simple CD player.
PIANO.BBC     A polyphonic 'piano' using the PC's keyboard.
POLLY.BBC     A demonstration of SOUND and ENVELOPE
SKATERS.BBC   A demonstration of four-voice music, enabled using *TEMPO .
SPEAK.BBC     Uses the COMLIB library to access the Windows™ Text-to-Speech engine.
TOCFUGUE.BBC  A demonstration of stereo sound using the HQSOUND library.
TOFAIR.BBC    Another demonstration of the High Quality sound library.

**TOOLS**

BEEBXFER.BBC  A utility for transferring files from a BBC Micro .
DLGEDIT.BBC   A 'visual' dialogue box editor, simplifying the design of custom dialogue boxes.
FCONVERT.BBC  A utility for converting Acorn data files .
ICONEDIT.BBC  A simple icon (or sprite ) editing program.
PROFILER.BBC  An execution time profiler to help tune programs for maximum speed.
SEARCHBBC.BBC Searches .BBC files for a specified string.
SETTAB.BBC    Programs the TAB key in the editor to move to a specified column.
WMF2EMF.BBC   A utility for converting .WMF files to .EMF files

**WINDOWS**
DLGDEMO.BBC  A demonstration of the creation of a dialogue box.
GUIDEMO.BBC  A program with a Windows™-style interface.
MDIDEMO.BBC  Demonstrates the Windows™ Multiple Document Interface.
MENU.BBC     A demonstration of how to add a menu bar.
MENU2.BBC    As MENU.BBC, but using a popup menu.
MULTIDEM.BBC Creates multiple output windows using the MULTIWIN library.
OPTICS.BBC   Uses pushbuttons to control a graphical demonstration.
REBAR.BBC    Creates a 'rebar' control using the WINLIB5A library.
SOLVE.BBC    Solves simultaneous equations using the ARRAYLIB and WINLIB5 libraries.
TEXTEDIT.BBC A simple text editor, utilising Windows' built-in edit control.
VSCROLL.BBC  A demonstration of the use of a scroll bar.
WIDGETS.BBC  A demonstration of a floating tool bar, a trackbar and a progress bar.
Differences from BBC BASIC (86)

BBC BASIC for Windows differs from BBC BASIC (86) for MS-DOS in the following respects:

- Line numbers may be omitted, except where they are required as the destination of a GOTO, GOSUB or RESTORE.
- Keywords may optionally be accepted and displayed in lower-case.
- The following BASIC V statements are accepted:
  - CASE ... OF ... WHEN ... OTHERWISE ... ENDCASE
  - CIRCLE [FILL ] x,y,r
  - ELLIPSE [FILL ] x,y,a,b
  - ERROR n, "string"
  - FILL x,y
  - Multi-line IF ... THEN ... ELSE ... ENDIF
  - INSTALL "libfile"
  - LINE x1,y1,x2,y2
  - MOUSE x,y,b
  - MOUSE ON n
  - MOUSE OFF
  - OFF
  - ORIGIN x,y
  - QUIT
  - RECTANGLE [FILL ] x,y,dx,dy [TO xnew,ynew]
  - SWAP var, var
  - WHILE ... ENDWHILE
- The following BASIC V functions are accepted:
  - DIM (array())
  - DIM (array(), n)
  - GET$ #file
  - REPORT$
  - SUM (array())
- The following BASIC V operators are accepted:
  - +=
  - -=
  - <<
  - >>
  - >>>
• The following statements have new forms:
  o BPUT #file, string$
  o COLOUR l,p
  o COLOUR l,r,g,b
  o GCOL c
  o LOCAL array()
  o ON
  o ON CLOSE
  o ON MOUSE
  o ON MOVE
  o ON SYS
  o ON TIME
  o PLOT x,y
  o PLOT 160-207
  o RESTORE ERROR

• You can assign a whole array, or set the elements of an array to constant values:
  o A() = B()
  o A$() = B$()
  o A() = 1,2,3
  o A$() = "one","two","three"

• You can assign to a sub-string:
  o LEFT$( A$,N) = string$
  o MID$( A$,N,M) = string$
  o RIGHT$( A$,N) = string$

• You can access the Windows™ Application Program Interface with the SYS statement.

• You can use binary constants, e.g. %01011111.

• The resident assembler generates 32-bit code and accepts all 80286, 80386 and 80486 instructions (except special-register and privileged instructions). A few Pentium instructions (notably CPUID) are also accepted.

If you want to run a program which was originally written for BBC BASIC (86) you can include the *EGA command in your program; this results in subsequent MODE statements selecting modes with dimensions compatible with BBC BASIC (86).
Version 6 changes

Versions 6.00a and later of BBC BASIC for Windows incorporate some significant changes compared with earlier versions, as follows:

BASIC interpreter

- Support for legacy 40-bit (5 byte) floating-point variables has been removed.
- 'Suffixless' variables are always 80-bit (10-byte) numeric variants which can contain either a 64-bit signed integer (a whole number in the range -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807) or an 80-bit long double floating-point value (with a precision of approximately 19 significant figures).
- A new 64-bit signed integer variable type is introduced, which has a %% suffix (for example 'number%%'). This data type can also be used as an array or a structure member.
- Strings are no longer limited to a maximum length of 65,535 characters. A string now has a 32-bit length value, so effectively may be any length, subject only to the amount of available memory (heap).
- The PTR# and EXT# pseudo-variables, which read or write the file pointer and file length respectively, now return or accept 64-bit values, so random-access files are no longer limited to a maximum length of 2 Gbytes.
- The assembler accepts the FCOMI, FCOMIP, FUCOMI and FUCOMIP instructions.

Operating System interface

- The *FLOAT command is extended to accept the parameter 80 in addition to 40 and 64 (i.e. *FLOAT 80). The command affects only the precision with which floating-point values are stored and retrieved using floating-point indirection or in data files (PRINT# and INPUT# statements). For compatibility with earlier versions of BBC BASIC the default is *FLOAT 40.
- A new *HEX command determines whether the hexadecimal conversion
and ~) and bit-shift (<< and >>) operators treat their operands as 32-bit or 64-bit integers. In the (default) *HEX 32 mode the operators work in a fashion which is compatible with earlier versions of BBC BASIC. In *HEX 64 mode they support 64-bit integers.

- The *FONT command supports the strikeout attribute, signified by the character Q. The strikeout attribute (Q) and the underscore attribute (U) are mutually-exclusive.

**Compiler**

- A new compiler directive REM!Fast is provided, which can speed up run-time accesses to specified variables, arrays, structures, functions and procedures in the compiled program.

- The manifest embedded in a compiled executable, when the Use Windows XP visual styles option is enabled, includes the necessary identifier to indicate compatibility with Windows 10™.

**Program editor and IDE**

- A new menu selection Run... Debug, and associated toolbar button (it looks like a beetle), are provided. The effect is to run the current program but immediately to Pause it, enable Trace mode and open the List Variables window.

- A new toolbar button corresponding to the Step Line menu selection is provided, for the convenience of debugging. It looks like an upside-down version of the eject symbol.

**Compatibility considerations**

Most programs should not require any alteration to run properly in BBC BASIC for Windows version 6. However in a few cases changes may be necessary to ensure compatibility, as follows:

- Variant (suffixless) numeric variables and string variables - and particularly arrays of these data types - use more memory. To accommodate this, the default amount of memory available has been doubled (from 16K to 32K in the trial version, and from 1 Mbyte to 2 Mbytes in the full version) but if your
program explicitly allocates more memory by raising HIMEM it is quite likely that you will need to raise it even higher.

- A program will fail if it assumes that, in *FLOAT 64 mode, a 'suffixless' variable or array (for example var) is 64-bits. If such a variable is passed by reference to a Windows™ API function, which requires a parameter of type 'double', a hash suffix character must be added to all occurrences of the variable (for example var# ) to restore correct operation. This will not affect compatibility with earlier versions of BBC BASIC for Windows.

- Programs which make assumptions about the size and/or format of data items in memory, for example which calculate an offset into a variant ('suffixless') numeric array or string array, and then access the data using indirection or assembler code, will need to be adapted to take account of the different sizes and formats of those data types.

- Be careful when comparing two numeric values, if one may have suffered a reduction in precision (for example by being stored in memory using floating-point indirection or having been written to and subsequently read from a data file using PRINT# and INPUT# ). In earlier versions of BBC BASIC for Windows you could usually assume that no loss of precision would result, and therefore that 'before' and 'after' values could safely be compared for equality. In version 6 this is no longer true, unless you specify *FLOAT 80 mode.

- If you update a file pointer and/or file size in assembler code ('setptr' and 'settext' functions respectively) you must modify the code to initialise the edx register to contain the most-significant 32-bits of the required pointer or size (normally that will mean setting it to zero, e.g. xor edx,edx).

- If your program contains assembler code which is activated using the CALL statement, bear in mind that if any parameters passed to the code are 'suffixless' numerics or 'movable' strings the code will receive different 'type' values and data formats. Specifically, a variant numeric will have a type of 10 (74 for an array) and be stored in 10 bytes, and a 'movable' string will have a type of 136 (200 for an array) and have an 8-byte descriptor (32-bit pointer and 32-bit length).

If you have a program which is not compatible with version 6, and you prefer not to make the necessary changes, BBC BASIC for Windows version 5.95a remains available.
Notices

**BBC BASIC for Windows** is thought to be free from serious bugs, and it is highly unlikely that it could do any harm to other software or systems. However there is NO warranty that it will not and its use is at your own risk.

**BBC BASIC for Windows** is the Copyright © 2001-2017 of R. T. Russell. It is supplied for use on one computer at a time only. Neither the program nor its accompanying documentation may be copied for simultaneous use by the purchaser nor for use by anyone other than the purchaser.

Schools and colleges are granted permission to reproduce or adapt sections of this manual to support teaching in a classroom context.

The name **BBC BASIC** is used with the permission of the British Broadcasting Corporation.

The aPLib compression library is used with permission. Copyright © 1998-2014 by Joergen Ibsen, All Rights Reserved.
Introduction to the IDE

When you start BBC BASIC for Windows you are presented with the command and editing window. This has the normal features which make up the modern Windows™ interface: a title bar, a set of drop-down menus, a toolbar, an editing pane and a status bar. It is possible to hide the toolbar and/or the status bar by means of selections in the Options menu.

You can move this window wherever you like on your Windows™ desktop, and you can re-size it by dragging one of the corners or sides. The bottom right-hand corner has a special area which makes it easier to resize the window by dragging there. BBC BASIC for Windows will remember the size of the window and set it to the same size when it is next started.
The title bar

The title bar contains the name of the application (BBC BASIC for Windows), the name of the current BASIC program (or 'untitled' if the New command has been issued), a minimise button, a maximise button and a close button. If you right-click in the title bar you will get a context menu containing the Restore, Move, Size, Minimize, Maximize and Close items.

If you close the window (by clicking on the close button, selecting Close from the right-click menu or using the keyboard shortcut Alt-F4) you will be prompted to save the current program if any changes have been made.
The menu bar

The menu bar contains the drop-down menus File, Edit, Utilities, Options, Run and Help. These menus may be activated by clicking on the appropriate menu name or by using a keyboard shortcut (hold down Alt and press the key corresponding to the underlined character). If any of the menu items are unavailable, they will be 'greyed out' and clicking on them will have no effect.
The toolbar

The toolbar provides easy access to the most common commands, without having to select them from the drop-down menus, simply by clicking on the appropriate button. If any of the commands are unavailable the button will be 'greyed out' and clicking on it will have no effect. The commands for which buttons are provided are as follows:

From the File menu: New, Load, Save and Print.
From the Edit menu: Undo, Redo, Cut, Copy, Paste, Find and Replace.
From the Utilities menu: Renumber and Compile.
From the Run menu: Run, Debug, Stop, Pause, Step, Step Line and Immediate.

The final (question mark) button does not correspond to any of the menu commands, but instead activates context sensitive help. If you click on this button, then click on a keyword within the editing pane, a short description of the use and function of that keyword will be displayed.
The status bar

When a menu item is highlighted, the status bar displays a brief description of the function of that command. Otherwise, the status bar is split into eight regions: a message region (which always reads "Press F1 for Help"), the current character encoding, the current length of the BASIC program, the current position of the text cursor (caret) in the editing pane, the current Insert/Overtype status, the current Caps Lock status, the current Num Lock status and the current Scroll Lock status.

Character encoding

The character encoding (ANSI or Unicode) assumed for the current BASIC program, as selected in the Options menu or determined automatically when the program was loaded.

Program length

The length, in bytes, of the currently-loaded BBC BASIC program. This corresponds to the size of the file if the program is saved in tokenised (.BBC) format, and (usually) to the value of TOP -PAGE when the program is running.

Cursor position

The current text cursor (caret) position is displayed in the form x,y where x is the horizontal position (the leftmost column is 0) and y is the vertical position (the topmost line is 0).

Insert/Overtype

Pressing the Insert key on the keyboard toggles between insert mode and overtype mode. Overtype mode is indicated by OVR being displayed in the status line; in this mode any character typed will over-write the existing character in that position on the line. The normal editing mode is insert mode, in which any characters typed 'push' the rest of the line to the right to make room.
Caps Lock

Pressing the Caps Lock key on the keyboard toggles the **caps lock** mode on and off. Caps lock mode is indicated by **CAPS** being displayed in the status line; in this mode the alphabetic keys (A-Z) generate capital letters, unless the Shift key is held down in which case they generate lower-case letters. When caps lock mode if off, the alphabetic keys generate lower-case letters, unless the Shift key is held down.

Num Lock

Pressing the Num Lock key on the keyboard toggles the **num lock** mode on and off. Num lock mode is indicated by **NUM** being displayed in the status line; in this mode the number keys (0-9) in the separate numeric cluster generate numbers, unless the Shift key is held down in which case they generate cursor-movement codes. When num lock mode if off, the numeric cluster generates cursor-movement codes.

Scroll Lock

Pressing the Scroll Lock key on the keyboard toggles the **scroll lock** mode on and off. Scroll lock mode is indicated by **SCRL** being displayed in the status line; *BBC BASIC for Windows* ignores this mode.
File menu

<table>
<thead>
<tr>
<th>File Menu Items</th>
<th>Keyboard Shortcuts</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>Ctrl+N</td>
</tr>
<tr>
<td>Load...</td>
<td>Ctrl+L</td>
</tr>
<tr>
<td>Save</td>
<td>Ctrl+S</td>
</tr>
<tr>
<td>Save As...</td>
<td></td>
</tr>
<tr>
<td>Chain...</td>
<td></td>
</tr>
<tr>
<td>Insert...</td>
<td>Ctrl+I</td>
</tr>
<tr>
<td>Page Setup...</td>
<td></td>
</tr>
<tr>
<td>Print...</td>
<td>Ctrl+P</td>
</tr>
<tr>
<td>1 MODE7DEM.BBC</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Exit</td>
<td></td>
</tr>
</tbody>
</table>

The **file menu** is concerned with loading, saving and printing BBC BASIC programs, and with exiting from *BBC BASIC for Windows*. The file menu can be accessed by clicking on its name in the menu bar or by means of the keyboard shortcut Alt-F.

**New**

The **New** command (keyboard shortcut Ctrl+N) deletes the current BASIC program from memory, giving you a 'clean sheet' into which to type a new program. If the previous program was not saved, you are first prompted to save it. If a BASIC program is currently running (i.e. its output window is open) the New command is inhibited.

**Load**

---

Page 44
The **Load** command (keyboard shortcut Ctrl+L or Ctrl+O) loads a BBC BASIC program into memory. It will accept files in BBC BASIC (86) internal (tokenised) format, in Acorn (BBC Micro or Archimedes) internal format, or in ASCII (plain text) format. The command calls up the Load dialogue box:

If the previous program was not saved, you are first prompted to save it. If a BASIC program is currently running (i.e. its output window is open) the Load command is inhibited.

**Save**

The **Save** command (keyboard shortcut Ctrl+S) saves the BBC BASIC program currently in memory to a file. The filename used is the same as that from which the program was loaded (i.e. the contents of the file are replaced). If the program was loaded from a BBC BASIC (86) or Acorn internal format (tokenised) file it will be saved as a BBC BASIC (86) internal format file (in the latter case you are warned that the file format is being altered). If the program was loaded from an ASCII (plain text) file it will be saved as an ASCII file. You can use the **Save As** command to override these defaults (a program must be saved in internal format if it is to be loaded at run-time with the CALL, CHAIN or INSTALL statement).
If the program was not loaded from a file (i.e. it was typed in or imported via the clipboard) the Save command will display the Save As dialogue box so you can enter the desired filename and format. If the current BASIC program is unchanged, so it doesn't need to be saved, the Save command is inhibited.

If the file you are about to overwrite has been modified by another application (e.g. another copy of *BBC BASIC for Windows*) since it was first loaded, the Save command issues a warning. If you continue with the Save operation the modifications carried out by the other program will be lost.

**Save As**

The **Save As** command (keyboard shortcut Shift+Ctrl+S) saves the BBC BASIC program currently in memory to a file. You are prompted to select a file name and a file format: BBC BASIC internal (tokenised) format or ASCII (plain text) format. Use Save As rather than Save if you need to change the disk, directory (folder), filename or format to something different from their current values. By default the program is saved to the directory last used for a successful Save As operation (or failing that to the user's **Documents** folder).

**Chain**

The **Chain** command loads a BBC BASIC program into memory and then runs it automatically. It is equivalent to Load followed by Run. If the previous program was not saved, you are first prompted to save it. If a BASIC program is currently running (i.e. its output window is open) the Chain command is inhibited.

**Insert**

The **Insert** command (keyboard shortcut Ctrl+I) loads a BBC BASIC program (or part of a program) into memory, but instead of it *replacing* the current program it is *inserted* in the program before the line containing the cursor (caret). This line, and subsequent lines, are 'moved down' to make space for the inserted lines.

The effect is similar to the Paste command, except that the inserted lines are read from a file (and may be in any of the formats accepted by the Load command). If a BASIC program is currently running (i.e. its output window is open) the Insert command is inhibited.
Page Setup

The Page Setup command allows you to select the page orientation (portrait or landscape) and set the size of the page margins. These affect the page layout when you print out the current BASIC program using the Print command.

Print

The Print command (keyboard shortcut Ctrl+P) allows you to print out one or more copies of the BASIC program currently in memory. A 'dialogue box' is displayed which allows you to:

- Select the printer to use.
- Set the printer's properties (for example paper type and printing quality).
- Optionally print to a file.
- Print the entire program, a range of pages or a selected region.
- Choose whether and how to highlight syntactical elements:

If you have a colour printer you can choose to highlight the printout in the same colours as are used on the screen (set with the Set Colours command). Alternatively you can choose to highlight the printout using underlining, italics and bold-face emphasis; these work with either a black-and-white or a colour printer.

You can force a 'new page' by including the following statement in your program:

REM!Eject

When the statement is encountered printing will continue at the top of the next page.

Recent files
The nine most-recently loaded files are listed in the File menu, and can be re-loaded simply by clicking on the appropriate entry or by using the keyboard shortcuts Ctrl+1 to Ctrl+9 (or Alt-F followed by the digit 1 to 9). When one of the entries is highlighted, the full pathname of the file is displayed in the Status Bar. If the previous program was not saved, you are first prompted to save it. If a BASIC program is currently running (i.e. its output window is open) the recent files are inhibited.

Exit

The **Exit** command quits from *BBC BASIC for Windows* and closes all its windows. If the current program in memory has not been saved, you are first prompted to save it.
Edit menu

<table>
<thead>
<tr>
<th>Menu</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undo</td>
<td>Ctrl+Z</td>
</tr>
<tr>
<td>Redo</td>
<td>Ctrl+Y</td>
</tr>
<tr>
<td>Cut</td>
<td>Ctrl+X</td>
</tr>
<tr>
<td>Copy</td>
<td>Ctrl+C</td>
</tr>
<tr>
<td>Paste</td>
<td>Ctrl+V</td>
</tr>
<tr>
<td>Delete</td>
<td>Del</td>
</tr>
<tr>
<td>Select All</td>
<td>Ctrl+A</td>
</tr>
<tr>
<td>Find...</td>
<td>Ctrl+F</td>
</tr>
<tr>
<td>Find Next</td>
<td>F3</td>
</tr>
<tr>
<td>Replace...</td>
<td>Ctrl+R</td>
</tr>
</tbody>
</table>

The **edit menu** is concerned with editing or searching the current BBC BASIC program (i.e. the program loaded into memory). The edit menu can be accessed by clicking on its name in the menu bar or by means of the keyboard shortcut Alt-E.

**Undo**

The **Undo** command (keyboard shortcut Ctrl+Z) reverses the effect of the last editing operation (typing, delete, paste, renumber, insert file, replace or drop); the type of operation is indicated in the menu. If there is nothing to be undone, the message "Can't undo" is shown in grey. The maximum number of undo levels is determined by the Customize command.

**Redo**

The **Redo** command (keyboard shortcut Ctrl+Y) reverses the effect of a previous undo command; the type of operation to be re-done is indicated in the menu. If there is nothing that can be re-done, the message "Can't redo" is shown in grey.

**Cut**

The **Cut** command (keyboard shortcut Ctrl+X) transfers any selected text to the clipboard, and deletes it from the program. If no text is selected, the Cut command is inhibited. Once the selected text is in the clipboard, you can Paste it either elsewhere
in your BASIC program or into any other application which supports pasting of text from the clipboard. Any previous contents of the clipboard are discarded.

Selected text is highlighted in reverse-video. You can select text in the following ways:

- Double-clicking on a word selects that word (and the following space, if any).
- Holding down the left mouse button and 'dragging' it selects a block of text. Dragging the mouse horizontally selects part or all of a line. Dragging the mouse vertically selects a block of lines (only entire lines may be selected).
- Holding down the Shift key and moving the cursor with the arrow (or other cursor-movement) keys selects a block of text. Moving the cursor horizontally selects part or all of a line. Moving the cursor vertically selects a block of lines (only entire lines may be selected).
- Holding down the Ctrl key and clicking on a line selects the entire line.

Once a block of text has been selected, you can alter the selection by holding down Shift and moving the cursor. If you want to change the start point of a selection, make the initial selection from right to left or from bottom to top. If you want to change the end point of a selection, make the initial selection from left to right or from top to bottom.

Copy  

The Copy command (keyboard shortcut Ctrl+C) transfers any selected text to the clipboard, but unlike Cut it leaves the text in the BASIC program. If no text is selected, the Copy command is inhibited. Once the selected text is in the clipboard, you can Paste it either elsewhere in your BASIC program or into any other application which supports pasting of text from the clipboard. Any previous contents of the clipboard are discarded.

See Cut for details of how text may be selected.

Paste  

The Paste command (keyboard shortcut Ctrl+V) inserts the contents of the
clipboard into the BBC BASIC program at the current position of the text cursor (caret). If the clipboard contains no text, the Paste command is inhibited. You can use Paste to transfer a block of code from elsewhere in your BASIC program, or to insert a block of code which has been placed in the clipboard by any other application. For example you can transfer code examples in the BASIC documentation files into your program by first copying them to the clipboard from the Help window then pasting them at the required place.

**Delete**

The **Delete** command deletes any selected text, without copying it into the clipboard. See Cut for details of how text may be selected. The delete command has an identical effect to pressing the Delete key on the keyboard. Text which is deleted can only be recovered by using the Undo command.

**Select All**

The **Select All** command (keyboard shortcut Ctrl+A) selects the entire BASIC program. It is useful if you want to transfer the program into another application (e.g. a word processor) via the clipboard

**Find**

The **Find** command (keyboard shortcut Ctrl+F) allows you to search your BASIC program for the occurrence of some specified text. The command opens the Find dialogue box into which you can type the text you wish to search for. You can also specify whether the case (capitals or lower-case) is significant and whether to match only an entire word (i.e. the text must be preceded and followed by non-alphabetic characters):

![Find Dialogue Box]

To start the search click on Find Next, press the Return (Enter) key or use the
shortcut Alt+F. The search begins at the position of the text cursor (caret) when the Find command was issued. If you want to search the entire program, ensure that you move the cursor to the start of the program before issuing the command.

If the specified text is found it is highlighted (i.e. selected) and the cursor is moved to the end of the highlighted region. The display is automatically scrolled so that the selected text is visible. Clicking on Find Next again causes the search to resume from this point and the next occurrence of the text (if any) to be found. If the text is not found before the end of the program the message "Text not found" is displayed; clicking Find Next again causes the search to restart from the beginning of the program.

You can click on Cancel at any time to abort the search and dismiss the dialogue box.

**Find Next**

Once the Find dialogue box has been dismissed, you can continue to search for the next and subsequent occurrences of the search string by using the Find Next command or by pressing F3. If the text is not found before the end of the program the message "Text not found" is displayed; selecting Find Next again causes the search to restart from the beginning of the program.

**Replace**

The Replace command (keyboard shortcut Ctrl+R) allows you to search your BASIC program for the occurrence of some specified text and then, optionally, replace it with some different text. The command opens the Replace dialogue box into which you can type the text you wish to search for and the text you want it to be replaced with. You can also specify whether the case (capitals or lower-case) is significant and whether to match only an entire word (i.e. the text must be preceded and followed by non-alphabetic characters):
To start the search click on Find Next, press the Return (Enter) key or use the shortcut Alt-F. The search begins at the position of the text cursor (caret) when the Replace command was issued. If you want to search the entire program, ensure that you move the cursor to the start of the program before issuing the command.

If the specified text is found it is highlighted (i.e. selected) and the cursor is moved to the end of the highlighted region. The display is automatically scrolled so that the selected text is visible. To confirm that you want the text to be replaced click on Replace; this will replace the text and search for the next occurrence of the text (if any).

Alternatively if you do not wish to replace this occurrence of the text click on Find Next. This will skip the replace operation and search for the next occurrence of the text (if any). If the text is not found before the end of the program the message "Text not found" is displayed.

If you are certain that you want every occurrence of the text (from the current point to the end of the program) to be replaced, click on Replace All.

You can click on Cancel at any time to abort the search and dismiss the dialogue box.
Utilities menu

The **utilities menu** contains commands to aid debugging of your BASIC program, to renumber your program and to convert your program to a stand-alone executable file. The utilities menu can be accessed by clicking on its name in the menu bar or by means of the keyboard shortcut Alt-U.

**Trace**

The **Trace** command allows you to monitor execution of your program. Selecting this command alternately enables and disables Trace mode, as indicated by the presence of a tick mark next to Trace in the menu. You can enable (or disable) Trace mode either before executing your program or once it is running.

In Trace mode, the program statement which is currently being executed is highlighted, and this highlight moves as your program is running (unless the statement currently being executed is in a CALLeD or INSTALLed module). If your program gets stuck in an 'infinite loop', you will be able to see what part of the program is being executed. If your program stops running because of an error (or an END or STOP statement) the last statement executed will remain highlighted for as long as the program's output window remains open, or until you enter a command in immediate mode.

Trace can be used in conjunction with the Pause and Step commands in order to trace execution of your program one statement at a time.
List Variables

The **List Variables** command opens a window which displays the current values of the variables in your program (variables which have not yet been accessed are not displayed, except for the static variables which are always listed). The displayed values change as your program runs. You can change the size of and/or scroll the List Variables window so that the variable(s) in which you are interested are displayed.

In the case of arrays, the name, number of dimensions and sizes of each dimension are displayed, but the values of the individual elements are not. If you need to monitor the value of an array element, add an appropriate statement or statements to your program to copy the value to a conventional variable.

In *FLOAT64 mode* floating-point variable names (also procedure and function names) are shown with a `#` (hash) suffix.

You can select the List Variables command either before executing your program or once it is running. List Variables can be used in conjunction with the Pause and Step commands in order to monitor the values of variables as your program is executed one statement at a time.

---

Renumber

The **Renumber** command allows you to renumber the lines of your program, or alternatively to remove all the line numbers (except those needed as the destination of a GOTO, GOSUB or RESTORE statement). The command opens the Renumber dialogue box into which you can type the line number for the first line of your program and the increment (step) between consecutive line numbers. Both values default to 10:
Renumbering your program can be useful when debugging, because if an (untrapped) error occurs the line number at which the error occurred is displayed. Once your program is complete and debugged, you may prefer to remove the line numbers so as not to clutter the program listing.

The Renumber command will fail if the specified increment is zero or if the specified values would result in a line number greater than the maximum allowed (65535).

**Compile**

The **Compile** command allows you to create a stand-alone executable file from your BBC BASIC program. Although this is not a true compiler in the traditional sense (your program is still interpreted) the end result is effectively the same. The executable file generated by the Compile command requires only Windows™ to run: it does not require *BBC BASIC for Windows* to be loaded, nor does it need any DLL files other than those supplied as standard with Windows. You can distribute such executables freely without any royalty for BBC BASIC being due.

The Compile command opens a dialogue box which allows you to choose the name of the executable file, select an appropriate icon, determine the initial window size and choose whether or not to 'sign', 'crunch' or 'encrypt' the program:
Executable

The filename of the executable file is set by default to be the same as the name of your BASIC program, but with the extension '.exe' rather than '.bbc' (or '.bas'). However you can change the name and/or the directory path (folder) by typing them directly into the Filename field, by clicking on the Browse button or by using the REM!Exefile compiler directive.

By default the executable file is stored in the exe subdirectory of the directory (folder) containing the BASIC program. This ensures that it is safe to run the executable file without any risk of embedded files, when extracted, overwriting wanted files in the source directory.

A default icon is used unless the specified executable file already exists, in which case the icon is taken from that file. You can change the icon by using the REM!Icon compiler directive or by clicking on the Change Icon button which calls up the
Change Icon dialogue box:

You can choose an icon either from an existing executable file, or from a separate icon file (for example as produced by an icon editor). If a file contains a number of different icons, you can select the one you want to use. Please respect Copyright, and use only an icon which you have generated yourself or have legitimately obtained for this purpose.

If the selected icon contains multiple variants with different resolutions (e.g. 16x16, 32x32, 48x48) and/or different colour depths (e.g. 16 colours, 256 colours) all the variants will be stored in the compiled executable file. The version used will be the one most suited to the current display settings. (This only works when BBC BASIC is running under Windows NT4.0™ or later; under Windows 95, 98 or Me a 32x32 16-colour icon is stored).

You can choose to **digitally sign** your executable file, if you have a Code Signing Certificate. For this to work, information on the code-signing tool and its parameters must be stored in the Registry, and unless present the checkbox is disabled.

You can choose to **encrypt** your program and any other embedded files (libraries,
fonts, bitmaps etc.); this will make it very difficult for somebody to discover their contents by examining the executable file. However the encryption is not highly secure and you should not rely on it to protect sensitive data.

Selecting the 'encrypt' option also causes your program and any embedded files to be **compressed** in the executable, which can result in a significant reduction in the file size.

You can choose to create a **console** application rather than a **GUI** application. A console application is intended to be run from a command prompt and usually won't have a visible window (with *BBC BASIC for Windows* a window is always created, but normally it will be hidden). To arrange that console input/output is used by your program it should have the following code near the beginning:

```basic
SYS "GetStdHandle", -10 TO @hfile%(1)
SYS "GetStdHandle", -11 TO @hfile%(2)
SYS "SetConsoleMode", @hfile%(1), 0
*INPUT 13
*OUTPUT 14
```

The program should exit (whether normally or as the result of an error) using QUIT, optionally followed by an exit code. You should always incorporate an ON ERROR statement since otherwise errors will be reported on the (hidden) output window and not be visible.

**Appearance**

You can set the **initial width and height** of your program's window, in pixels, by entering the appropriate values in the Compile dialogue box. There is little point in doing this if your program uses the MODE statement to initialise the screen, but otherwise it can be useful to set the window size to suit your application. A width or height value of zero causes Windows to select a default size.

You can choose whether the dimensions you specify apply to the **entire** window or only to the **client** area, i.e. that region of the window used for BASIC's output. The latter will often be more useful.

The **initial window state** can be selected from the drop-down list as Hidden, Normal, Maximised or Minimised. If Maximised is selected, the program will start with the window as large as possible (irrespective of the specified width and height). If Minimised is selected, the program will start with no window (just an icon on the
Taskbar). If Hidden is selected your program will neither have a visible window nor an icon in the Task Bar; this is useful for programs which operate in the background and for console mode programs. In all three cases you should normally avoid the use of the MODE statement in your program.

You can enable the use of Windows XP™ Visual Styles by your compiled program. If this option is selected, and your program is run under Windows XP or later (with the default appearance settings), toolbars, status bars, dialogue boxes and other window furniture will have a more colourful and modern style. This selection also improves compatibility with Windows Vista™, Windows 7™, Windows 8/8.1™ and Windows 10™.

Crunch

You can choose to crunch your program when it is incorporated into the executable file. This reduces the size of the file, often by a significant amount, and increases the speed of execution of your program somewhat. It also makes it more difficult for somebody to discover how your program works. The following options are provided:

- **Discard spaces**. All superfluous spaces in your program are removed.
- **Concatenate lines**. Short lines are concatenated (joined end-to-end) to make fewer long lines, where this is possible.
- **Remove REMs**. All REM statements are removed (unless referenced by a GOTO, GOSUB or RESTORE statement).
- **Abbreviate names**. Variable, procedure and function names are replaced with short names, distributed across the alphabet.

When any of these options are selected, blank lines are also removed from your program.

When using the Remove REMs and Abbreviate names options you can include verbose comments in your program, and use long descriptive variable names, without the size or performance of the final compiled program being affected. In this way you can accord with good programming practice and make your 'source' program easier to understand and modify.

Although you can normally 'crunch' a program without it having any unwanted side
effects, there are a few circumstances when it is incompatible with correct program operation:

- If the program relies on variables having a specific name, for example a variable is referenced by name in a DATA statement or in an EVAL function, you should either disable the **Abbreviate names** option or use the **REM!Keep** compiler directive.

- If the program performs calculations on line numbers to determine the destination of a GOTO, GOSUB or RESTORE, the **Concatenate lines** option should not be used.

If the crunch operation reports an error the offending statement will be highlighted.

**Embedded files**

If your program uses the CALL or INSTALL statement to load modules, or loads other files at run time (such as graphics files, sound files, font files etc.) you must ensure that the files are available to the compiled program where it expects to find them. This is particularly important if you want to run the program on a computer on which **BBC BASIC for Windows** is not installed. You can achieve this by putting the files either in the library directory (as identified by the @lib$ system variable) or in the same directory as the program itself (as identified by the @dir$ system variable) and loading them as follows:

```
INSTALL @lib$+"LIBFILE.BBC"
OSCLI "PLAY """"+@dir$+"MIDFILE.MID"
SYS "AddFontResource", @dir$+"BBCWIN.FON"
```

Files loaded this way will automatically be incorporated into the compiled executable file and copied to appropriate locations on the target machine. Files in the library directory will be deleted from the target machine after the program has completed execution; files in the program's directory will be left there.

You can also manually edit the list of embedded files using the **Add @dir$**, **Add @lib$** and **Remove** buttons. The Remove button deletes all selected (highlighted) entries in the list. You can select multiple files for addition and removal, using the Shift and Ctrl keys in the usual way (there is a limit to the number of files that can be added **in one go**), but more can be added by clicking on the **Add** button again. If the specified executable file already exists, the list of embedded files is read from that,
so any previous edits will be remembered.

Note that it is not intended that this facility should substitute for a proper installer. If you need to do anything more complicated on installation than copying 'resource' files to the destination PC you should consider using an installer utility. For example see the free Inno Setup program.

You can choose to crunch embedded BBC BASIC (tokenised) program files, as well as the main program. If this option is selected any embedded files with a .BBC extension are crunched as if they were part of the main program. The same crunch options are used, and in the case of the abbreviate names option the same abbreviations are used. This ensures that any variables or structures shared between the main program and external modules, imported using INSTALL or CALL, will remain consistent. You must ensure that all imported modules are embedded, otherwise function and procedure names will not be consistent (unless you use the REM!Keep compiler directive to prevent the names being abbreviated).

Embedded files may come from the @dir$ or @lib$ folder, or a subfolder of either of these. When the executable program is run the necessary subfolder(s) will be created and the file(s) copied there. This can be useful if your program uses many resource files, perhaps sound files or bitmap files, and you prefer not to clutter up your program's folder with them. By creating subfolders called, for example, SOUNDS and IMAGES you can keep your files organised. In your program you could access these files as follows:

```
OSCLI "PLAY "+@dir$+"SOUNDS\TUNE1.MID"
OSCLI "DISPLAY "+@dir$+"IMAGES\PICTURE.BMP"
```

**Compiler directives**

If an existing executable file, from a previous compilation, is found at the specified location it is used to initialise the compiler options. Alternatively you can incorporate compiler directives in your BASIC program, using specially formatted REM statements.

The compiler directives must be entered exactly as shown; they are case-sensitive. They may be included anywhere in your program. In the following list square brackets indicate optional parameters:

```
REM!Exefile filename[,signed][,encrypt][,console]

This directive allows you to specify the path and filename for the executable program, if it is different from the default. You can also optionally specify that the file should be signed, its contents encrypted (and compressed), or that a console-mode executable should be created.

REM!Icon iconfile,index

This directive allows you to specify a file from which the executable's icon will be extracted. It may be either an executable program (.EXE) file, which can contain multiple icons (an index number of zero signifies the first icon in the file), or an icon (.ICO) file, in which case specify an index of zero.

REM!Window width,height[,client][,hidden][,minimised][,maximised][,xpstyle]

This directive allows you to specify the (initial) window dimensions, in pixels, and whether those dimensions apply to the entire window (the default) or the client area. You can also optionally specify the initial window state (hidden, minimised or maximised) and whether Windows XP visual styles should be used by the executable program. **It is important to include the xpstyle option if your program is intended to run under Windows Vista, Windows 7, Windows 8/8.1 or Windows 10.**

REM!Crunch [spaces][,lines][,rems][,names][,embedded]

This directive allows you to specify whether the program will be crunched, and if so which crunch options are used. You can also specify that any embedded .BBC (program or library) files should also be crunched.

REM!Embed [file1][, file2...]

Page  63
This directive allows you to specify which files should be embedded in the executable. The files must be specified in the form `@dir$+"filename"` or `@lib$+"filename"`. You may include more than one `REM!Embed` line if there are many such files, however you may not use wildcards. The presence of one or more `REM!Embed` directives overrides the selection of embedded files by other means, so every file that you need to be embedded must be specified using the directive.

- `REM!Keep var1[, var2, var3...]`

This directive allows you to specify a list of variable, array or structure names which will not be abbreviated, even when the `REM!Crunch names` option has been selected. This is useful if you refer to named variables in a DATA statement or using EVAL but you still want to abbreviate the remaining variable names for best performance. Note that such variable names must have at least three characters (excluding the type character, if any) to minimise the risk that they will clash with one of the automatically-generated abbreviated names. You may include more than one `REM!Keep` line if there are many such variables.

You can also specify that a procedure or function name should not be abbreviated:

```
REM!Keep PROCmyprocedure, FNmyfunction()
```

Note that in this context `PROC` and `FN` must be in capital letters, even if you use the Lowercase keywords option.

- `REM!Fast var1[, var2, var3...]`

**BBC BASIC for Windows version 6.00a or later only**

This directive has the same syntax as `REM!Keep`. It accepts a list of variable, array, structure, function and procedure names which will be accessed more quickly at run-time. If you have a speed-critical program you may be able to achieve a worthwhile benefit by listing those variables etc. which are accessed
frequently (the Profiler utility can be useful in identifying such variables). There is no effect on the execution speed when run from the IDE.

You must **not** include in the list the names of any structure members (even if they happen to share their name with a normal variable). Doing so will result in the program failing to run, but no warning is issued at the compilation stage.

```
REM!Resource [file1][, file2...]
```

This directive allows you to incorporate Windows Resources in the executable file. The resources must be specified in the form `@dir$+"filename"` or `@lib$+"filename"` and they must be in the standard `.res` file format. One file can contain any number of resources, but you may include more than one `REM!Resource` line if desired. Resources that you might want to include are typically **Version Information** and/or a custom **Manifest** (when adding a custom manifest deselect the Use Windows XP Visual Styles option).

Apart from `REM!Embed`, `REM!Fast`, `REM!Keep` and `REM!Resource` it is not useful to have more than one directive of the same kind; if you do the one that is *latest* in the program will be used. Generally the `REM!Exefile` directive (if any) should be the first, since the other compiler options will be initialised from the contents of a previously-compiled file at the specified location, if present.

Here is an example set of compiler directives:

```
REM!Exefile C:\Documents and Settings\RTR\Desktop\index.exe
REM!Icon C:\Program Files\Help Workshop\hcw.exe, 1
REM!Window 640, 480, client, xpstyle
REM!Crunch spaces, lines, rems, names, embedded
REM!Embed @dir$+"images\owl.bmp", @lib$+"WINLIB2"
REM!Fast speedy%, critical{}
REM!Keep header$, footer$
REM!Resource @dir$+"resources\manifest.res"
```

**Add-in Utilities**

Up to nine **add-in** (or **plug-in**) utilities may be available in this menu. The following are installed by default:
• Differences - show the differences between a previous and current version of a program.
• Macro Recorder - create keyboard shortcuts for use when developing programs.
• Memory Monitor - monitor your program's memory usage, and display memory contents.
• Module Viewer - view, trace and single-step through your INSTALLED modules.
• Cross Reference - report variable and function usage, and warn about possible errors.
• Profiler - optimise your program for speed without needing to run a separate tool.
• Version Info - add a VERSIONINFO resource to your compiled executable.
• Windows Constants - automatically add declarations to your program.
• My Utilities - manage and organise your add-in utilities.

More information on these utilities, as well as a list of additional utilities available for download, can be found here.
Options menu

The **options menu** allows you to configure *BBC BASIC for Windows* to suit your own preferences. These preferences are saved on exit from BASIC, and are re-established whenever you run BBC BASIC in the future. The options menu can be accessed by clicking on its name in the menu bar or by means of the keyboard shortcut Alt-O.

### Syntax Colouring

Clicking on **Syntax Colouring** toggles this feature on and off. When Syntax Colouring is enabled all the keywords, REMarks, text strings and line numbers in your program are highlighted in colour. You can change the colours by means of the Set Colours command.

When Syntax Colouring is disabled, all the program is listed in black.

### Lowercase Keywords

Clicking on **Lowercase Keywords** toggles on and off the acceptance of BASIC keywords in lower-case letters. When this mode is enabled all keywords are listed in lower-case (both on the screen and on the printer). A keyword will be accepted on input from the keyboard, or from a file, either in all capital letters or in all lower-case letters. A word consisting of a mixture of upper and lower-case letters will never be mistaken for a keyword.

When Lowercase Keywords mode is disabled, keywords are only accepted in all
capitals and are listed in capitals. This mode is the default and is compatible with earlier versions of BBC BASIC.

**Indentation**

Clicking on **Indentation** brings up the following sub-menu:

- ✔ Enable Indentation
- ✔ Indent FOR...NEXT
- ✔ Indent REPEAT...UNTIL
- ✔ Indent WHILE...ENDWHILE
- ✔ Indent CASE...ENDCASE
- ✔ Indent IF...THEN...ENDIF

Selecting each of the options causes the appropriate program blocks to be indented (by two spaces) with respect to the surrounding code. This can make the program structure much clearer, and make it easier to spot mistakes.

Be aware that if you contrive to have (for example) two NEXT statements which match the same FOR statement, or vice versa, the resulting indentation will not be correct. In a well-structured program you would hopefully never do this anyway!

**Unicode**

Clicking on **Unicode** alternately enables and disables support for Unicode and bi-directional text in string constants (i.e. enclosed in double-quotes) and REMarks (comments). When enabled, you can directly enter and edit non-ANSI text within the editor, even languages written right-to-left such as Arabic or Hebrew. This feature requires Windows 2000 or later.

Note that Unicode text is encoded as UTF-8 within your program; if you want to display it at run time you will either need to use the UTF-8 option provided by the VDU 23,22 command or use one of the Unicode-enabled Windows™ libraries (WINLIBU, WINLIB2U, WINLIB4U or WINLIB5U).

When a BASIC program is loaded the character encoding used is determined automatically (if possible) and the necessary selection made; if the selection was changed a message is displayed to inform you. If the program is not sensitive to encoding (only 7-bit ASCII characters are present in quoted strings) the selection is not changed. If the encoding cannot be determined (for example both ANSI and
UTF-8 strings appear to be present in the program) a warning is issued.

**Toolbar**

Clicking on **Toolbar** toggles the toolbar on and off. You can turn the toolbar off if you want to maximise the area available for the editing pane.

**Status Bar**

Clicking on **Status Bar** toggles the status bar on and off. You can turn the status bar off if you want to maximise the area available for the editing pane.

**Set Font**

The **Set Font** command allows you to select the font style and size for use in the editing pane. The command opens the Set Font dialogue box:

![Set Font Dialogue Box](image)

Normally all available fonts are listed; an optional registry setting causes only those fonts which have a fixed pitch (i.e. *monospaced* fonts) to be included. The default font is the System Fixed Font, which is selected by ticking the Use System Fixed
Font option.

**Set Printer Font**

The **Set Printer Font** command allows you to select the font style and size used when you print your program using the Print command.

Normally all available fonts are listed; an optional registry setting causes only those fonts which have a fixed pitch (i.e. *monospaced* fonts) to be included. If no printer font has been selected the screen font is used (if the System Fixed Font has been selected a default printer font is used).

**Set Colours**

The **Set Colours** command allows you to select the colours which will be used for highlighting the various syntactical elements in your program (keywords, strings, remarks etc.), if the Syntax Colouring option has been enabled. The command opens the Set Colours dialogue box:

Selecting the appropriate syntactical element causes the current colour for that element to be identified by means of a box around the colour sample. You can change the colour for that element simply by clicking on the appropriate coloured...
rectangle, and then clicking on OK or Apply (OK closes the dialogue box, whereas Apply leaves it open so that further changes can be made).

If the selection of 48 pre-defined colours is not sufficient, you can click on Define Custom Colour which allows you to choose a colour from the full range supported by your video hardware and settings.

Customize

The **Customize** command allows you to customize certain settings according to your personal preferences. The command opens the Customize dialogue box:

The **Default program** box allows you to enter the name of a BASIC program which will be automatically loaded every time *BBC BASIC for Windows* is started. To ensure that this works correctly whatever the current *(Start in )* directory, you should enter the full path to the program file (the box will scroll horizontally as required). Alternatively you can browse for the file by clicking on the button with three dots.

The **Initial user memory** box allows you to set the amount of memory available for loading your BASIC programs (it also determines the initial value of HIMEM when
a program is run). The default value is about two Megabytes (2097152 bytes) which should be large enough for all but the very largest programs. Increasing this value more than necessary will cause *BBC BASIC for Windows* to use extra memory and may affect performance if insufficient RAM is fitted in your PC; it also affects the memory requirements of compiled executables. For best performance always set this value to an exact multiple of four. For a change to take effect you must exit and re-start *BBC BASIC*.

The **Number of undo levels** setting determines how many editing operations you can reverse with the Undo command. Increasing this value can result in *BBC BASIC for Windows* using up much more memory and may affect performance if insufficient RAM is fitted in your PC.

The **Accept keyword abbreviations** setting controls whether the program editor will accept keyword abbreviations such as `P.` instead of `PRINT`. One reason why you might want to turn off this facility is that structure names may be mistaken for abbreviated keywords.

The **Maximum number of columns** setting determines the maximum width of the right-click context menu. You should set it so the menu always fits on your screen, with the display resolution you are using.

The **Maximum number of rows** setting determines the maximum height of the right-click context menu. You should set it so the menu always fits on your screen, with the display resolution you are using.

The **Include ON statements** setting controls whether ON CLOSE, ON ERROR, ON MOUSE, ON MOVE, ON TIME and ON SYS statements appear in the right-click context menu. In any case they only appear when at the very beginning of a program line.

The **Show FN/PROC parentheses** setting controls whether functions and procedures, which take parameters, are shown in the context menu with a pair of trailing parentheses (brackets). This can be useful when two functions or procedures with the same name are present, but one takes parameters and the other doesn't. Note that if the parameter list is separated from the FN/PROC name by one or more spaces, parentheses will not be shown.
Run menu

The run menu is concerned with running and debugging your BBC BASIC program. The run menu can be accessed by clicking on its name in the menu bar or by means of the keyboard shortcut Alt-R.

Run

The Run command (keyboard shortcut F9) executes the BASIC program currently in memory. It opens a new window where any output from the program appears, and into which you type any keyboard input the program requires. Error messages also appear in this window. If there is no program currently in memory, Run has the same effect as the Immediate command.

If the program is already running (i.e. its output window is open), Run stops the program and restarts it from the beginning (note the caveat listed under Stop about any ON CLOSE routine not being executed).

If the line numbers (if any) in your program are not in ascending sequence, an error message is displayed and the first out-of-sequence line number is highlighted. You must correct the line numbers, for example by using the Renumber utility, before you can run the program.

Debug

BBC BASIC for Windows version 6.00a or later only
The Debug command executes the BASIC program currently in memory but immediately pauses it, enables Trace mode and opens the List Variables window.
This can be useful when you want to debug the code from the very start rather than using the Run To Cursor command.

Stop

The **Stop** command stops the program currently running and closes its window. Stop will normally close the program immediately, whatever it is doing, but if your program is waiting for a Windows API (SYS) function to return, or is executing some assembler code, this may delay the response.

Note that any ON CLOSE routine in your program will not be executed. It is therefore better to use the normal methods of ending your program (e.g. clicking on the Close button) when possible and to use the **Stop** command only as a last resort.

If the program is not currently running (i.e. its output window is not open) the Stop command is inhibited.

Pause

The **Pause** command (keyboard shortcut F8) pauses the currently-executing BASIC program at the start of the next statement. If the program is currently waiting for input (e.g. is executing a GET or INPUT statement) it will enter the paused state once the input has been provided. When paused, the program will produce no output nor accept any input (keypresses will still be put in the keyboard buffer). You can find out where the program has paused with the Trace command and/or examine the current values of the program's variables with the List Variables command.

If the program is not currently running (i.e. its output window is not open) the Pause command is inhibited.

Step

The **Step** command (keyboard shortcut F7) allows you to single-step a paused program, one statement at a time. You can monitor the execution of the program with the Trace command and/or follow the changing values of the program's variables with the List Variables command.

If the program is not currently paused, the Step command is inhibited. If you want to
single-step through only a specific part of your program, add a TRACE STEP ON statement at that point. Alternatively use the Run To Cursor command.

**Step Line**

The **Step Line** command (keyboard shortcut F5) is similar to the Step command except that all the remaining statements in the current line (and the first statement in the next line) are executed before pausing. This is particularly useful in allowing you to 'step over' a procedure or function call without having to single-step through all its statements, or to 'step out' of a loop without having to single-step through every iteration.

If the program never executes a statement in the next line, it will not pause.

**Run To Cursor**

The **Run To Cursor** command (keyboard shortcut F6) behaves the same as the Run command except that the program automatically pauses (as if the Pause command was issued) when it reaches the line containing the text cursor (caret). This is useful if you need to single-step through part of the program but don't want to start at the beginning.

The first (or only) statement in the line containing the cursor is executed before the program pauses. If the program never executes a statement in the line containing the cursor, it will not pause. Once the program is paused, issuing the **Run To Cursor** command again (or pressing F6) will cause execution to continue from that point; the program will pause again if there are more statements in the line, or if the line is reentered later.

Note that the program will not pause on any of the following lines:

- A blank line.
- A REM statement.
- A 'star' command .
- A CALL filename$ statement.
- A GOTO or GOSUB statement.
- An IF statement where the condition is false and there is no ELSE clause.
Immediate Mode

The **Immediate Mode** command opens the output window but does not execute the BASIC program; instead the BASIC command prompt is displayed. At this prompt you can type BASIC statements which are executed immediately on pressing Return (Enter). For example you can use this facility as a calculator by typing PRINT followed by a sum:

>`PRINT 2+2`

Immediate Mode can also be useful for doing things like deleting a file (*DELETE*), or setting a file to read-only status (*LOCK*). When in immediate mode you can execute the current program by typing RUN. If the current program is already running (i.e. its output window is open) the Immediate command will stop the program before entering immediate mode.
Help menu

The help menu gives access to the BBC BASIC for Windows documentation, provides links to information via email or from the web, and displays version information. The help menu can be accessed by clicking on its name in the menu bar or by means of the keyboard shortcut Alt-H.

Help Topics

The Help Topics command calls up the main help window, from which you can browse or search the BBC BASIC for Windows Help documentation. This command can also be accessed by pressing F1.

Tutorial

The Tutorial command opens the beginners' tutorial in a separate window.

Email

The Email command calls up your default mail program (if any) so that you can send a message to request help or information about BBC BASIC.

Website

The Website command calls up your default web browser (if any) and automatically directs it to the R.T.Russell home page.

Discussion Group
The Discussion Group command takes you to the online BB4W discussion group, where you can seek help and advice, report bugs, download (or upload) useful utilities and learn about new developments in *BBC BASIC for Windows*.

**Message Board**

The Message Board command takes you to the BBC BASIC Conforums message board, where you can discuss with other users all aspects of programming in *BBC BASIC for Windows*, including the use of advanced techniques.

**Facebook Page**

The Facebook Page command takes you to the BBC BASIC Group on Facebook, where you can ask questions or post pictures and videos.

**Wiki**

The Wiki command takes you to the BB4W Programmers' Reference Wiki. This is an ever-growing resource of articles on all aspects of programming in *BBC BASIC for Windows*.

**About BBC BASIC**

The About command displays the version number of *BBC BASIC for Windows* and (in the case of the full version) your personal serial number.
The editing pane

The **editing pane** is where the BASIC program currently loaded into memory (if any) is displayed for viewing or editing, or where a new program is entered. Most of the normal features which you expect from a Windows™ editor are available. You can manipulate and edit the program using the keyboard, mouse and touch screen (if applicable).

**Keyboard commands**

The following operations are available using the keyboard. Note that references to **left** and **right** apply to normal left-to-right text, and may be reversed when editing right-to-left text such as Arabic or Hebrew:

<table>
<thead>
<tr>
<th>Key</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>Toggle between insert and overtype mode. In overtype mode any character typed will over-write the existing character at that position on the line. In insert mode any character typed will 'push' the rest of the line to the right to make room. The insert/overtype status is indicated in the status bar.</td>
</tr>
<tr>
<td>Delete</td>
<td>This key has an identical effect to the Delete command. If any text is selected (highlighted) it is deleted, without being copied to the clipboard. Otherwise, the character to the immediate right of the cursor (caret) is deleted and the rest of the line is moved left to fill the gap.</td>
</tr>
<tr>
<td>Home</td>
<td>Move the cursor (caret) to the start of the current program line, and scroll the display horizontally (if necessary) to ensure that the cursor is visible.</td>
</tr>
<tr>
<td>Ctrl+Home</td>
<td>Move the cursor (caret) to the start of the program, and scroll the display horizontally and/or vertically (if necessary) to ensure that the cursor is visible.</td>
</tr>
<tr>
<td>End</td>
<td>Move the cursor (caret) to the end of the current program line, and scroll the display horizontally (if necessary) to ensure that the cursor is visible.</td>
</tr>
<tr>
<td>Ctrl+End</td>
<td>Move the cursor (caret) to the end of the program, and scroll the display horizontally and/or vertically (if necessary) to ensure that the cursor is visible.</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Page Up</td>
<td>Display the previous 'page' of the program by scrolling the display down by one screenful (less one line) or until the first line of the program is visible, whichever is less. If the first line of the program is already in view, Page Up has no effect.</td>
</tr>
<tr>
<td>Page Down</td>
<td>Display the next 'page' of the program by scrolling the display up by one screenful (less one line) or until the last line of the program is visible, whichever is less. If the last line of the program is already in view, Page Down has no effect.</td>
</tr>
<tr>
<td>←</td>
<td>Move the cursor (caret) one character to the left, and scroll the display horizontally (if necessary) to ensure that the cursor remains visible. If Shift is held down, select (or de-select) a character. If the cursor is already at the beginning of a line, the key has no effect.</td>
</tr>
<tr>
<td>→</td>
<td>Move the cursor (caret) one character to the right, and scroll the display horizontally (if necessary) to ensure that the cursor remains visible. If Shift is held down, select (or de-select) a character.</td>
</tr>
<tr>
<td>↑</td>
<td>Move the cursor (caret) up one line, and scroll the display vertically (if necessary) to ensure that the cursor remains visible. If Shift is held down, select (or de-select) a line. If the cursor is already on the first line, the key has no effect.</td>
</tr>
<tr>
<td>↓</td>
<td>Move the cursor (caret) down one line, and scroll the display vertically (if necessary) to ensure that the cursor remains visible. If Shift is held down, select (or de-select) a line. If the cursor is already on the last line, the key has no effect.</td>
</tr>
<tr>
<td>Backspace</td>
<td>Delete the character immediately to the left of the cursor (caret), and move the cursor left one position. The rest of the line, to the right of the cursor, is moved left to fill the gap. If the cursor is at the beginning of a line, delete the preceding 'new line' (and the line number, if any) and concatenate the contents of the present line onto the end of the previous line.</td>
</tr>
</tbody>
</table>
| Return (Enter) | Insert a 'new line' at the current position of the cursor (caret). Everything to the right of the cursor will be moved onto the next line, and the rest of the program is moved down one line to make space. By this means you can split an existing line into two or
<table>
<thead>
<tr>
<th>Action</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl+←</td>
<td>Move the cursor (caret) one <strong>word</strong> to the left, and scroll the display horizontally (if necessary) to ensure that the cursor remains visible. If <strong>Shift</strong> is also held down, select (or de-select) a word.</td>
</tr>
<tr>
<td>Ctrl+→</td>
<td>Move the cursor (caret) one <strong>word</strong> to the right, and scroll the display horizontally (if necessary) to ensure that the cursor remains visible. If <strong>Shift</strong> is also held down, select (or de-select) a word.</td>
</tr>
<tr>
<td>Tab</td>
<td>Normally has no effect, but can be programmed to execute a macro. See the <strong>SETTAB</strong> program supplied.</td>
</tr>
<tr>
<td>Ctrl+Backspace</td>
<td>Delete the <strong>previous word</strong> (i.e. to the left of the cursor) and move the rest of the line left to fill the gap.</td>
</tr>
<tr>
<td>Ctrl+Delete</td>
<td>Delete the <strong>next word</strong> (i.e. to the right of the cursor) and move the rest of the line left to fill the gap.</td>
</tr>
</tbody>
</table>

The 'printing' keys (letters, numbers and symbols) cause the appropriate character to be entered at the position of the cursor (caret), and the cursor is then moved right by one position. If anything is selected (highlighted) when the key is pressed, it is first deleted.

**Mouse commands**

The following operations are available using the mouse:

<table>
<thead>
<tr>
<th>Action</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left click</td>
<td>Clicking in the editing pane with the left mouse button causes the text cursor (caret) to be moved to the character position nearest to the mouse pointer. If <strong>Shift</strong> is held down, the region between the previous position of the text cursor and the new position will be selected. If <strong>Ctrl</strong> is held down the entire line containing the text cursor will be selected.</td>
</tr>
<tr>
<td>Left double-click</td>
<td>Double-clicking in the editing pane with the left mouse button causes the word under the mouse pointer (and the following space, if any) to be selected. The text cursor (caret) is moved to the end of the selected region. The selected word can then be cut, copied or deleted. If the Find or Replace command is issued, the selected word is copied to the Find What field.</td>
</tr>
<tr>
<td>Left drag</td>
<td>Holding down the left mouse button and 'dragging' the mouse pointer...</td>
</tr>
</tbody>
</table>
over the program displayed in the editing pane causes a section of the program to be selected. Dragging the mouse horizontally selects part or all of a line; dragging the mouse vertically selects a block of lines (only entire lines may be selected).

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Right click</td>
<td>Clicking in the editing pane with the right mouse button causes the Context Menu to be displayed.</td>
</tr>
<tr>
<td>Wheel</td>
<td>Operating the mouse wheel causes the editing pane to scroll vertically, usually by one line per 'notch'.</td>
</tr>
<tr>
<td>Ctrl+Wheel</td>
<td>Holding down the Ctrl key and operating the mouse wheel causes the editing pane to zoom; note that this will not work if you have selected the System Fixed Font.</td>
</tr>
</tbody>
</table>

**Drag and Drop**

If you move the mouse pointer over a selected (highlighted) region of your program, the pointer shape changes to an arrow. In this situation holding down the left mouse button and 'dragging' the mouse pointer initiates a Drag and Drop operation. If you drag the mouse pointer to a different point in your program (outside the selected region), and then release the left button, the selected text will be deleted from its original place and inserted where the button was released. This is equivalent to a cut-and-paste operation via the clipboard.

Alternatively, if you hold down the Ctrl key before releasing the left button the selected text is copied to the location where the button was released (a plus sign appears next to the mouse pointer indicating a copy rather than a move). This is equivalent to a copy-and-paste operation via the clipboard. If you want to cancel the Drag and Drop operation you can do that by pressing the ESCape key before you release the left button, or by releasing the button when the pointer is inside the selected area.

You can also Drag and Drop between BBC BASIC for Windows and any other program supporting that facility. You will often find that a plus sign is shown next to the mouse pointer, even if you are not holding down the Ctrl key. That signifies that you cannot delete the text in the window from which you are dragging it (for example if you drag program examples from the BBC BASIC for Windows Help window).

**Context Menu**
Clicking in the editing pane with the right mouse button causes the 'floating' context menu to be displayed at the position of the mouse pointer. The context menu contains one or more of the following items:

- If any part of the program is selected, the context menu contains the Cut, Copy and Add REMs commands. Clicking on Cut or Copy has an identical effect to that obtained by selecting the command from the edit menu, clicking on the appropriate button on the toolbar or using the associated keyboard shortcut (Ctrl+X and Ctrl+C respectively).

- Clicking on Add REMs automatically adds a REM to the start of each line in the selected block. If every line already starts with a REM, the context menu contains the Remove REMs command.

- If there is any text on the clipboard, the context menu contains the Paste command. Clicking on this command has an identical effect to that obtained by selecting the command from the edit menu, clicking on the appropriate button on the toolbar or using the associated keyboard shortcut (Ctrl+V).

- The context menu always includes the "What's This?" item. If the mouse pointer was positioned over a BASIC keyword when the context menu was called up, clicking on this item causes a short description of the use and function of that keyword to be displayed. The effect is the same as clicking the button on the toolbar and then clicking on the keyword.

- If a 'jump' to a function or procedure (see below) was previously made, the context menu contains the "Go back" item. Clicking on this item scrolls the display to the position it had before the last jump was made.

- If the current BASIC program contains any user-defined functions or procedures, they are listed in the context menu (in the order in which they occur in the program). Clicking on one of the function or procedure names causes the display to 'jump' (scroll) so that the first line of that function or procedure is at the top of the editing pane (or until the last line of the program is scrolled into view). This provides a quick way of navigating your program between functions and procedures.

As well as procedure and function names, the list optionally includes lines beginning with ON CLOSE, ON ERROR, ON MOUSE, ON MOVE, ON TIME or ON SYS. This feature can be enabled or disabled with the Customize
command.

- If you right-clicked on PROC or FN the procedure or function name is included near the beginning of the menu. Selecting this item causes the display to 'jump' (scroll) directly to that procedure or function. This can be easier than finding the name in the list.

- If the context menu gets too large, it is automatically split into two or more pages. Selecting Previous... or More... navigates between the pages. The maximum number of rows and columns is set by the Customize command.

If you have a touch screen you can display the context menu by pressing-and-holding until the square or circle appears. You can also access it by pressing Shift+F10.

### Scroll bars

If the current program cannot be displayed in its entirety within the editing pane, vertical and/or horizontal scroll bars are displayed. These allow you to scroll the display until the part of the program in which you are interested is in view. The scroll bars can be operated in the following ways:

- Dragging the scroll box (thumb) along the bar causes the display to scroll accordingly. The size of the scroll box indicates the proportion of the entire program which is displayed within the editing pane - the larger the scroll box the more of the program which is visible.

- Clicking on one of the arrow buttons causes the display to scroll in units of one character (horizontally) or one line (vertically).

- Clicking in the shaded regions between the scroll box and the arrow buttons causes the display to scroll by one page (less one line) vertically, or by eight characters horizontally.

- If your mouse has a 'wheel', rotating the wheel causes the display to scroll vertically, usually by one line per 'notch'.

- If you have a touch screen, you can scroll by 'flicking' in the appropriate direction. Alternatively you can scroll vertically by dragging with a single-finger and horizontally by dragging with two-fingers.
Registry settings

The following custom registry settings, which are not accessible via the user interface, are recognised by the IDE. They must be stored in `HKCU\Software\RT Russell\BBC BASIC for Windows\Settings` (or `HKCU\Software\Wow6432Node\RT Russell\BBC BASIC for Windows\Settings` in the case of 64-bit Windows):

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Data</th>
</tr>
</thead>
</table>
| ConfirmOnExit   | REG_BINARY | 00: The IDE will not prompt for confirmation on exit (unless the program is unsaved).  
                 |             | 01: The IDE will prompt for confirmation on exit if the output window is open (default). |
| UsePrinter      | REG_BINARY | 00: No check will be made for the default printer when the output window is opened.  
                 |             | 01: The default printer will be queried when the output window is opened (default). |
| FixedPitchOnly  | REG_BINARY | 00: The **Set Font** and **Set Printer Font** options will list all available fonts (default).  
                 |             | 01: The **Set Font** and **Set Printer Font** options will list only monospaced fonts. |
| SignTool        | REG_SZ     | The full path and filename of your Authenticode code-signing tool (if any). |
| SignParm        | REG_SZ     | The parameters required by your code-signing tool (use "%s" for the EXE filename). |
Line numbers

Line numbers are optional in *BBC BASIC for Windows*. A line must be numbered (or labelled) if it is referenced by a GOTO, GOSUB or RESTORE statement but otherwise the line number may be omitted. It can also be useful to number lines when debugging a program, since error messages can contain the number of the line in which the error occurs.

It is *never* necessary to use line numbers. Instead of GOSUB you should use named user-defined procedures or functions. In the case of RESTORE you can use the relative option of the statement. You can avoid GOTO by making use of structures like REPEAT ... UNTIL and WHILE ... ENDWHILE and using the techniques described under Program Flow Control.

The Renumber utility can be used to add or remove line numbers automatically. Permissible line numbers are in the range 1 to 65535.
Statement separators

If you want to write more than one statement on a line, the statements should be separated by a colon (:). *BBC BASIC for Windows* will sometimes tolerate the omission of the colon if this does not lead to ambiguity, but it's safer to leave it in and the program is easier to read.

For example, the following will work:

```
FOR i=1 TO 5 PRINT i : NEXT
```

but it's better to include the colon:

```
FOR i=1 TO 5 : PRINT i : NEXT
```

You may prefer to write each statement on a separate line. This will not impair the performance of a compiled program so long as you select the 'concatenate lines' Crunch option.
Line continuation

BBC BASIC's (tokenised) program lines are limited to 251 characters, excluding the line number. Although this is usually adequate, occasionally you may want to use longer lines or to split a program line over several screen lines for improved readability. You can do that by using the line continuation character \ (backslash) as follows:

```plaintext
PROC_render(pdev%, \ pointer to Direct3D device
\   nobj%, \ number of objects
\   yaw(), \ array of yaw angles
\   pitch(),\ array of pitch angles
\   roll()  \ array of roll angles
\ )
```

Here a single procedure call has been split between several lines for reasons of clarity and to allow the various parameters to be commented. Everything after the continuation character is ignored by BASIC when the program is executed so comments can conveniently be put there. Note that the continuation lines themselves must begin with another \ character.

You can use the line continuation character almost anywhere that a space would be allowed, but not:

- within a quoted (constant) string
- in a star command (use OSCLI instead)
- in a REM statement
- in a DATA statement
- immediately after the initial keyword of a statement
- when a single-line IF and a multi-line IF are combined.

You can use the continuation character in a single-line IF statement or in an ON statement (e.g. ON GOSUB, ON SYS) but you must not put anything (not even a comment) after the continuation character. If the IF or ON statement has an ELSE clause, the continuation character can only be used after the ELSE.
Expression priority

Order of evaluation

The various mathematical and logical operators have a priority order. BBC BASIC for Windows will evaluate an expression taking this priority order into account. Operators with the same priority will be evaluated from left to right. For example, in a line containing multiplication and subtraction, ALL the multiplications would be performed before any of the subtractions were carried out. The various operators are listed below in priority order (highest priority at the top):

```
variables functions () ! ? & % unary+- NOT
^ * / MOD DIV
+ - = <> <= >= > < << >> >>>
AND
EOR OR
```

Examples

The following are some examples of the way expression priority can be used. Including brackets even when they are not strictly necessary can make the intention clearer, but will slightly slow down program execution.

```
IF A=2 AND B=3 THEN
IF ((A=2)AND(B=3))THEN
IF A=1 OR C=2 AND B=3 THEN
IF (((A=1)OR((C=2)AND(B=3))))THEN
IF NOT(A=1 AND B=2) THEN
IF NOT(((A=1)AND(B=2)))THEN
N=A+B/C-D
N=A+(B/C)-D
N=A/B+C/D
N=(A/B)+(C/D)
```
Array arithmetic

A limited number of arithmetic and logical operations are available which operate on entire arrays rather than on single values. These are addition (+), subtraction (-), multiplication (*), division (/), logical OR, logical AND, logical exclusive-or (EOR), integer quotient (DIV), integer remainder (MOD) and the dot product (.). For example:

```
C%() = A%() * B%() + C%()
B() = A() - PI
A() = B() . C()
A$() = B$() + C$() + D$()
A() = B() * 2 / C() + 3 - A()
```

Array expressions are evaluated strictly left-to-right, so higher priority operators must come first and brackets cannot be used to override the order of execution:

```
C%() = C%() + B%() * A%() : REM not allowed
C%() = A%() * (B%() + C%()) : REM not allowed
```

All arrays must be DIMensioned before use, and (with the exception of the dot product) the number of elements in all the arrays within an expression must be the same; if not, the Type mismatch error will result. In the case of the dot product the rules are as follows:

- Both source arrays must be 1-dimensional or 2-dimensional.
- If the first array is 1-dimensional it is treated as a single row.
- If the second array is 1-dimensional it is treated as a single column.
- The number of columns in the first array must equal the number of rows in the second array.
- The total number of elements in the destination array must equal the product of the number of rows in the first array and the number of columns in the second array.

The MOD function returns the modulus (square-root of the sum of the squares of all the elements) of a numeric array, so an array can be normalised as follows:

```
A() = A() / MOD(A())
```
Don't confuse this with the MOD operator, which returns the remainder after an integer division.

You can use the compound assignment operators with arrays (e.g. +=, -=, *= or /=) but there is an important restriction. If the array is an integer (or byte) array the expression on the right-hand-side of the operator is converted to an integer before the operation is carried out. So the statement:

\[
A{%}() *= 2.5
\]

will multiply all the elements of array \(A{%}()\) by 2, not by 2.5.
Initialising arrays

You can initialise the individual elements of an array in exactly the same way as you would normal variables. However you can also initialise the contents of an entire array in one operation:

\[
\begin{align*}
A\%() &= 1, 2, 3, 4, 5 \\
B\$() &= "Alpha", "Beta", "Gamma", "Delta" \\
C() &= PI
\end{align*}
\]

If there are fewer values supplied than the number of elements in the array, the remaining elements are unaffected. However in the special case of a single value being supplied, all the elements of the array are initialised to that value. So in the last example all the elements of C() will be initialised to PI. There must be enough free space on the stack to contain a copy of the array.
Variables

Variable names

Variable names may be of any length, limited only by the maximum length of a line, and all characters are significant. They must start with a letter (A..Z, a..z) an underline (_) or a ` character (CHR$96). The remaining characters must be A..Z, a..z, 0..9, underline or ` (CHR$96). Variable names are case-sensitive ('number', 'Number' and 'NUMBER' are different variables). Variable names must not begin with a BASIC keyword (with some exceptions).

Using long, meaningful, variable names will make your program easier to understand and debug, and will not affect performance so long as the program is compiled with the abbreviate names crunch option.

Naming conventions

You are recommended to adopt the following conventions when naming variables:

- **Local** variables (e.g. variables with only a limited scope, formal parameters of functions and procedures, and variables declared in a LOCAL or PRIVATE statement) should be entirely in lower case (except for the static integer variables ). For example number%, text$.
- **Global** variables (i.e. those declared and used throughout your main program, and possibly accessed in functions or procedures) should use a mixture of lower case and capital letters. For example Click%, Title$.
- **Constants** (i.e. variables set to a constant value during initialisation and never changed again) should be in all capitals . For example MAXFILES, VERSION$.

Following these recommendations will make your programs easier to understand and avoid problems such as an interrupt routine erroneously accessing a local variable instead of a global variable.

Variable types

*BBC BASIC for Windows* uses the following types of variable:

- Static variables
• Integer numeric variables
• 64-bit integer numeric variables
• Byte numeric variables
• Variant numeric variables
• 64-bit floating-point numeric variables
• String variables
• Arrays
• Structures
• Pseudo-variables
• System variables

Note that in BBC BASIC the different types of variable are completely independent. For example, the integer variable `list%`, the numeric variable `list`, the string variable `list$` and the arrays `list%()`, `list()` and `list$()` are all entirely separate.

**Static variables**

The variables A%..Z% inclusive are called **static variables**. They are a special type of integer numeric variable which are not cleared by the statements RUN, CHAIN or CLEAR. As such, these variables can be useful for transferring values between programs or for holding information which must be retained when the rest of the variables are cleared. They are also accessed slightly more quickly than other variables.

In addition the variables A%, B%, C%, D%, F%, X% and Y% have special uses in the CALL and USR routines, and L%, O% and P% have special meaning in the assembler.

Static variables are stored in 32 bits and can contain any whole number from -2147483648 to +2147483647.

**Integer numeric variables**

Integer numeric variables have names which end with a percent sign (%). They are stored in 32 bits and can contain any whole number in the range -2147483648 to +2147483647. It is not necessary to declare a variable as an integer for advantage to be taken of fast integer arithmetic. For example, FOR ... NEXT loops execute at integer speed whether or not the control variable is an 'integer variable' (% type), so long as it has an integer value.
64-bit integer numeric variables

*BBC BASIC for Windows version 6.00a or later only*

64-bit integer numeric variables have names which end with a pair of percent signs (%%). They can contain any whole number in the range -9223372036854775808 to +9223372036854775807.

Byte numeric variables

Byte numeric variables have names which end with an ampersand sign (&). They are stored in 8 bits and can contain any whole number in the range 0 to +255 (byte variables are *unsigned*). Byte variables are particularly useful as structure members, because they can be used as building blocks to create data structures of any size.

Variant numeric variables

Variant numeric variables have no suffix character and can contain either integers or real (floating-point) values.

*BBC BASIC for Windows version 6.00a or later only*

Variant numeric variables can contain either a 64-bit signed integer value or an 80-bit (10-byte) floating-point value consisting of a 64-bit mantissa and a 16-bit exponent. They have a range of approximately ±3.4E-4932 to ±1.1E4932 and a precision of approximately 19 significant figures.

*BBC BASIC for Windows version 5.95a and earlier only*

Variant numeric variables come in two forms, depending on the *FLOAT mode in effect. In *FLOAT 40 mode they can contain either a 32-bit signed integer value or a 40-bit (5 byte) floating-point value consisting of a 32-bit mantissa and an 8-bit exponent. They have a range of approximately ±5.9E-39 to ±3.4E38 and a precision of approximately 9 significant figures.

In *FLOAT 64 mode they can contain either a 32-bit signed integer value or a 64 bit (8 byte) floating-point value consisting of a 53-bit mantissa and an 11-bit exponent. They have a range of approximately ±2.3E-308 to ±1.7E308 and a precision of approximately 15 significant figures.

An explanation of how variant numeric variables are stored is given in the Format of data in memory section.

64-bit ('double') floating-point variables
Double numeric variables have names which end with a hash sign (#); they contain a 64 bit (8 byte) floating-point value consisting of a 53-bit mantissa and an 11-bit exponent. They have a range of approximately ±2.3E-308 to ±1.7E308 and a precision of approximately 15 significant figures.

String variables

String variables have names which end in a dollar sign ($). In BBC BASIC for Windows version 5.95a or earlier their maximum length is 65535 characters; in BBC BASIC for Windows version 6.00a or later their length is limited only by the amount of available memory. An explanation of how string variables are stored is given in the Format of data in memory section.

Arrays

Arrays of variant numeric, double, integer, byte and string types are allowed. All arrays must be dimensioned before use. Variant numerics, doubles, integers, bytes and strings cannot be mixed in a multi-dimensional array; you have to use one array for each type of variable you need. A double array is indicated by a name ending in a hash sign (#), an integer array by a name ending in a percent sign (% or %%), a byte array by a name ending in an ampersand (&) and a string array by a name ending in a dollar sign ($).

The value given when the array is declared is the maximum value that the subscript can take. Since the minimum value for a subscript is zero, the total number of elements in that dimension is equal to the given value plus one. For example, the two-dimensional array declared with:

\[
\text{DIM marks\%}(10,5)
\]

has a total of 66 elements (10+1 rows by 5+1 columns).

Pseudo-variables

Pseudo-variables are keywords which behave like variables. They are LOMEM, HIMEM, PAGE, PTR and TIME/TIME$. Their values may be written (e.g. \(\text{TIME} = 0\)) or read (e.g. \(T = \text{TIME}\)) depending on the context. Pseudo-variables cannot be used as formal parameters of functions or procedures, cannot be made
LOCAL, cannot be used as the control variable of a FOR statement, cannot be passed as a parameter to CALL and cannot be assigned in INPUT, MOUSE, READ or SYS statements.

System variables

System variables have names which begin with the '@' sign. System variable names are predefined: you cannot create your own. The best known system variable - and the only one in the original version of BBC BASIC - is @%. This variable controls print formatting; see the description of the PRINT statement for details.

The other system variables are as follows. They are mostly of use when accessing Windows API functions from BASIC (see the section Accessing the Windows API for more information).

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>@hwnd%</td>
<td>The 'window handle' for BASIC's program (output) window</td>
</tr>
<tr>
<td>@memhdc%</td>
<td>The 'device context' for BASIC's screen memory</td>
</tr>
<tr>
<td>@prthdc%</td>
<td>The 'device context' for the current printer (if any)</td>
</tr>
<tr>
<td>@hcsr%</td>
<td>The handle for the mouse pointer (cursor)</td>
</tr>
<tr>
<td>@hpal%</td>
<td>The handle for the colour palette</td>
</tr>
<tr>
<td>@msg%</td>
<td>The MSG value (for use with ON MOUSE, ON MOVE and ON SYS)</td>
</tr>
<tr>
<td>@wparam%</td>
<td>The WPARAM value (for use with ON MOUSE, ON MOVE and ON SYS)</td>
</tr>
<tr>
<td>@lparam%</td>
<td>The LPARAM value (for use with ON MOUSE, ON MOVE and ON SYS)</td>
</tr>
<tr>
<td>@midi%</td>
<td>The MIDI device ID (non-zero if a MIDI file is playing)</td>
</tr>
<tr>
<td>@ispal%</td>
<td>A Boolean which is non-zero if the display is paletted</td>
</tr>
<tr>
<td>@hfile%(n)</td>
<td>An array of file handles indexed by channel number</td>
</tr>
<tr>
<td>@vdu%</td>
<td>A pointer to BASIC's text and graphics parameters (see below)</td>
</tr>
<tr>
<td>@cmd$</td>
<td>The command line of a 'compiled' program</td>
</tr>
<tr>
<td>@dir$</td>
<td>The directory (folder) from which your program was loaded</td>
</tr>
<tr>
<td>@hmdi%</td>
<td>The Multiple Document Interface window handle (if any)</td>
</tr>
<tr>
<td>@flags%</td>
<td>An integer incorporating BASIC's control flags. The MSB, if set, indicates that there is a pending Escape condition</td>
</tr>
</tbody>
</table>
@lib$ The directory (folder) containing the BBC BASIC library files
@ox% The horizontal offset (in pixels) between the output bitmap and the window contents. Must be zero or positive
@oy% The vertical offset (in pixels) between the output bitmap and the window contents. Must be zero or positive
@hwo% The handle of the WAVEOUTPUT device
@hevent% The handle of the event used to prevent blocking in serial and parallel I/O
@tmp$ The temporary directory (folder)
@usr$ The user's Documents directory (folder)
@vdu{} A structure containing the main VDU variables
@haccel% The handle of the keyboard accelerator, if used
@hwacc% The window handle to which keyboard accelerator commands should be sent

The @cmd$ variable

The variable @cmd$ allows an executable created with the Compile utility to access the command line with which it was executed. @cmd$ is empty in the case of a program run from the interactive environment.

The @dir$ and @lib$ variables

The variables @dir$ and @lib$ are useful when INSTALL ing libraries or when loading any other resource files (images, data files etc.) needed by your program. If you ensure that these files are stored either in the library directory or in the same directory as the program itself you can simply prefix their filenames with @lib$ or @dir$ respectively:

```
INSTALL @lib$+"MYLIB"
OSCLI "DISPLAY "+@dir$+"MYIMAGE"
SYS "PlaySound", @dir$+"MYSOUND", 0, &20001
```

Files specified this way are automatically incorporated into an executable created by the Compile command.

The @tmp$ and @usr$ variables
The variables @tmp$ and @usr$ are useful when storing data files. The first can be used for temporary files that may be discarded once the program has completed, and the second for files containing information relevant to the current user, such as saved output from the program. Unlike @dir$, these locations are guaranteed to be writable (in normal circumstances).

**VDU variables**

The variable @vdu% allows you to access the values of a number of BASIC's internal text and graphics parameters. Some of the more useful of these are listed below; in general you should only *read* these values (they should be *written* using the associated BASIC statements).

The main VDU variables may alternatively be accessed via the @vdu{} structure, as shown below:

<table>
<thead>
<tr>
<th>Value</th>
<th>Alternative</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>@vdu%!0</td>
<td>@vdu.o.x%</td>
<td>The horizontal graphics origin (in BBC BASIC graphics units), as set by ORIGIN or VDU 29.</td>
</tr>
<tr>
<td>@vdu%!4</td>
<td>@vdu.o.y%</td>
<td>The vertical graphics origin (in BBC BASIC graphics units), as set by ORIGIN or VDU 29.</td>
</tr>
<tr>
<td>@vdu%!8</td>
<td>@vdu.l.x%</td>
<td>The horizontal coordinate (in pixels) of the last point 'visited', as set by MOVE, PLOT etc.</td>
</tr>
<tr>
<td>@vdu%!12</td>
<td>@vdu.l.y%</td>
<td>The vertical coordinate (in pixels) of the last point 'visited', as set by MOVE, PLOT etc.</td>
</tr>
<tr>
<td>@vdu%!16</td>
<td>@vdu.p.x%</td>
<td>The horizontal coordinate (in pixels) of the previous point 'visited', as set by MOVE, PLOT etc.</td>
</tr>
<tr>
<td>@vdu%!20</td>
<td>@vdu.p.y%</td>
<td>The vertical coordinate (in pixels) of the previous point 'visited', as set by MOVE, PLOT etc.</td>
</tr>
<tr>
<td>@vdu%!24</td>
<td>@vdu.tl%</td>
<td>The left edge of the text viewport (in pixels), as set by VDU 28.</td>
</tr>
<tr>
<td>@vdu%!28</td>
<td>@vdu.tr%</td>
<td>The right edge of the text viewport (in pixels), as set by VDU 28.</td>
</tr>
<tr>
<td>@vdu%!32</td>
<td>@vdu.tt%</td>
<td>The top of the text viewport (in pixels), as set by VDU 28.</td>
</tr>
</tbody>
</table>
The bottom of the text viewport (in pixels), as set by VDU 28.

The width of a graphics 'dot', in pixels.

The height of a graphics 'dot', in pixels.

The horizontal text cursor position, in pixels.

The vertical text cursor position, in pixels.

The handle of the current font, as set by *FONT.

The handle of the current graphics clipping rectangle, as set by VDU 24.

The current graphics foreground action (ROP2 code), as set by GCOL.

The current graphics foreground colour, as set by GCOL.

The current graphics background action (ROP2 code), as set by GCOL.

The current graphics background colour, as set by GCOL.

The top line of the text cursor (caret), as set by VDU 23,0,10.

The bottom line of the text cursor (caret), as set by VDU 23,0,11.

The text foreground colour, as set by COLOUR.

The text background colour, as set by COLOUR.

The current MODE number (255 for a user-defined mode).

The colour mask for the current MODE (number of colours-1).

The VDU emulator flags byte (8 bits).

The scroll counter in VDU 14 paged mode.

The width, in pixels, of BASIC's output 'screen', as set by MODE or VDU 23,22.

The height, in pixels, of BASIC's output 'screen', as set by MODE or VDU 23,22.
The width, in pixels, of one character cell, as set by MODE or VDU 23,22.

The height, in pixels, of one character cell, as set by MODE or VDU 23,22.

The width of a character cell on the printer, as set by *PRINTERFONT.

The height of a character cell on the printer, as set by *PRINTERFONT.

The left edge of the printed page (left margin), as set by *MARGINS.

The right edge of the printed page, as set by *MARGINS.

The top edge of the printed page (top margin), as set by *MARGINS.

The bottom edge of the printed page, as set by *MARGINS.

The thickness, in pixels, of (solid) straight lines and outline shapes, as set by VDU 23,23.

The width, in pixels, of the text cursor (caret) as set by VDU 23,0,18.

The horizontal coordinate determining where text will appear on the page when sent to the printer.

The vertical coordinate determining where text will appear on the page when sent to the printer.

The current printer job ID (if any).

**Creation of variables**

Variables can be created in a number of ways:

- As the result of an assignment statement:

  ```
  Count% = 0
  name$ = ""
  Value += 1
  ```
• By making them LOCAL or PRIVATE variables in a function or procedure. In this case their values are initialised to zero or an empty string (in the case of PRIVATE, on first use only):

```plaintext
DEF PROC1 : LOCAL Count%, name$
DEF PROC2 : PRIVATE Count%, name$
```

• By using them as formal parameters in a function or procedure:

```plaintext
DEF PROC1(Count%, name$)
```

• By including them in the parameter list of a CALL statement:

```plaintext
CALL code, Count%, name$
```

• By assigning values to them with an INPUT, MOUSE, READ or SYS statement:

```plaintext
INPUT Count%
MOUSE x, y, b
READ name$
SYS "GetTickCount" TO Count%
```

• By passing them by reference to a function or procedure:

```plaintext
PROC1(Count%)  
DEF PROC1(RETURN C%)
```

• By using the address of operator:

```plaintext
SYS "GetFileSize", @hfile%(F%), ^sizehigh%
```
Structures

Structures are quite similar to arrays; both structures and arrays provide a means of grouping together several related variables. Whereas an array contains a number of elements of the same type, each identified by a numeric subscript (or subscripts), a structure contains a number of members (possibly) of different types, each identified by a name.

For example suppose you want to represent the position of an object in 3D space, which you can define by means of three numeric coordinates X, Y and Z. You could simply use three scalar variables, but it might be clearer or more convenient to group them together as a single entity describing the object position. One way of doing that would be to define an array with three elements:

```plaintext
DIM objpos(2)
objpos(0) = X
objpos(1) = Y
objpos(2) = Z
```

An alternative way would be to define a structure with three members:

```plaintext
DIM objpos(x,y,z)
objpos.x = X
objpos.y = Y
objpos.z = Z
```

In this case there is not much to choose between the alternatives, although the structure method does have the advantage of indicating the function of each member. However if the object description includes a name as well as a position then the array method cannot be used, because you can't store both a string and a number in the same array. In that case a structure is the answer:

```plaintext
DIM object{name$,pos{x,y,z}}
object.name$ = Name$
object.pos.x = X
object.pos.y = Y
object.pos.z = Z
```

**Structure members**
Structure members can be any of the normal variable types: variant numeric (having no suffix character), double numeric (indicated by a # suffix), integer numeric (indicated by a % or %% suffix), byte numeric (indicated by a & suffix), string (indicated by a $ suffix), array (variant numeric, double, integer, byte or string) or a sub-structure. A structure member is referenced by the name of the structure followed by a dot then the name of the member. With nested structures each sub-structure is separated by another dot:

```
structure_name.member_name
structure.substructure.subsubstructure.member
```

Normally a structure member can be treated in exactly the same way as a simple variable of the same type. However this is not so in the case of operations on entire arrays (e.g. SUM ) or entire structures (e.g. SWAP ) which cannot be used with structure members. Similarly you cannot pass an array member or a sub-structure member as the parameter of a procedure or function.

A byte-array structure member is treated as a special case. The construct `structure.array&()` is exactly equivalent to $$^structure.array&(0)$$, that is it refers to a NUL-terminated string stored in the array. This is convenient for certain structures used by the Windows™ API.

**Declaring a structure**

Structures, like arrays, are declared with DIM :

```
DIM mystuct{one,two%,three$,four(3),five%(1,2),six$(9),seven{a,b%})
```

This statement declares a structure called `mystuct` which contains the following members:
**mystruct.one**  A variant numeric value
**mystruct.two%**  An integer value
**mystruct.three$**  A string
**mystruct.four()**  An array of four variant numeric values
**mystruct.five%(())**  An array of six integer values (2 rows and 3 columns)
**mystruct.six$()**  An array of ten strings
**mystruct.seven{}**  A sub-structure containing the members `mystruct.seven.a` and `mystruct.seven.b%`

Note the use of brace characters ({{}}) rather than parentheses ((())), which are used when declaring an array.

**Arrays of structures**

As well as single structures, you can have arrays of multiple structures with the same format (i.e., each structure in the array has an identical set of members, but the values of the members are different for each array element). Arrays of structures are declared as follows:

```
DIM mystruct{(10)one,two%,three$}

or

DIM item{one,two%,three$}
DIM mystruct{(10)} = item{}
```

These declare an array of 11 structures (subscript 0 to subscript 10) each of which has three members. Individual members are referenced as follows:

```
mystruct{(1)}.one
mystruct{(5)}.two%
mystruct{(10)}.three$
```

Note that it is necessary to use both braces and parentheses to indicate to BASIC that you are referring to an element of an array of structures, rather than a simple array element. Arrays of structures can be multi-dimensional.
Structure prototypes

As well as the methods given in the previous two sections, you can declare structures, sub-structures and arrays of structures based on a previously declared structure. This is very useful in enabling you to declare several structures with identical layouts, or in using predefined structure declarations, for example in 'header' files executed using CALL.

The syntax is as follows:

```plaintext
DIM new_struct{} = proto_struct{}
DIM nest_struct{name$, pos{} = proto_struct{}}
DIM struct_array{(1,2)} = proto_struct{}
```

The first example declares a structure `new_struct{}` which has the same layout and member names as the structure `proto_struct{}`. The second example declares a structure `nest_struct{}` whose second member `pos` is a sub-structure with the same layout and member names as the structure `proto_struct{}`. The third example declares an array (with 2 rows and 3 columns) of structures, each element of which has the same layout and member names as `proto_struct{}`.

*Note:* You cannot use a LOCAL structure as a prototype for another structure.

Passing structures to procedures and functions

You can pass structure members to procedures and functions exactly as you would simple variables of the same type (with the exception of array and sub-structure members). However you can also pass entire structures to procedures and functions:

```plaintext
DIM objpos{x,y,z}
PRINT FN_distance(objpos{})
END
DEF FN_distance(s{})
= SQR(s.x^2 + s.y^2 + s.z^2)
```

Here the function `FN_distance` has been defined with a *formal* structure parameter `s{}` and has been called with the *actual* structure parameter `objpos{}`; note the use of opening and closing braces with nothing in between. In just the same way as passing an entire array to a procedure or function the parameter is passed *by reference* so any changes that take place to structure members within the procedure
or function affect the values of those members on return. See the section on Procedures and functions for more details.

You can also pass entire arrays of structures to procedures and functions:

```plaintext
DIM buffer{(1000) signal%, timestamp%}
PROC clear(buffer{()}, 0, 1000)
DEF PROC clear(B{()}, start%, finish%)
LOCAL I%
FOR I%=start% TO finish%
   B{(I%)}.signal% = 0
   B{(I%)}.timestamp% = 0
NEXT
ENDPROC
```

Be careful when passing structures to procedures or functions in libraries. If you crunch your main program with the Abbreviate names option, the names of the structure members will no longer agree with the names they have in the library, unless you use the crunch embedded files option.

**LOCAL and PRIVATE structures**

You can declare structures which are LOCAL or PRIVATE to a procedure or function. You do that using an equivalent syntax to that used for arrays:

```plaintext
DEF PROC test1
LOCAL locstruct{}
DIM locstruct{a,b,c}
DEF FN test2
PRIVATE pristruct{}
DIM pristruct{p%,q%,r%}
```

All members of a LOCAL structure are initialised to zero or an empty string. Members of a PRIVATE structure are initially zero or empty strings, but subsequently keep their values from one call of the procedure/function to the next.

**Returning structures from procedures and functions**

It is possible to declare a structure within a function or procedure and return that new structure to the calling code. To do that you must pass an undeclared structure as the parameter and use the RETURN keyword in the function or procedure definition:
If you pass a structure which already exists, it must have an identical format to the declaration within the function or procedure.

*Note:* You cannot use this technique to declare LOCAL structures.

### Using structures with the Windows API

Structures can often be useful when passing parameters to Windows™ Application Program Interface functions. For example the **ClientToScreen** API function uses a structure containing the X and Y coordinates of a point on the screen:

```plaintext
DIM pt{x%,y%}
pt.x% = clientX%
pt.y% = clientY%
SYS "ClientToScreen", @hwnd%, pt{}
screenX% = pt.x%
screenY% = pt.y%
```

In this case `pt{}` evaluates to the address of the structure in memory, so `pt.x%` is the same as `pt{!0` and `pt.y%` is the same as `pt{!4`.

In the case of a sub-structure the value returned is the offset from the start of its parent structure, so to discover the memory address of a sub-structure you must add the two together:

```plaintext
DIM window{hw%,rc{l%,t%,r%,b%})
window.hw% = @hwnd%
SYS "GetWindowRect", window.hw%, window()+window.rc()
```

If you need to create a structure member of an arbitrary size you can use a byte array. For example the Windows™ OSVERSIONINFO structure consists of five DWORDs (32-bit integers) followed by a 128-byte character array:
The address of the ServicePack member can be obtained as `osvi{}+20` or as `^osvi.ServicePack&(0)` . Its value as a NUL-terminated string can be obtained as `osvi.ServicePack&()`.

**User-defined data types**

Structures can be useful for creating data types which are not supported natively in *BBC BASIC for Windows*. For example BBC BASIC doesn't have a 16-bit (word) data type, but it is possible to create something similar using a structure containing two byte members:

```basic
DIM word{l&,h&}
```

Here the structure `word{}` consists of the least-significant byte `word.l&` and the most-significant byte `word.h&` . You could write a set of procedures for manipulating 16-bit values stored in such a structure.

**Finding the size of a structure**

You can discover the size (in bytes) of a structure or sub-structure as follows:

```basic
size% = DIM(struct_name{})
```

This can be particularly useful for some structures used by the Windows™ API which require the first member to contain the size of the structure:

```basic
DIM osvi{} = _OSVERSIONINFO{}
oovi.Size% = DIM(osvi{})
```

**Copying a structure**

You can copy the contents of one structure into another structure as follows:

```basic
dest{} = source{}
```
Note that BASIC only checks that the *sizes* of the structures are the same, not that their contents are compatible. Do *not* copy structures containing *string* members; since only the *string descriptor* is copied, not the string itself, you are likely to confuse BASIC and may even crash it.
Program flow control

Introduction

Whenever BBC BASIC for Windows comes across a FOR, REPEAT, WHILE, GOSUB, FN or PROC statement, it needs to remember where it is in the program so that it can loop back or return there when it encounters a NEXT, UNTIL, ENDWHILE or RETURN statement or when it reaches the end of a function or procedure. These 'return addresses' tell BBC BASIC for Windows where it is in the structure of your program.

Every time BBC BASIC for Windows encounters a FOR, REPEAT, WHILE, GOSUB, FN or PROC statement it 'pushes' the return address onto a 'stack' and every time it encounters a NEXT, UNTIL, ENDWHILE or RETURN statement, or the end of a function or procedure, it 'pops' the latest return address off the stack and goes back there.

Apart from memory size, there is no limit to the level of nesting of FOR ... NEXT, REPEAT ... UNTIL, WHILE ... ENDWHILE and GOSUB ... RETURN operations. The untrappable error message 'No room' will be issued if all the stack space is used up.

Program structure limitations

The use of a common stack has one disadvantage (if it is a disadvantage) in that it forces strict adherence to proper program structure. It is not good practice to exit from a FOR ... NEXT loop without passing through the NEXT statement: it makes the program more difficult to understand and the FOR address is left on the stack. Similarly, the loop or return address is left on the stack if a REPEAT ... UNTIL loop or a GOSUB ... RETURN structure is incorrectly exited. This means that if you leave a FOR..NEXT loop without executing the NEXT statement, and then subsequently encounter, for example, an ENDWHILE statement, BBC BASIC for Windows will report an error (in this case a 'Not in a WHILE loop' error). The example below would result in the error message 'Not in a REPEAT loop at line 460'.

400 REPEAT
410 INPUT "What number should I stop at", num
Leaving program loops

There are a number of ways to leave a program loop which do not conflict with the need to write tidy program structures. These are discussed below.

REPEAT ... UNTIL loops

One way to overcome the problem of exiting a FOR ... NEXT loop is to restructure it as a REPEAT ... UNTIL loop. The example below performs the same function as the previous example, but exits the structure properly. It has the additional advantage of more clearly showing the conditions which will cause the loop to be terminated (and does not require line numbers):

```pascal
REPEAT
  INPUT ' "What number should I stop at", num
  i=0
  REPEAT
    i=i+1
    PRINT i;
  UNTIL i=100 OR i=num
UNTIL num=-1
```

Changing the loop variable

Another way of forcing a premature exit from a FOR ... NEXT loop is to set the loop variable to a value equal to the limit value. Alternatively, you could set the loop variable to a value greater than the limit (assuming a positive step), but in this case the value on exit would be different depending on why the loop was terminated (in some circumstances, this might be an advantage). The example below uses this method to exit from the loop:

```pascal
REPEAT
  INPUT ' "What number should I stop at", num
  FOR i=1 TO 100
    PRINT i;
    IF i=num THEN
```
Using the EXIT statement

You can use the EXIT FOR statement to achieve the same effect as the first example, without requiring a GOTO:

```
REPEAT
    INPUT 'What number should I stop at', num
    FOR i=1 TO 100
        PRINT i;
        IF i=num THEN EXIT FOR
    REM More program here if necessary
NEXT i
UNTIL num=-1
```

Moving the loop into a procedure

A radically different approach is to move the loop into a user-defined procedure or function. This allows you to jump out of the loop directly:

```
DEF PROCloop(num)
    LOCAL i
    FOR i=1 TO 100
        PRINT i;
        IF i=num THEN ENDPROC
    REM More program here if necessary
NEXT i
ENDPROC
```

If you needed to know whether the loop had been terminated prematurely you could return a different value:

```
DEF FNloop(num)
    LOCAL i
    FOR i=1 TO 100
        PRINT i;
        IF i=num THEN =TRUE
    REM More program here if necessary
```

Local arrays

Since local variables are also stored on the processor's stack, you cannot use a FOR ... NEXT loop to make an array LOCAL. For example, the following program will give the error message 'Not in a FN or PROC':

```basic
DEF PROC_error_demo
FOR i=0 TO 10
   LOCAL data(i)
NEXT
ENDPROC
```

Fortunately, BBC BASIC for Windows allows you to make an entire array 'local' as follows:

```basic
DEF PROC_error_demo
LOCAL data()
DIM data(10)
ENDPROC
```

Note the use of an opening and closing bracket with nothing in between.
Indirection

Introduction

Many versions of BASIC allow access to the computer's memory with the PEEK function and the POKE statement. Such access, which is limited to one byte at a time, is sufficient for setting and reading screen locations or 'flags', but it is difficult to use for building more complicated data structures. The indirection operators provided in BBC BASIC for Windows enable you to read and write to memory in a far more flexible way. They provide a simple equivalent of PEEK and POKE, but they come into their own when used to build complicated data structures, pass data to or from Windows API functions or for use with machine code programs.

The indirection operators should be used with great care, since attempting to read from or write to an inappropriate address may well crash BBC BASIC for Windows. Generally, the address used should either be in an area of memory allocated with DIM or have been returned from a Windows API function.

There are four indirection operators:

<table>
<thead>
<tr>
<th>Name</th>
<th>Symbol</th>
<th>Data type</th>
<th>No. of bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query</td>
<td>?</td>
<td>Byte</td>
<td>1</td>
</tr>
<tr>
<td>Pling</td>
<td>!</td>
<td>Integer</td>
<td>4</td>
</tr>
<tr>
<td>Pipe</td>
<td></td>
<td></td>
<td>5, 8 or 10</td>
</tr>
<tr>
<td>Dollar</td>
<td>$, $$</td>
<td>String</td>
<td>1 to 65536</td>
</tr>
</tbody>
</table>

In BBC BASIC for Windows version 6.00a or later string indirection may access any amount of memory.

The ? operator

The query operator accesses individual bytes of memory. ?M means 'the contents of' memory location 'M'. The first example below writes &23 to memory location M, the second example sets 'number' to the contents of that memory location and the third example prints the contents of that memory location.
Thus, '?' provides a direct replacement for PEEK and POKE.

\[
\begin{align*}
?A=B & \text{ is equivalent to } \text{POKE A,B} \\
B=?A & \text{ is equivalent to } B=\text{PEEK(A)}
\end{align*}
\]

The range of values which may be stored is 0 to 255 (i.e. byte indirection is unsigned).

**Query as a byte variable**

A byte variable, '?count' for instance, may be used as the control variable in a FOR ... NEXT loop and only one byte of memory will be used.

```
DIM count% 0
FOR ?count%=0 TO 20
    --
NEXT
```

**The ! operator**

The pling (!) indirection operator accesses 4 bytes (32-bits) of memory. Thus,

\[
!M=&12345678
\]

would load

\[
\begin{align*}
&78 & \text{ into address M} \\
&56 & \text{ into address M+1} \\
&34 & \text{ into address M+2} \\
&12 & \text{ into address M+3.}
\end{align*}
\]

and

```
PRINT ~!M  \text{(print !M in hex format)}
```

would give
The range of values which may be stored is -2147483648 to +2147483647 (i.e. integer indirection is signed).

The | operator

The pipe (|) indirection operator accesses a floating-point value, which occupies 5 bytes (in *FLOAT 40 mode), 8 bytes (in *FLOAT 64 mode) or 10-bytes (in *FLOAT 80 mode) of memory. Thus,

```
|F% = PI
```

would load the floating-point representation of PI into addresses F% to F%+4, addresses F% to F%+7 or addresses F% to F%+9 depending on the *FLOAT mode in effect.

The $ operator

The string indirection operator ($) writes a string followed by a carriage-return into memory starting at the specified address. Do not confuse M$ with $M; the former is the familiar string variable whilst the latter means 'the string starting at memory location M'. For example,

```
$M="ABCDEF"
```

would load the ASCII characters A to F into addresses M to M+5 and &0D into address M+6, and

```
PRINT $M
```

would print

```
ABCDEF
```

The $$ operator

The indirection operator $$ operates like $ except that the string is terminated with NUL (&00) rather than carriage return (&0D). It is particularly useful when reading or writing strings through the Windows™ API. For example:
Use as binary (dyadic) operators

All the indirection examples so far have used only one operand. Provided the left-hand operand is a variable (such as 'memory') and not a constant, '?' and '!' can also be used as dyadic operators (in other words, they can be used with two operands). For instance, M?3 means 'the contents of memory location (M + 3)' and M!3 means 'the contents of the 4 bytes starting at (M + 3)'. In the following example, the contents of location (memory + 5) is first set to &50 and then printed:

```basic
memory?5=&50
PRINT memory?5
```

Thus,

```
A?I=B is equivalent to POKE A+I,B
B=A?I is equivalent to B=PEEK(A+I)
```

The two examples below show how two operands can be used with the byte indirection operator (?) to examine the contents of memory. The first example displays the contents of the first 13 bytes of the current BASIC program. The second example displays the memory contents for a variant numeric variable (see the section Format of data in memory).

```basic
memory=PAGE
FOR offset=0 TO 12
  PRINT ~memory+offset, ~memory?offset
NEXT
```

The memory address and the contents are printed in hexadecimal format.

```basic
NUMBER=0
DIM A% -1
REPEAT
  INPUT"NUMBER PLEASE "NUMBER
  PRINT ";
  FOR I%=2 TO 5
    NUM$=STR$~(A%?-I%)
```
Power of indirection operators

Indirection operators can be used to create special data structures, and as such they are an extremely powerful feature. For example, a structure consisting of a 10 character string, an 8 bit number and a reference to a similar structure can be constructed.

If M is the address of the start of the structure then:

\$M \quad \text{is the string}
M?11 \quad \text{is the 8 bit number}
M!12 \quad \text{is the address of the related structure}

Linked lists and tree structures can easily be created and manipulated in memory using this facility.

The ^ operator

You can discover the memory address at which a variable is stored using the 'address of' operator ^ . Once you know its address you can access the value of a variable by means of the appropriate indirection operator:

```
A% = 1234
PRINT !^A%
```

This will work for all types of variable (integer, floating-point, string, array etc.) but in the case of a normal string variable the address returned is not that of the first character of the string but of the 6-byte or 8-byte string descriptor (see the CALL statement for details of string descriptors). Therefore the address of the string itself is !^string$.

In the case of an array the address returned by ^array() is that of a pointer to the
array parameter block, therefore the address of the parameter block is \(^{\text{array}}()\). To obtain the address of the array data you should specify the name of the first element, e.g. \(^{\text{array}}(0)\).

Knowing the memory address of a variable may seem to be of little value but can be useful in special circumstances, particularly when calling Windows\textsuperscript{TM} API functions or in assembly language code. For example the ReadFile function requires you to pass the address of a variable in which the number of bytes read will be returned:

\begin{verbatim}
SYS "ReadFile", @hfile%(file%), store%, size%, ^bytesread%, 0
\end{verbatim}

In assembly language code you might want to copy the value of a BBC BASIC (integer) variable into one of the processor's registers:

\begin{verbatim}
mov eax,[^variable%]
\end{verbatim}

Note that the static variables A\% to Z\% occupy consecutive locations in memory, which can be handy when an API function requires the address of a structure rather than a simple variable. For example GetCursorPos requires the address of a \textit{POINT} structure in which it returns two values:

\begin{verbatim}
SYS "GetCursorPos", ^X%
\end{verbatim}

This will return the horizontal coordinate of the cursor in X\% and the vertical coordinate in Y\%.

Another use is to alter the byte-order of a variable, for example from \textit{little-endian} to \textit{big-endian}. The following code segment reverses the byte-order of the value stored in A\%:

\begin{verbatim}
SWAP ?(^A%+0), ?(^A%+3)
SWAP ?(^A%+1), ?(^A%+2)
\end{verbatim}
Operators and special symbols

The following list is a summary of the meaning of the various operators and special symbols used by *BBC BASIC for Windows*:

? An operator giving 8 bit indirection.
!
" A delimiting character for constant strings. Strings always have an even number of " in them. " may be introduced into a string by the escape convention "".
#
$ As a prefix indicates a file channel number (and is not optional). As a suffix indicates a 64-bit double variable.
% As a prefix indicates a 'fixed string' (the syntax $<expression> is used to position a string anywhere in memory, overriding the interpreter's space allocation). As a suffix on a variable name, indicates a string variable.
& As a prefix indicates a hexadecimal constant e.g. &EF. As a suffix on a variable name, indicates a byte (unsigned 8-bit) variable.
' A character which causes new lines in PRINT or INPUT.
( ) As a suffix on a variable name indicates an array. Objects in parentheses have highest priority.
{} Braces indicate a structure.
= 'Becomes' for LET statement and FOR ; 'result is' for FN ; relation of equal to.
- Negation and subtraction operator.
* Multiplication operator; prefix indicating operating system command (e.g. *DIR ).
: Multi-statement line statement delimiter.
Suppresses forthcoming action in PRINT, BPUT#, THEN and *RUN.
Comment delimiter in the assembler. Delimiter in VDU and INPUT.

+ Unary plus and addition operator; concatenation between strings.

, Delimiter in lists.

. Decimal point in real constants; abbreviation symbol on keyword entry;
introduce label in assembler; dot product of arrays; delimiter of
structure members.

< Relation of less than.

<< Left-shift operator (signed or unsigned).

> Relation of greater than.

>> Right-shift operator (signed).

>>> Right-shift operator (unsigned).

/ Division operator.

\ The line continuation character.

@ Prefix character for system variables.

<= Relation of less than or equal to.

>= Relation of greater than or equal to.

<> Relation of not equal to.

== Relation of equal to (alternative to =).

[ ] Delimiters for assembler statements. Statements between these delimiters
may need to be assembled twice in order to resolve any forward
references. The pseudo operation OPT (initially 3) controls errors and
listing.

^ Exponentiation (raise-to-the-power-of) operator; the address of operator.

~ A character in the start of a print field indicating that the item is to be
printed in hexadecimal. Also used with STR$ to cause conversion to a
hexadecimal string.
A delimiter in the VDU statement. A unary operator giving floating-point indirection.

+= Assignment with addition (A += B is equivalent to A = A + B)

-= Assignment with subtraction (A -= B is equivalent to A = A - B)

*= Assignment with multiplication (A *= B is equivalent to A = A * B)

/= Assignment with division (A /= B is equivalent to A = A / B)

AND= Assignment with AND (A AND= B is equivalent to A = A AND B)

DIV= Assignment with DIV (A DIV= B is equivalent to A = A DIV B)

EOR= Assignment with EOR (A EOR= B is equivalent to A = A EOR B)

MOD= Assignment with MOD (A MOD= B is equivalent to A = A MOD B)

OR= Assignment with OR (A OR= B is equivalent to A = A OR B)
Keywords

It is not always essential to separate keywords with spaces, for example `DEGASN1` is equivalent to `DEG ASN 1`: both the DEG and the ASN are recognized as keywords. However `ADEGASN1` is treated as a variable name: neither `DEG` nor `ASN` is recognized as a keyword. Nevertheless adding spaces between keywords makes your program easier to read, and will not impair performance so long as the program is compiled using the discard spaces crunch option.

In general variable names cannot start with a keyword, but there are some exceptions. For example the constant PI and the pseudo-variables LOMEM, HIMEM, PAGE, and TIME can form the first part of the name of a variable. Therefore PILE and TIMER are valid variable names although PI$ and TIME% are not.

37 out of the total of 138 keywords are allowed at the start of a variable name. They are shown in bold type in the table below:

<table>
<thead>
<tr>
<th>Keywords Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
</tr>
<tr>
<td>ATN</td>
</tr>
<tr>
<td>CHAIN</td>
</tr>
<tr>
<td>CLS</td>
</tr>
<tr>
<td>DEF</td>
</tr>
<tr>
<td>ELSE</td>
</tr>
<tr>
<td>ENVELOPPE</td>
</tr>
<tr>
<td>EVAL</td>
</tr>
<tr>
<td>FN</td>
</tr>
<tr>
<td>GOTO</td>
</tr>
<tr>
<td>INSTALL</td>
</tr>
<tr>
<td>LINE</td>
</tr>
</tbody>
</table>
MOD   MODE   MOUSE   MOVE   NEXT   NOT
OF    OFF    ON      OPENIN  OPENOUT OPENUP
OR    ORIGIN OSCLI   OTHERWISE PAGE   PI
PLOT  POINT( POS    PRINT  PRIVATE PROC
PTR   QUIT    RAD    READ   RECTANGLE REM
REPEAT REPORT RESTORE RETURN RIGHTS( RND
RUN   SGN     SIN     SOUND  SPC    SQR
STEP  STOP    STR$   STRING$( SUM    SWAP
SYS   TAB(    TAN     THEN   TIME   TINT
TO    TRACE   TRUE   UNTIL  USR    VAL
VDU   VPOS    WAIT   WHEN   WHILE  WIDTH
Debugging

"If debugging is the process of removing bugs, then programming must be the process of putting them in."

It is inevitable that programs will sometimes contain bugs. You may be able to write small programs which are error-free and work first time, but the larger your programs get the greater the likelihood that they will have errors. These fall into three categories:

- Syntax errors. This is where the code doesn't make sense to the BASIC interpreter, and often results from a typing mistake. For example you might type:

```basic
PRONT "Hello world!"
```

when what you meant was:

```basic
PRINT "Hello world!"
```

When the interpreter encounters the word `PRONT` it won't know what to do, and will report a Mistake error.

- Run-time errors. These are similar to syntax errors in that they trigger BASIC's error reporting mechanism, but whereas a syntax error will inevitably cause a failure as soon as the code is executed, run-time errors only occur when certain conditions are met. For example the code:

```basic
answer = A / B
```

is syntactically correct, and generally won't result in an error, but if variable `B` is zero it will result in the Division by zero error. It could also result in the Number too big error if `A` is very large and `B` is very small.

- Other errors. Unfortunately many programming errors do not trigger BASIC's error reporting mechanism, and therefore go undetected by the interpreter, but
result in your program producing incorrect output or failing to respond correctly to input. These can include some typing errors, errors of logic (incorrect processes take place, or take place in the wrong sequence) and computational errors (the processes themselves do the wrong things). In each case BASIC is doing what you said, but what you said is either not what you meant or doesn't have the outcome you wanted.

Errors of logic and computational errors are best avoided at the design stage, before you even start writing code, but this ideal situation is rarely achieved in practice.

*BBC BASIC for Windows* incorporates a number of features to make it easier to find and fix bugs.

**Syntax colouring**

By default the syntactical elements of your program (keywords, strings, comments etc.) are shown in different colours in the program editor. This allows you, at a glance, to spot some errors which would cause your program to fail, for example:

- Misspelt keywords (they will not be coloured).
- Variable names starting with a keyword (the keyword will be coloured).
- Mis-matched quotation marks (strings will be coloured incorrectly).

**Indentation**

By default looping and nesting constructs in your program are indented by the program editor. This allows you to quickly spot errors such as:

- Unpaired keywords, e.g. a FOR without a NEXT, or a REPEAT without an UNTIL.
- Multi-line IF statements with the THEN omitted.

**Error reporting**

If the interpreter detects an error condition whilst your program is running it will activate the built-in error handling mechanism. In addition, when *not* in trace mode, it will automatically scroll to and highlight the statement that triggered the error (unless it is in a CALLeD or INSTALLLeD module); this happens even if the error is trapped using ON ERROR.
If you hover the mouse over the highlighted statement, the error message will be displayed alongside in a *tooltip*. The statement remains highlighted so long as your program's output window remains open, or until you enter a command in immediate mode.

**Fast edit-run cycle**

*As BBC BASIC for Windows* is an interpreter (albeit a very fast one) it has the advantage of allowing you to edit and re-run a program *instantly*. After you have made a change to a program you don't have to wait for it to be re-compiled and re-linked before you can run it. Although trial-and-error debugging isn't to be encouraged, there are occasions when it is the most efficient way of solving a problem.

**Run-time monitoring**

You can monitor the execution of your program in two main ways: using the Trace facility and the List Variables facility.

In Trace mode, the program statement which is currently being executed is highlighted, and this highlight moves as your program is running (unless the statement currently being executed is in a CALLeD or INSTALLed module). If your program gets stuck in an 'infinite loop', you will be able to see what part of the program is being executed. If your program stops running because of an error (or an END or STOP statement) the last statement executed will remain highlighted for as long as the program's output window remains open, or until you enter a command in immediate mode.

If List Variables is enabled a window opens in which the values of the currently-defined variables are displayed (along with a list of arrays and function and procedure names). This list is updated as your program runs so you can monitor how the values change and possibly spot anomalous values. Note that variables are only listed once they have been accessed by your program (except for the static integer variables which are always listed).

**Single-step execution**

If you know (or suspect) the region of your program in which an error is occurring,
but it happens too quickly to spot using Trace or List Variables, then you can step through that region of the program one statement at a time. You can execute the program at full-speed until just before the section of interest (using Run to cursor) then single step execution whilst monitoring variable values and output.

**Immediate mode**

If an untrapped error occurs at run time the program exits to *immediate mode* and displays an error message followed by the > prompt. At the prompt you can type any valid BASIC statement then press the Enter key, whereupon (as the name suggests) the statement will be executed immediately. This can be very useful in ascertaining the values of, or performing calculations on, variables; issuing 'star' commands (for example you might want to change the current directory with *CD*) or even calling procedures and functions.
Error handling

Default error handling

By default, if BBC BASIC detects an error while running a program it will immediately halt execution and report an error message, the filename of the INSTALLed or CALLed module in which the error occurred (if appropriate) and the line number (if any). This behaviour may often be adequate, but you have the option of *trapping* errors and writing your own code to handle them.

Reasons for trapping errors

There are a number of reasons why you might want to trap errors:

To make your program more 'friendly'

When you write a program for yourself, you know what you want it to do and you also know what it can't do. If, by accident, you try to make it do something which could give rise to an error, you accept the fact that BBC BASIC for Windows might terminate the program and return to immediate mode. However, when somebody else uses your program they are not blessed with your insight and they may find the program 'crashing out' without knowing what they have done wrong. Such programs are called 'fragile'. You can protect your user from much frustration if you display a more 'friendly' error report and fail 'gracefully'.

To ensure the error message is visible

The default error reporting writes the error message to the screen just like any other text output. Unfortunately there are several reasons why your program might not display a useful message, for example:

- The text colours, font, size, position etc. are such that the message is invisible.
- Screen output is disabled, e.g. by means of *REFRESH OFF or VDU 21*.
- The message is obscured, for example because of a dialogue box displayed in front of it.

To allow clearup operations to take place
There can be situations, particularly when using the Windows API, where aborting an operation 'mid stream' leaves the system in an unstable state. For example it might result in your program failing to work again unless you quit and restart *BBC BASIC for Windows*. It could even, in unusual circumstances, crash BBC BASIC. In such cases your own error handler can carry out the necessary tidying-up operations before reporting the error and exiting gracefully.

**To allow execution to continue**

Although it is preferable to prevent errors occurring in the first place, there can be occasions when this is difficult or impossible to achieve, and it is better to allow the error to happen. In these circumstances error trapping allows the program to continue execution, possibly with some specific action being taken as a result.

**Error trapping commands**

The two main commands provided by *BBC BASIC for Windows* for error trapping and recovery are:

```plaintext
ON ERROR ....
```

and

```plaintext
ON ERROR LOCAL ....
```

**ON ERROR**

The ON ERROR command directs *BBC BASIC for Windows* to execute the statement(s) following ON ERROR when a trappable error occurs:

```plaintext
ON ERROR PRINT "Oh No!":END
```

If an error was detected in a program after this line had been encountered, the message 'Oh No!' would be printed and the program would terminate. If, as in this example, the ON ERROR line contains the END statement or transfers control elsewhere (e.g. using GOTO) then the position of the line within the program is unimportant *so long as it is encountered before the error occurs*. If there is no transfer of control, execution following the error continues as usual on the succeeding line, so in this case the position of the ON ERROR line can matter.
As explained in the Program Flow Control sub-section, every time *BBC BASIC for Windows* encounters a FOR, REPEAT, GOSUB, FN, PROC or WHILE statement it 'pushes' the return address on to a 'stack' and every time it encounters a NEXT, UNTIL, RETURN, ENDWHILE statement or the end of a function or procedure it 'pops' the latest return address of the stack and goes back there. The program stack is where *BBC BASIC for Windows* records where it is within the structure of your program.

When an error is detected by *BBC BASIC for Windows*, the stack is cleared. Thus, you cannot just take any necessary action depending on the error and return to where you were because *BBC BASIC for Windows* no longer knows where you were.

If an error occurs within a procedure or function, the value of any LOCAL variables will be the last value they were set to within the procedure or function which gave rise to the error (the values, if any, they had before entry to the procedure or function are lost).

If an error occurs within a procedure or function containing PRIVATE variables, all subsequent calls to that procedure or function will be treated by BASIC as if they were re-entrant and the PRIVATE statement will be ignored. It is therefore most important that, if your program attempts to recover from errors using ON ERROR, no errors are permitted to occur within a procedure or function containing PRIVATE variables. If necessary use *ESC OFF* to ensure that not even the Escape error can occur.

**ON ERROR LOCAL**

The ON ERROR LOCAL command prevents *BBC BASIC for Windows* clearing the program stack. By using this command, you can trap errors within a FOR ... NEXT, REPEAT ... UNTIL or WHILE ... ENDWHILE loop or a subroutine, function or procedure without *BBC BASIC for Windows* losing its place within the program structure.

The following example program will continue after the inevitable 'Division by zero ' error:

```basic
FOR n=-5 TO 5
   ok% = TRUE
   ON ERROR LOCAL PRINT "Infinity" : ok% = FALSE
   IF ok% PRINT "The reciprocal of ";n;" is ";1/n
NEXT n
```
This is, of course, a poor use of error trapping. You should test for n=0 rather than allow the error to occur. However, it does provide a simple demonstration of the action of ON ERROR LOCAL. Also, you should test ERR to ensure that the error was the one you expected rather than, for example, Escape (ERR is a function which returns the number of the last error; it is explained later in this sub-section).

After the loop terminates (when 'n' reaches 6) the previous error trapping state is restored. Alternatively, you can explicitly restore the previous state using RESTORE ERROR.

You can call a subroutine, function or procedure to 'process' the error, but it must return to the loop, subroutine, function or procedure where the error was trapped:

```basica
x=OPENOUT "TEST"
FOR i=-5 TO 5
  ok% = TRUE
  ON ERROR LOCAL PROCerr : ok% = FALSE
  IF ok% PRINT#x,1/i
NEXT
CLOSE#x
: x=OPENIN "TEST"
REPEAT
  INPUT#x,i
  PRINT i
UNTIL EOF#x
CLOSE#x
END
: DEF PROCerr
  IF ERR <> 18 REPORT:END
  PRINT#x,3.4E38
ENDPROC
```

The position of the ON ERROR LOCAL within a procedure or function can be important. Consider the following two examples:

```
DEF PROC1(A)
  ON ERROR LOCAL REPORT : ENDPROC
  LOCAL B
    ....

DEF PROC1(A)
  LOCAL B
```

In the case of the first example, if an error occurs within the procedure, the formal parameter \( A \) will be automatically restored to its original value, but the LOCAL variable \( B \) will not; it will retain whatever value it had when the error occurred. In the case of the second example, both the formal parameter \( A \) and the LOCAL variable \( B \) will be restored to the values they had before the procedure was called.

If a procedure or function uses PRIVATE variables it is essential that any ON ERROR LOCAL statement be placed after the PRIVATE statement.

**ON ERROR or ON ERROR LOCAL?**

If you use ON ERROR, BBC BASIC for Windows clears the program stack. Once your program has decided what to do about the error, it must re-enter the program at a point that is not within a loop, subroutine, function or procedure. In practice, this generally means somewhere towards the start of the program. This is often undesirable as it makes it difficult to build well structured programs which include error handling. On the other hand, you will probably only need to write one or two program sections to deal with errors.

At first sight ON ERROR LOCAL seems a more attractive proposition since BBC BASIC for Windows remembers where it is within the program structure. The one disadvantage of ON ERROR LOCAL is that you will need to include an error handling section at every level of your program where you need to trap errors. Many of these sections of program could be identical.

You can mix the use of ON ERROR and ON ERROR LOCAL within your program. A single ON ERROR statement can act as a 'catch all' for unexpected errors, and one or more ON ERROR LOCAL statements can be used when your program is able to recover from predictable errors. ON ERROR LOCAL 'remembers' the current ON ERROR setting, and restores it when the loop, procedure or function containing the ON ERROR LOCAL command is finished, or when a RESTORE ERROR or RESTORE LOCAL is executed.

**Error reporting**

There are three functions, ERR, ERL, REPORT$ and one statement, REPORT, which may be used to investigate and report on errors. Using these, you can trap out
errors, check that you can deal with them and abort the program run if you cannot.

**ERR**

ERR returns the error number (see the section entitled Error messages and codes).

**ERL**

ERL returns the line number where the error occurred. If an error occurs in a procedure or function call, ERL will return the number of the calling line, not the number of the line in which the procedure/function is defined. If an error in a DATA statement causes a READ to fail, ERL will return the number of the line containing the DATA, not the number of the line containing the READ statement. If the line is not numbered, ERL returns zero.

**REPORT$**

REPORT$ returns the error string associated with the last error which occurred.

**REPORT**

REPORT prints out the error string associated with the last error which occurred. It is equivalent to PRINT REPORT$;

**Error trapping examples**

The majority of the following error trapping examples are all very simple programs without nested loops, subroutines, functions or procedures. Consequently, they use ON ERROR for error trapping in preference to ON ERROR LOCAL.

The example below does not try to deal with errors, it just uses ERR and REPORT to tell the user about the error. Its only advantage over *BBC BASIC for Windows' normal error handling is that it gives the error number; it would probably not be used in practice. As you can see from the second run, pressing <ESC> is treated as an error (number 17).

```basic
ON ERROR PROCerror
REPEAT
  INPUT "Type a number " num
  PRINT num," ",SQR(num)
```

PRINT
UNTIL FALSE
:
DEF PROCerror
PRINT
PRINT "Error No ";ERR
REPORT
END

Example run:

RUN
Type a number 1
   1
Type a number -2
   -2
Error No 21
Negative root

RUN
Type a number <Esc>
Error No 17
Escape

The example below has been further expanded to include error trapping. The only 'predictable' error is that the user will try a negative number. Any other error is unacceptable, so it is reported and the program aborted. Consequently, when <ESC> is used to abort the program, it is reported as an error. However, a further test for ERR=17 could be included so that the program would halt on ESCAPE without an error being reported.

ON ERROR PROCerror
REPEAT
   INPUT "Type a number " num
   PRINT num," ",SQR(num)
   PRINT
UNTIL FALSE
:
DEF PROCerror
PRINT
IF ERR=21 THEN PRINT "No negatives":ENDP
END

This is a case where the placement of the ON ERROR statement is important. When PROCerror exits, execution continues after the ON ERROR statement, which in this
case causes the program to restart from the beginning.

Example run:

```
RUN
Type a number 5
      5  2.23606798
Type a number 2
      2  1.41421356
Type a number -1
      -1
No negatives
Type a number 4
      4  2
Type a number <Esc>
Escape
```

The above example is very simple and was chosen for clarity. In practice, it would be better to test for a negative number before using SQR rather than trap the 'Negative root' error. A more realistic example is the evaluation of a user-supplied HEX number, where trapping 'Bad hex or binary' would be much easier than testing the input string beforehand.

```
ON ERROR PROCerror
REPEAT
   INPUT "Type a HEX number " Input$
   num=EVAL("&"+Input$)
   PRINT Input$,num
   PRINT
UNTIL FALSE:
DEF PROCerror
PRINT
IF ERR=28 THEN PRINT "Not hex":ENDPROC
REPORT
END
```

The next example is similar to the previous one, but it uses the ON ERROR LOCAL command to trap the error.

```
REPEAT
   ON ERROR LOCAL PROCerror
```
INPUT "Type a HEX number " Input$
num=EVAL("&"+Input$)
PRINT Input$,num
PRINT
UNTIL FALSE :
DEF PROC error
PRINT
IF ERR=28 THEN PRINT "Not hex":ENDPROC
REPORT
END

Note that had ON ERROR (rather than ON ERROR LOCAL) been used in this case, an error would give rise to a **Not in a REPEAT loop** error at the UNTIL statement. This is because ON ERROR clears the program's stack.
Procedures and functions

Introduction

Procedures and functions are similar to subroutines in that they are 'bits' of program which perform a discrete function. Like subroutines, they can be performed (called) from several places in the program. However, they have two great advantages over subroutines: you can refer to them by name and the variables used within them can be made private to the procedure or function.

Arguably, the major advantage of procedures and functions is that they can be referred to by name. Consider the two similar program lines below.

```plaintext
IF name$="ZZ" THEN GOSUB 500 ELSE GOSUB 800

IF name$="ZZ" THEN PROC_end ELSE PROC_print
```

The first statement gives no indication of what the subroutines at lines 500 and 800 actually do. The second, however, tells you what to expect from the two procedures. This enhanced readability stems from the choice of meaningful names for the two procedures.

A function often carries out a number of actions, but it always produces a single result. For instance, the 'built in' function INT returns the integer part of its argument.

```plaintext
age=INT(months/12)
```

A procedure on the other hand, is specifically intended to carry out a number of actions, some of which may affect program variables, but it does not directly return a result.

Whilst *BBC BASIC for Windows* has a large number of pre-defined functions (INT and LEN for example) it is very useful to be able to define your own to do something special. Suppose you had written a function called FN_discount to calculate the discount price from the normal retail price. You could write something similar to the following example anywhere in your program where you wished this calculation to be carried out.
It may seem hardly worth while defining a function to do something this simple. However, functions and procedures are not confined to single line definitions and they are very useful for improving the structure and readability of your program.

**Names**

The names of procedures and functions MUST start with PROC or FN and, like variable names, they cannot contain spaces. This restriction can give rise to some pretty unreadable names. However, the underline character can be used to advantage. Consider the procedure and function names below and decide which is the easier to read.

```
PROC_PRINTDETAILS      FNDISCOUNT
PROC_print_details    FN_discount
```

Function and procedure names may end with a '$'. However, this is not compulsory for functions which return strings.

**Function and procedure definitions**

**Starting a definition**

Function and procedure definitions are 'signalled' to *BBC BASIC for Windows* by preceding the function or procedure name with the keyword DEF . DEF must be at the beginning of the line. If the computer encounters DEF during execution of the program, the rest of the line is ignored. Consequently, you can put single line definitions anywhere in your program.

If two or more functions or procedures have the same name (either deliberately or by mistake) the one which is executed is determined by the following rules:

- If the same function or procedure name occurs both in your main program and in an INSTALL ed library, the one in the main program is used in preference to the one in the library. This allows you to replace a library routine with your own version.
• If your program contains two or more functions or procedures with the same name, the one nearest the end of the program is used.

• If the same function or procedure name occurs in two or more INSTALLed libraries, the one used is that in the last library to be loaded.

The function/procedure body

The 'body' of a procedure or function must not be executed directly - it must be performed (called) by another part of the program. Since BBC BASIC for Windows only skips the rest of the line when it encounters DEF, there is a danger that the remaining lines of a multi-line definition might be executed directly. You can avoid this by putting multi-line definitions at the end of the main program text after the END statement. Procedures and functions do not need to be declared before they are used and there is no speed advantage to be gained by placing them at the start of the program.

Ending a definition

The end of a procedure definition is indicated by the keyword ENDPROC. The end of a function definition is signalled by using a statement which starts with an equals (=) sign. The function returns the value of the expression to the right of the equals sign.

Single line functions/procedures

For single line definitions, the start and end are signalled on the same line. The first example below defines a function which returns the average of two numbers. The second defines a procedure which clears from the current cursor position to the end of the line (on a 40 column screen):

```
DEF FN_average(A,B) = (A+B)/2
DEF PROC_clear:PRINT SPC(40-POS);:ENDPROC
```

Extending the language

You can define a whole library of procedures and functions and include them in your programs, or INSTALL them at run-time. By doing this you can effectively extend the scope of the language. For instance, BBC BASIC for Windows does not
have a 'clear to end of screen' command. The example below is a procedure to clear to the end of 'screen' (BASIC's output window) with an 80 column by 25 row display (e.g. MODE 3). The three variables used (i%, x%, and y%) are declared as LOCAL to the procedure (see later).

```basic
DEF PROC_clear_to_end
    LOCAL i%,x%,y%
    x%=POS:y%=VPOS
    REM If not already on the last line, print lines of
    REM spaces which will wrap around until the last line
    IF y% < 24 FOR i%=y% TO 23:PRINT SPC(80);:NEXT
    REM Print spaces to the end of the last line.
    PRINT SPC(80-x%);,
    REM Return the cursor to its original position
    PRINT TAB(x%,y%);
ENDPROC
```

Passing parameters

When you define a procedure or a function, you list the parameters to be passed to it in brackets. For instance, the discount example expected one parameter (the retail price) to be passed to it. You can write the definition to accept any number of parameters. For example, we may wish to pass both the retail price and the discount percentage. The function definition would then look something like this:

```basic
DEF FN_discount(price,pcent)=price*(1-pcent/100)
```

In this case, to use the function we would need to pass two parameters.

```basic
retail_price=26.55
discount_price=FN_discount(retail_price,25)
```

or

```basic
price=26.55
discount=25
price=FN_discount(price,discount)
```

or
Formal and actual parameters

The value of the first parameter in the call to the procedure or function is passed to the first variable named in the parameter list in the definition, the second to the second, and so on. This is termed 'passing by value' (see also passing parameters by reference ). The parameters declared in the definition are called 'formal parameters' and the values passed in the statements which perform (call) the procedure or function are called 'actual parameters'. There must be as many actual parameters passed as there are formal parameters declared in the definition.

```plaintext
FOR I% = 1 TO 10
    PROC_printit(I%) : REM I% is the Actual parameter
NEXT
END
DEF PROC_printit(num1) : REM num1 is the Formal parameter
PRINT num1
ENDPROC
```

The formal parameters must be variables, but the actual parameters may be variables, constants or expressions. When the actual parameter is a variable, it need not be (and usually won't be) the same as the variable used as the formal parameter. Formal parameters are automatically made local to the procedure or function.

You can pass a mix of string and numeric parameters to the procedure or function, and a function can return either a string or numeric value, irrespective of the type of parameters passed to it. However, you must make sure that the parameter types match up. The first example below is correct; the second would give rise to an 'Incorrect arguments ' error message and the third would cause a 'Type mismatch ' error to be reported.

Correct

```plaintext
PROC_printit(1,"FRED",2)
END
:
DEF PROC_printit(num1,name$,num2)
PRINT num1,name$,num2
ENDPROC
```
Incorrect arguments error

```
PROC_printit(1,"FRED",2,4)
END
:
DEF PROC_printit(num1,name$,num2)
PRINT num1,name$,num2
ENDPROC
```

Type mismatch error

```
PROC_printit(1,"FRED","JIM")
END
:
DEF PROC_printit(num1,name$,num2)
PRINT num1,name$,num2
ENDPROC
```

Local variables

You can use the statements LOCAL and PRIVATE to declare variables for use locally within individual procedures or functions (the formal parameters are also automatically local to the procedure or function declaring them). Changing the values of local variables has no effect on variables of the same name (if any) elsewhere in the program. This technique is used extensively in the example file handling programs in this manual.

Declaring variables as LOCAL initialises them to zero (in the case of numeric variables) or null/empty (in the case of string variables). Declaring variables as PRIVATE causes them to retain their values from one call of the function/procedure to the next. If an array or structure is made LOCAL or PRIVATE it must be re-DIMensioned before use.

Variables which are not formal parameters nor declared as LOCAL or PRIVATE are known to the whole program, including all the procedures and functions. Such variables are called **global**.

Recursive functions/procedures

Because the formal parameters which receive the passed parameters are local, all
procedures and functions can be recursive. That is, they can call themselves. But for this feature, the short example program below would be difficult to code. It is the often used example of a factorial number routine (the factorial of a number \( n \) is \( n \times (n-1) \times (n-2) \times \ldots \times 1 \). Factorial 6, for instance, is \( 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720 \)).

```
REPEAT
  INPUT "Enter an INTEGER less than 35 "num
  UNTIL INT(num)=num AND num<35
  fact=FN_fact_num(num)
  PRINT num,fact
END :

DEF FN_fact_num(n)
IF n=1 OR n=0 THEN =1
REM Return with 1 if n= 0 or 1
=n*FN_fact_num(n-1)
REM Else go round again
```

Since 'n' is the input variable to the function FN_fact_num, it is local to each and every use of the function. The function keeps calling itself until it returns the answer 1. It then works its way back through all the calls until it has completed the final multiplication, when it returns the answer. The limit of 35 on the input number prevents the answer being too big for the computer to handle.

**Passing arrays to functions and procedures**

*BBC BASIC for Windows* allows you to pass an entire array (rather than just a single element) to a function or procedure. Unlike single values or strings, which are usually passed to a function or procedure by value, an array is passed by reference. That is, a pointer to the array contents is passed rather than the contents themselves. The consequence of this is that if the array contents are modified within the function or procedure, they remain modified on exit from the function or procedure. In the following example the procedure PROC_multiply multiplies all the values in a numeric array by a specified amount:

```
DIM arr(20)
FOR n%=0 TO 20 : arr(n%) = n% : NEXT
PROC_multiply(arr(), 2)
FOR n%=0 TO 20: PRINTarr(n%) : NEXT
END :

DEF PROC_multiply(B(), M)
LOCAL n%
FOR n% = 0 TO DIM(B(),1)
```
Note the use of DIM as a function to return the number of elements in the array.

The advantage of passing an array as a parameter, rather than accessing the 'global' array directly, is that the function or procedure doesn't need to know the name of the array. Ideally a FN/PROC shouldn't need to know the names of variables used in the main program from which it is called, and the main program shouldn't need to know the names of variables contained in a function or procedure it calls. This is a principle known as information hiding and is especially important in the case of functions and procedures contained in an INSTALLed library.

Passing parameters by reference

Although function and procedure parameters are usually passed by value, it is possible to specify that they should be passed by reference by using the keyword RETURN in the definition:

```
DEF PROCcomplexsquare(RETURN r, RETURN i)
```

If a parameter passed by reference is modified within the function or procedure, it remains modified on exit. This is the case even if the actual parameter and the formal parameter have different names.

Suppose you want to write a function which returns a complex number. Since a complex number is represented by two numeric values (the real part and the imaginary part) a conventional user-defined function, which can return only a single value, is not suitable. Conventionally you would have to resort to using global variables or an array or structure to return the result, but by passing parameters by reference this can be avoided.

The following example shows the use of a procedure to return the square of a complex number:

```
INPUT "Enter complex number (real, imaginary): " real, imag
PROCcomplexsquare(real, imag)
PRINT "The square of the number is ";real " + ";imag " i"
END
```
You can use a similar technique to return strings rather than numbers. The following example, rather pointlessly, takes two strings and mixes them up by exchanging alternate characters between them:

```basic
INPUT "Enter two strings, separated by a comma: " A$, B$
PROCmixupstrings(A$, B$)
PRINT "The mixed-up strings are " A$ " and " B$
END

DEF PROCmixupstrings(RETURN a$, RETURN b$)
LOCAL c$, I%
FOR I% = 1 TO LEN(a$) STEP 2
   c$ = MID$(a$, I%, 1)
   MID$(a$, I%, 1) = MID$(b$, I%, 1)
   MID$(b$, I%, 1) = c$
NEXT I%
ENDPROC
```

You can also use this facility to create a variable whose name is determined at runtime, something which is otherwise impossible to achieve:

```basic
INPUT "Enter a variable name: " name$
INPUT "Enter a numeric value for the variable: " value$
dummy% = EVAL("FNassign("+name$+","+value$+)")
PRINT "The variable "name$" has the value "; EVAL(name$)
END
DEF FNassign(RETURN n, v) : n = v : = 0
```

### Returning arrays from procedures and functions

It is possible to declare an array within a function or procedure and return that new array to the calling code. To do that you must pass an *undeclared* array as the parameter and use the RETURN keyword in the function or procedure definition:

```basic
PROCnewarray(alpha())
PROCnewarray(beta())
PROCnewarray(gamma())
DEF PROCnewarray(RETURN a())
```
DIM a(3, 4, 5)
...
ENDPROC

If you pass an array which already exists, it must have an identical format to the declaration within the function or procedure.

*Note:* You cannot use this technique to declare LOCAL arrays.

**Indirect procedure and function calls**

You can call a procedure or function *indirectly* using a pointer, rather than by specifying the actual name of the procedure or function:

```plaintext
pptr% = ^PROC1
REM ....
PROC(pptr%)
```

The above code calls the procedure PROC1 (without any parameters) but the name of the procedure is specified separately from the call itself. This allows you to do things such as determine which procedure or function should be called at run time in a more flexible way than can be achieved using ON PROC or EVAL. For example you could have an array of function pointers and select the wanted one according to an index value:

```plaintext
DIM fptr%(4)
fptr%() = ^FNzero, ^FNone, ^FNtwo, ^FNthree, ^FNfour
REM ....
answer = FN(fptr%(index%))
```

If the procedure or function takes parameters then you must include a pair of parentheses when you use ^ to find the pointer value:

```plaintext
DIM fptr%(4)
fptr%() = ^FN0(), ^FN1(), ^FN2(), ^FN3(), ^FN4()
REM ....
answer = FN(fptr%(index%))(parameters)
```

There are two very important caveats when using indirect procedure and function calls. Firstly, at least one conventional PROC or FN call must have been made...
before you attempt to read a procedure or function pointer (if not a No such FN/PROC error will result). Secondly, you must be very careful to make an indirect call only with a valid procedure or function pointer; using an incorrect value may well crash BASIC.

Passing procedures and functions as parameters

You can use a procedure or function name as a parameter to a procedure or function. For example you might want to write a library function to create a push button; you could pass to the function the name of a procedure which will be called when the button is pressed:

```
hb% = FN_button("Name",10,100,32,12,PROCbut)
```

Within FN_button an integer variable should be used to receive the procedure or function's address. An indirect procedure or function call can then be made using this variable:

```
DEF FN_button(text$,x%,y%,cx%,cy%,RETURN pr%)
REM ....
IF button_pressed PROC(^pr%)
```

Note that the use of RETURN is mandatory. In the case of a function being passed rather than a procedure, this ensures that it's the function's address not its value that is passed into FN_button. As before you must be very careful to use only a valid procedure or function pointer; using an incorrect value (e.g. as a result of omitting the ^ ) is likely to crash BASIC.
**Input editing**

When a *BBC BASIC for Windows* program is waiting for user input (as the result of an INPUT statement), or when it is waiting for a command in immediate mode, the following editing keys are active:

<table>
<thead>
<tr>
<th>Key</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>Toggle between insert and overtype modes. In overtype mode any character typed will over-write the existing character at that position on the line. In insert mode any character typed will 'push' the rest of the line (if any) to the right to make room. The mode is indicated by the shape of the cursor: normally an <em>underline</em> indicates insert mode and a <em>solid block</em> indicates overtype mode.</td>
</tr>
<tr>
<td>Backspace</td>
<td>Delete the character immediately to the left of the cursor, and move the cursor left one position. The rest of the line (if any) is moved left to fill the gap. If the cursor is already at the beginning of a line, the key has no effect.</td>
</tr>
<tr>
<td>Delete</td>
<td>Delete the character at the cursor position and move the rest of the line (if any) left to fill the gap. If the cursor is already at the end of the line, the key has no effect.</td>
</tr>
<tr>
<td>Home</td>
<td>Move the cursor to the start of the line.</td>
</tr>
<tr>
<td>End</td>
<td>Move the cursor to the end of the line.</td>
</tr>
<tr>
<td>←</td>
<td>Move the cursor one character to the left; if the cursor is already at the beginning of a line, the key has no effect.</td>
</tr>
<tr>
<td>→</td>
<td>Move the cursor one character to the right; if the cursor is already at the end of the line, the key has no effect.</td>
</tr>
<tr>
<td>↑</td>
<td>Move the cursor up one line, if the input has 'wrapped around' to two or more lines. Otherwise the cursor is moved to the start of the line.</td>
</tr>
<tr>
<td>↓</td>
<td>Move the cursor down one line, if the input has 'wrapped around' to two or more lines. Otherwise the cursor is moved to the end of the line.</td>
</tr>
<tr>
<td>Ctrl+U</td>
<td>Delete everything to the left of the cursor. If the cursor is already at the beginning of the line the key has no effect.</td>
</tr>
<tr>
<td>Ctrl+V</td>
<td>Paste in the contents of the clipboard (if any). If there is more than one line of text in the clipboard only the first line is pasted; the rest is discarded.</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Ctrl+Enter (Ctrl+Return)</td>
<td>Delete everything from the cursor position to the end of the line. If the cursor is already at the end of the line the key has no effect.</td>
</tr>
<tr>
<td>Tab</td>
<td>Enter Copy key editing mode.</td>
</tr>
<tr>
<td>Shift/Tab</td>
<td>Exit Copy key editing mode.</td>
</tr>
</tbody>
</table>

In situations when use of the INPUT statement is unacceptable (for example if a proportional-spaced font has been selected, or it's important that event interrupts not be delayed) you can use a replacement written in BASIC. Examples are the **FNpsinput** routine listed here or the **FNinput** routine contained in the NOWAIT library.
Copy key editing

When a *BBC BASIC for Windows* program is waiting for user input (as the result of an INPUT statement), or when it is waiting for a command in immediate mode, Copy Key editing is available. As the PC keyboard does not have a dedicated COPY key the Tab key is used for this purpose. Unlike the BBC Micro, copy-key editing must be initiated by first pressing this key.

When you press the Tab (copy) key, *BBC BASIC for Windows* displays two cursors. The large 'block' cursor shows where anything you enter will appear; it is called the **write cursor**. The other small flashing cursor is the one that can be moved around by the cursor control keys. For reasons which will become clear later, this is called the **read cursor**.

You can exit the copy-key editing mode (without pressing Enter) by pressing Shift/Tab; this restores the normal line-editing features. Thus, you can switch between the two forms of editing by using Tab and Shift/Tab.

**The copy key**

Once you have moved the **read cursor** to the line on the screen you wish to edit, you may copy the characters on that line to the **write cursor** 's position by pressing the Tab (copy) key. Every time you press the Tab key, the character at the **read cursor** 's position will be written to the **write cursor** 's position; then, the **read cursor** and the **write cursor** will move on one character position. At any time you wish, you may type characters on the keyboard and they will be entered at the **write cursor** 's position. The **write cursor** will move on one character position after each character is typed on the keyboard.

You don't have to confine yourself to copying from a single line. You can move the **read cursor** around the screen and copy any character or sequence of characters displayed in any position on the screen. You can copy the same thing over and over again, or skip bits of a line by moving the **read cursor** over words you don't want in the new line.

**The backspace key**

The backspace key works as it normally does. When you press the backspace key,
the character to the left of the write cursor is deleted and the write cursor moves back one character position. Thus, you can delete characters you have copied or entered in error.
Hardcopy output to a printer

BBC BASIC is a little unusual in the way it controls hardcopy output to a printer. Many BASIC dialects use the LPRINT keyword, which acts like PRINT but outputs to the printer rather than to the screen. BBC BASIC does not have LPRINT, but instead enables or disables printer output using VDU commands.

Printer output is enabled using VDU 2 and disabled using VDU 3. Therefore to output to both the printer and the screen you would do:

```
VDU 2 : REM Turn printer on
PRINT "Here is some output to the printer."
PRINT "It will also appear on the screen."
VDU 1,12,3 : REM Eject page and turn printer off
```

Note the VDU 1,12 ; that tells the printer that the page is complete and instructs it to print and eject the sheet. Without it the page won't be printed until your program exits or a timeout occurs.

If you want to output only to the printer and not to the screen you can redirect output to the printer using *OUTPUT 15:

```
*OUTPUT 15
PRINT "Here is some output to the printer."
PRINT "It will not appear on the screen."
*OUTPUT 0
VDU 2,1,12,3
```

If your printout isn't the right size, or you want to select a different font, add a *PRINTERFONT command at the beginning of your program, for example:

```
*PRINTERFONT Courier New,12
*MARGINS 10,10,10,10
```

The *MARGINS command resets the margins to the default size of 10 mm.
Labelling program lines

**BBC BASIC** was one of the first dialects of BASIC to support *structured programming* and hence to make the use of GOTO unnecessary. Nevertheless **GOTO** (and **GOSUB**) are still provided, principally to make it easier to convert programs from older dialects. In a similar way **BBC BASIC for Windows** provides the ability to *label* program lines instead of using line numbers. A label may be used as the target of a **GOTO**, **GOSUB** or **RESTORE**:

```
A = 5
(loop)
PRINT A
A = A + 1
IF A < 10 THEN GOTO (loop)
```

Note that the label must be enclosed in parentheses, and be the first thing on a program line. Labels and variables share the same *namespace* so you should not normally use the same name for a label and a variable in your program.

Labels are limited in much the same way as line numbers: they are global to the whole program and cannot be used in **INSTALLed or CALLed modules**. You are advised only to use labels for debugging, or when converting a program from another dialect of BASIC.
Multilingual (Unicode) text output

By default all text output to the screen or the printer uses the 8-bit ANSI character set; the only foreign language characters available are accented letters. If you need to output text in other alphabets (such as Arabic, Greek, Hebrew or Cyrillic) - without needing to select a different font for each one - you can do so by enabling **UTF-8 mode**, which you do using a variant of the VDU 23,22 command in conjunction with selecting a suitable Unicode font (e.g. Arial, Courier New or Times New Roman, under Windows™ 2000, XP, Vista, Windows 7, 8/8.1 or 10).

Once UTF-8 mode has been selected, all text output to the screen or the printer uses the multibyte UTF-8 character set; each character requiring 1, 2 or 3 bytes depending on its position within the Basic Multilingual Plane. The following code snippet illustrates its use:

```
VDU 23,22,640;512;8,16,16,8
*FONT Arial,24
aleph$ = CHR$(&D7)+CHR$(&90)
thorn$ = CHR$(&C3)+CHR$(&9E)
euro$ = CHR$(&E2)+CHR$(&82)+CHR$(&AC)
PRINT aleph$ " " thorn$ " " euro$
```

Because UTF-8 text uses a variable number of bytes for each character, features that rely on counting characters (for example COUNT, WIDTH and TAB(X)) won't work reliably, even when a fixed-pitch font is selected.
Introduction to graphics

BBC BASIC for Windows has an extensive range of graphics commands. These commands are compatible with those available on the BBC Micro/Master and Acorn Archimedes, but are different from many other versions of BASIC.

This section draws together the information about graphics facilities and colours given under various keywords. Although it repeats much of the information presented elsewhere in this manual it provides a consolidated explanation of the graphics facilities and colours available.
Display modes

_BBC BASIC for Windows_ programs start with a default output window, having black text on a white background. The size of this initial window is determined by Windows™ and depends on the current display settings of your PC. Whilst this default window may be adequate for simple programs outputting only text, it is unlikely to be acceptable if your program mixes graphics and text, or needs a specific number of text rows and columns, especially if you want it to work correctly with different display settings (for example on other people's PCs).

In these cases you should always select an explicit window size. You can do that using the MODE statement, or by selecting a _user-defined_ mode with VDU 23,22... . In BBC BASIC text rows are counted from the **top** of the window and graphics coordinates are measured from the **bottom** of the window, so selecting a display mode with known dimensions is particularly important to ensure consistent alignment of text and graphics. When a MODE statement is executed the window dimensions are changed to those appropriate to that mode, with (initially) white text on a black background. The 'screen' (BASIC's output window) is cleared and all the graphics and text parameters (colours, origin, etc) are reset to their default values.

A good general-purpose mode is **MODE 8** which has 80 columns by 32 rows of text, and graphics dimensions of 640 pixels by 512 pixels (1280 x 1024 graphics units). This mode is small enough to fit easily within the desktop area on the majority of PCs. You can select MODE 8 as follows:

```plaintext
MODE 8
```

If you prefer black text on a white background a mode of the same size can be selected as follows:

```plaintext
VDU 23,22,640;512;8,16,16,128
```

Don't be tempted to use a MODE whose nominal dimensions are the same as your display resolution; by the time you've made allowances for the window's title bar and borders, and the task bar, it won't fit! So if your display is set to a resolution of 1024 x 768 don't use a MODE bigger than about 800 x 600 pixels (e.g. **MODE 20**). If you (exceptionally) want to use the entire screen you can either maximise the
window or operate in full screen mode. You should not attempt to achieve the same effect by choosing a MODE larger than will fit.

**Available modes**

The available display modes and the resolution of which they are capable are listed below. MODEs 0 to 7 are compatible with MODEs 0 to 7 on the BBC Micro (with some minor differences) and MODEs 8 to 33 are unique to *BBC BASIC for Windows*. No attempt has been made to emulate the screen modes of the Acorn Archimedes, other than MODEs 0 to 7.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Text (chars)</th>
<th>Graphics (pixels)</th>
<th>Graphics units</th>
<th>Logical Colours</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>80x32</td>
<td>640x512</td>
<td>1280x1024</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>40x32</td>
<td>640x512</td>
<td>1280x1024</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>20x32</td>
<td>640x512</td>
<td>1280x1024</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>80x25</td>
<td>640x500</td>
<td>1280x1000</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>40x32</td>
<td>640x512</td>
<td>1280x1024</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>20x32</td>
<td>640x512</td>
<td>1280x1024</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>40x25</td>
<td>640x500</td>
<td>1280x1000</td>
<td>16</td>
</tr>
<tr>
<td>7</td>
<td>40x25</td>
<td>teletext</td>
<td>teletext</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>80x32</td>
<td>640x512</td>
<td>1280x1024</td>
<td>16</td>
</tr>
<tr>
<td>9</td>
<td>40x32</td>
<td>640x512</td>
<td>1280x1024</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>90x36</td>
<td>720x576</td>
<td>1440x1152</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>45x36</td>
<td>720x576</td>
<td>1440x1152</td>
<td>16</td>
</tr>
<tr>
<td>12</td>
<td>120x48</td>
<td>960x768</td>
<td>1920x1536</td>
<td>16</td>
</tr>
<tr>
<td>13</td>
<td>60x48</td>
<td>960x768</td>
<td>1920x1536</td>
<td>16</td>
</tr>
<tr>
<td>14</td>
<td>160x64</td>
<td>1280x1024</td>
<td>2560x2048</td>
<td>16</td>
</tr>
<tr>
<td>15</td>
<td>80x64</td>
<td>1280x1024</td>
<td>2560x2048</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>80x25</td>
<td>640x400</td>
<td>1280x800</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>40x25</td>
<td>640x400</td>
<td>1280x800</td>
<td>16</td>
</tr>
<tr>
<td>18</td>
<td>80x30</td>
<td>640x480</td>
<td>1280x960</td>
<td>16</td>
</tr>
<tr>
<td>19</td>
<td>40x30</td>
<td>640x480</td>
<td>1280x960</td>
<td>16</td>
</tr>
<tr>
<td>20</td>
<td>100x30</td>
<td>800x600</td>
<td>1600x1200</td>
<td>16</td>
</tr>
<tr>
<td>21</td>
<td>50x30</td>
<td>800x600</td>
<td>1600x1200</td>
<td>16</td>
</tr>
<tr>
<td>Mode</td>
<td>Width</td>
<td>Height</td>
<td>Size</td>
<td>Logical Colours</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>--------</td>
<td>--------</td>
<td>-----------------</td>
</tr>
<tr>
<td>22</td>
<td>128x48</td>
<td>1024x768</td>
<td>2048x1536</td>
<td>16</td>
</tr>
<tr>
<td>23</td>
<td>64x48</td>
<td>1024x768</td>
<td>2048x1536</td>
<td>16</td>
</tr>
<tr>
<td>24</td>
<td>144x54</td>
<td>1152x864</td>
<td>2304x1728</td>
<td>16</td>
</tr>
<tr>
<td>25</td>
<td>72x54</td>
<td>1152x864</td>
<td>2304x1728</td>
<td>16</td>
</tr>
<tr>
<td>26</td>
<td>160x60</td>
<td>1280x960</td>
<td>2560x1920</td>
<td>16</td>
</tr>
<tr>
<td>27</td>
<td>80x60</td>
<td>1280x960</td>
<td>2560x1920</td>
<td>16</td>
</tr>
<tr>
<td>28</td>
<td>180x54</td>
<td>1440x1080</td>
<td>2880x2160</td>
<td>16</td>
</tr>
<tr>
<td>29</td>
<td>90x54</td>
<td>1440x1080</td>
<td>2880x2160</td>
<td>16</td>
</tr>
<tr>
<td>30</td>
<td>200x75</td>
<td>1600x1200</td>
<td>3200x2400</td>
<td>16</td>
</tr>
<tr>
<td>31</td>
<td>100x75</td>
<td>1600x1200</td>
<td>3200x2400</td>
<td>16</td>
</tr>
<tr>
<td>32</td>
<td>80x25</td>
<td>640x400</td>
<td>1280x800</td>
<td>2</td>
</tr>
<tr>
<td>33</td>
<td>40x25</td>
<td>640x400</td>
<td>1280x800</td>
<td>4</td>
</tr>
</tbody>
</table>

This table indicates the number of logical colours available in each display mode. As described in The Palette sub-section, the number of physical colours available depends upon the display settings.

With 'paletted' displays (256 colours or fewer) the number of logical colours represents the maximum number of different colours that can be displayed in the BBC BASIC window at any one time. Palette animation is possible: changing the contents of the palette affects colours previously plotted.

With 'non-paletted' displays (32768 colours or more) the number of colours which can be shown at any one time is limited only by the display settings. Changing the contents of the palette affects only colours subsequently plotted.

- MODE 3 is compatible with BBC BASIC (86) MODEs 3, 11 and 15.
- MODE 6 is compatible with BBC BASIC (86) MODEs 6 and 14.
- MODE 7, the teletext mode, is fully implemented.
- MODE 16 is compatible with BBC BASIC (86) MODE 8.
- MODE 17 is compatible with BBC BASIC (86) MODEs 9, 10, 12 and 13.
- MODE 18 is compatible with BBC BASIC (86) MODE 18.
- MODE 32 is compatible with BBC BASIC (86) MODE 0.
- MODE 33 is compatible with BBC BASIC (86) MODEs 1, 2, 4 and 5.
- MODEs 3, 6, 20, 21, 28 and 29 have increased line spacing, and therefore fewer lines of text. Text can be easier to read in these modes.
The command *EGA ON automatically maps mode numbers 0 to 7 to EGA-compatible modes in the same way as it does with BBC BASIC (86).

The command *EGA OFF automatically maps mode numbers 0 to 7 to CGA-compatible modes, as used by BBC BASIC (86). When converting a BBC BASIC(86) graphics program to run under BBC BASIC for Windows add a *EGA OFF command at the beginning of the program. A program modified in this way will still run under BBC BASIC (86).

As well as these predefined MODEs, you can also select a 'custom' mode using VDU 23,22.
Colours

The Palette

You can set the text and graphics background and foreground colours using the COLOUR and GCOL (Graphics COLOur) statements. These commands use numbers to specify the colours.

Initially, and after a MODE change, the colour numbers refer to default colours which depend on the mode, as follows:

**Two colour modes: 0, 4 and 32**

<table>
<thead>
<tr>
<th>Foreground</th>
<th>Background</th>
<th>Colour</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>128</td>
<td>Black</td>
</tr>
<tr>
<td>1</td>
<td>129</td>
<td>White</td>
</tr>
</tbody>
</table>

**Four colour modes: 1, 5 and 33**

<table>
<thead>
<tr>
<th>Foreground</th>
<th>Background</th>
<th>Colour</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>128</td>
<td>Black</td>
</tr>
<tr>
<td>1</td>
<td>129</td>
<td>Magenta</td>
</tr>
<tr>
<td>2</td>
<td>130</td>
<td>Green</td>
</tr>
<tr>
<td>3</td>
<td>131</td>
<td>White</td>
</tr>
</tbody>
</table>

**Sixteen colour modes: 2, 3, 6 and 8-31**
<table>
<thead>
<tr>
<th>Foreground</th>
<th>Background</th>
<th>Colour</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>128</td>
<td>Black</td>
</tr>
<tr>
<td>1</td>
<td>129</td>
<td>Red</td>
</tr>
<tr>
<td>2</td>
<td>130</td>
<td>Green</td>
</tr>
<tr>
<td>3</td>
<td>131</td>
<td>Yellow</td>
</tr>
<tr>
<td>4</td>
<td>132</td>
<td>Blue</td>
</tr>
<tr>
<td>5</td>
<td>133</td>
<td>Magenta</td>
</tr>
<tr>
<td>6</td>
<td>134</td>
<td>Cyan</td>
</tr>
<tr>
<td>7</td>
<td>135</td>
<td>White</td>
</tr>
<tr>
<td>8</td>
<td>136</td>
<td>Intensified Black (Grey)</td>
</tr>
<tr>
<td>9</td>
<td>137</td>
<td>Intensified Red</td>
</tr>
<tr>
<td>10</td>
<td>138</td>
<td>Intensified Green</td>
</tr>
<tr>
<td>11</td>
<td>139</td>
<td>Intensified Yellow</td>
</tr>
<tr>
<td>12</td>
<td>140</td>
<td>Intensified Blue</td>
</tr>
<tr>
<td>13</td>
<td>141</td>
<td>Intensified Magenta</td>
</tr>
<tr>
<td>14</td>
<td>142</td>
<td>Intensified Cyan</td>
</tr>
<tr>
<td>15</td>
<td>143</td>
<td>Intensified White</td>
</tr>
</tbody>
</table>

However you can (with certain limitations) set the actual colour displayed by these colour numbers to any one of the physical colours available, using the VDU 19 command or a variant of the COLOUR statement.

Because you can change the actual colour displayed by a colour number to any one of the physical colours available to you, these numbers are called logical colours. You could, if you wished, set all the logical colour numbers to the same physical colour.

It may help you to visualise the logical colours as an artist's palette with up to 16 numbered areas where colours may be placed. You could place any of the colours available to you in any one of the areas on the palette. Once the colours were in place on the palette, you could 'paint by numbers'.

**Logical colours**

The number of logical colours available depends upon the chosen display mode:
In MODEs 0, 4 and 32 there are two logical colours available.

In MODEs 1, 5 and 33 there are four logical colours available.

In all other modes (except MODE 7) there are sixteen logical colours available.

MODE 7 (the teletext mode) has eight colours, but there is no colour palette.

**Physical colours**

The physical colours are also referred to by numbers. However, a physical colour number always refers to the same colour. You can specify a physical colour in one of three ways:

- As an entry (0-15) in the physical colour palette.
- As an RGB (red, green, blue) value, with 6-bits per colour (0-63).
- As an RGB (red, green, blue) value, with 8-bits per colour (0-255).

**The physical palette**

The physical colour palette has sixteen colours, as follows:

<table>
<thead>
<tr>
<th>Colour Number</th>
<th>Physical Colour</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Black (normal background)</td>
</tr>
<tr>
<td>1</td>
<td>Red</td>
</tr>
<tr>
<td>2</td>
<td>Green</td>
</tr>
<tr>
<td>3</td>
<td>Yellow</td>
</tr>
<tr>
<td>4</td>
<td>Blue</td>
</tr>
<tr>
<td>5</td>
<td>Magenta (blue-red)</td>
</tr>
<tr>
<td>6</td>
<td>Cyan (blue-green)</td>
</tr>
<tr>
<td>7</td>
<td>White (normal foreground)</td>
</tr>
<tr>
<td>8</td>
<td>Intensified Black (grey)</td>
</tr>
<tr>
<td>9</td>
<td>Intensified Red</td>
</tr>
<tr>
<td>10</td>
<td>Intensified Green</td>
</tr>
<tr>
<td>11</td>
<td>Intensified Yellow</td>
</tr>
</tbody>
</table>
To select a logical colour to one of these physical colours you use either the VDU 19 command or the COLOUR statement:

VDU 19, logical, physical, 0, 0, 0  
COLOUR logical, physical

6-bit RGB colours

In the 6-bit RGB mode you set the physical colour by specifying the values of the individual red, green and blue components as numbers in the range 0 (minimum) to 63 (maximum). Hence the value 0,0,0 corresponds to black and the value 63,63,63 corresponds to peak (intensified) white.

To select a logical colour to one of these physical colours you must use the VDU 19 command:

VDU 19, logical, -1, red, green, blue

This mode is provided for compatibility with BBC BASIC (86).

8-bit RGB colours

In the 8-bit RGB mode you set the physical colour by specifying the values of the individual red, green and blue components as numbers in the range 0 (minimum) to 255 (maximum). Hence the value 0,0,0 corresponds to black and the value 255,255,255 corresponds to peak (intensified) white.

To select a logical colour to one of these physical colours you use either the VDU 19 command or the COLOUR statement:

VDU 19, logical, 16, Red, Green, Blue  
COLOUR logical, Red, Green, Blue
Text colours

You can set the text foreground and background colours with the COLOUR (or VDU 17) statement. The colour referenced by this statement is the logical colour, in the range 0 to 15:

\[ \text{COLOUR colour} \]

If the colour number is less than 128, the colour of the text is set to that value. If the number is 128 or greater, the colour of the text background is set to the value minus 128.

Graphics colours

You can set the graphics foreground and background colours with the GCOL (or VDU 18) statement. The colour referenced by this statement is the logical colour, in the range 0 to 15.

The GCOL command also (optionally) specifies how the colour is to be plotted on the screen. It can be plotted directly, ANDed, ORed or Exclusive-ORed with the colour already there, or the existing colour can be inverted. If the plotting mode is omitted, it defaults to zero (plot):

\[ \text{GCOL colour} \]
\[ \text{GCOL mode,colour} \]

If the colour number is less than 128, the graphics colour is set to that value. If the number is 128 or greater, the colour of the graphics background is set to the value minus 128. Generally, you will not notice a change in the graphics background colour until you clear the graphics screen with a CLG statement.

The plotting modes are as follows:
mode=0  Plot the colour specified.
mode=1  OR the specified colour with the colour that is already there.
mode=2  AND the specified colour with the colour that is already there.
mode=3  Exclusive-OR the specified colour with the colour that is already there.
mode=4  Invert the colour that is already there.

See the Compatibility limitations section for important notes on the use of these plotting modes.
Drawing on the screen

Introduction

This sub-section describes the statements and functions available for drawing on the graphics screen and discovering the colour of any pixel (picture element) on the screen.

Screen dimensions

*BBC BASIC for Windows* uses 'graphics units' such that one pixel (picture element) corresponds to two graphics units. For example, in MODE 8, BASIC's 'screen' (the output window) is 640 pixels wide by 512 pixels high. This corresponds to a size of 1280 graphics units wide by 1024 graphics units high.

The graphics viewport

The graphics viewport is a rectangular region of the 'screen' (BASIC's output window) in which you can draw graphics. You can move the graphics origin and the current graphics cursor position outside of this area, but any graphics drawn outside the graphics viewport are invisible.

By default, the graphics viewport is the full size of BASIC's output window and the graphics origin is at the bottom left-hand corner of that window. You may change the size of the graphics viewport with the VDU 24 command and the graphics origin with the VDU 29 command or ORIGIN statement. The action of these commands is described later in the Text and graphics viewports sub-section.

You can move or draw to any point, whether or not this point is outside the current graphics viewport. If you draw a line to a point outside the graphics viewport, the line is not displayed beyond the boundary of the viewport.

Drawing a line

You can use the DRAW statement to draw a straight line on the screen in any mode except MODE 7. The keyword DRAW is followed by the X and Y coordinates of the end of the line. The coordinates must be in the range -32768 to +32767.
The start of the line is the current graphics cursor position. This is either the end of the last line drawn or a point specified by a previous MOVE or PLOT statement.

The line is drawn in the current graphics foreground colour. This colour can be changed using the GCOL statement.

```
DRAW x,y
MOVE 200,300
DRAW 640,800
```

The DRAW statement is identical to PLOT 5.

Alternatively you can draw a straight line by specifying both ends:

```
LINE x1,y1,x2,y2
LINE 200,300,640,800
```

### Moving the graphics cursor

You can move the graphics cursor to an absolute position without drawing a line with the MOVE statement.

```
MOVE X,Y
MOVE 124,327
```

The MOVE statement is identical to PLOT 4.

### Plotting modes

PLOT is a multi-purpose drawing statement. Three numbers follow the PLOT statement: the first specifies the type of point, line or shape to be drawn; the second and third give the X and Y coordinates to be used. The coordinates must be in the range -32768 to +32767.

```
PLOT mode,X,Y
```

As previously described, the two most commonly used statements, PLOT 4 and PLOT 5, have the duplicate keywords MOVE and DRAW. The keywords CIRCLE ,
ELLIPSE, FILL, LINE and RECTANGLE also duplicate certain of the PLOT commands.

The available PLOT modes are as follows:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-7</td>
<td>Basic line drawing modes.</td>
</tr>
<tr>
<td>8-63</td>
<td>Enhanced line drawing modes.</td>
</tr>
<tr>
<td>64-71</td>
<td>Plot a single point.</td>
</tr>
<tr>
<td>72-79</td>
<td>Horizontal line fill to non-background.</td>
</tr>
<tr>
<td>80-87</td>
<td>Plot and fill a triangle.</td>
</tr>
<tr>
<td>88-95</td>
<td>Horizontal line fill to background right.</td>
</tr>
<tr>
<td>96-103</td>
<td>Plot and fill an axis-aligned rectangle.</td>
</tr>
<tr>
<td>104-111</td>
<td>Horizontal line fill to foreground.</td>
</tr>
<tr>
<td>112-119</td>
<td>Plot and fill a parallelogram.</td>
</tr>
<tr>
<td>120-127</td>
<td>Horizontal line fill to non-foreground right.</td>
</tr>
<tr>
<td>128-135</td>
<td>Flood-fill to non-background.</td>
</tr>
<tr>
<td>136-143</td>
<td>Flood-fill to foreground.</td>
</tr>
<tr>
<td>144-151</td>
<td>Draw a circle.</td>
</tr>
<tr>
<td>152-159</td>
<td>Plot and fill a disc.</td>
</tr>
<tr>
<td>160-167</td>
<td>Draw a circular arc.</td>
</tr>
<tr>
<td>168-175</td>
<td>Plot and fill a segment.</td>
</tr>
<tr>
<td>176-183</td>
<td>Plot and fill a sector.</td>
</tr>
<tr>
<td>185/189</td>
<td>Move a rectangular block.</td>
</tr>
<tr>
<td>187/191</td>
<td>Copy a rectangular block.</td>
</tr>
<tr>
<td>192-199</td>
<td>Draw an outline axis-aligned ellipse.</td>
</tr>
<tr>
<td>200-207</td>
<td>Plot and fill a solid axis-aligned ellipse.</td>
</tr>
<tr>
<td>249/253</td>
<td>Swap a rectangular block.</td>
</tr>
</tbody>
</table>

**Basic line drawing modes - PLOT 0 to 7**
### Mode  Action

0  Move relative to the last point.

1  Draw a line, in the current graphics foreground colour, relative to the last point.

2  Draw a line, in the logical inverse colour, relative to the last point.

3  Draw a line, in the current graphics background colour, relative to the last point.

4  Move to the absolute position.

5  Draw a line, in the current graphics foreground colour, to the absolute coordinates specified by X and Y.

6  Draw a line, in the logical inverse colour, to the absolute coordinates specified by X and Y.

7  Draw a line, in the current graphics background colour, to the absolute coordinates specified by X and Y.

Lines are drawn from the current graphics cursor position (the last point 'visited') to the specified X,Y coordinates. The coordinates must be in the range -32768 to +32767.

### Enhanced line drawing modes - PLOT 8 to 63

<table>
<thead>
<tr>
<th>Mode</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-15</td>
<td>As 0-7, except that the last point is omitted.</td>
</tr>
<tr>
<td>16-31</td>
<td>As 0-15, but the line is drawn dotted.</td>
</tr>
<tr>
<td>32-47</td>
<td>As 0-15, but the line is drawn dashed.</td>
</tr>
<tr>
<td>48-63</td>
<td>As 0-15, but the line is drawn broken.</td>
</tr>
</tbody>
</table>

There may seem little point in omitting the last point (modes 8-15) but this can be useful in 'inverse' or 'exclusive-or' plotting modes. For example, when drawing a polygon in an 'exclusive-or' mode this can ensure that each vertex is correctly plotted.
Note that dotted, dashed and broken lines can only be drawn at the (default) thickness of one pixel. If you increase the thickness using VDU 23,23 lines are always drawn continuous.

**Plot a single point - PLOT 64 to 71**

As 0-7, but plot a single point. Exceptionally, the size of the point plotted depends on the MODE in use: it is always one eighth of the width of a character cell by one eighth of the height of a character cell. PLOT X,Y is synonymous with PLOT 69,X,Y.

If you want to plot a single *pixel* you can use:

```
MOVE X,Y:DRAW X,Y or
LINE X,Y,X,Y
```

**Horizontal line fill to non-background - PLOT 72 to 79**

The graphics cursor first moves to the specified coordinate (relative or absolute). Next it moves left until the first non-background pixel is encountered. Then it moves right until the first non-background pixel is encountered. Finally, a line is drawn between these two points. Note that in *BBC BASIC for Windows* the graphics cursor is moved to the specified X,Y position *not* the end of the drawn line. This is different from the behaviour of the BBC Micro and of BBC BASIC (86).

**Triangle plot and fill - PLOT 80 to 87**

As 0-7, but plot and fill a triangle. The triangle is formed from the given X,Y coordinates and the previous two points 'visited', however they were specified. The coordinates must be in the range -32768 to +32767.

The following example plots and fills the triangle illustrated in the current graphics foreground colour.
If the PLOT number is 81 or 85, the triangle is drawn and filled in the current foreground colour.

**Horizontal line fill to background right - PLOT 88 to 95**

The graphics cursor moves to the specified coordinate. It then moves to the right until the first background coloured pixel is encountered. Finally, a line is drawn between these two points. Note that in *BBC BASIC for Windows* the graphics cursor is moved to the specified X,Y position **not** the end of the drawn line. This is different from the behaviour of the BBC Micro and of BBC BASIC (86).

**Rectangle plot and fill - PLOT 96 to 103**

Plot and fill an axis-aligned rectangle. The opposite corners are at the point plotted and the point last 'visited', however it was specified.

The following example plots and fills the rectangle illustrated in the current graphics foreground colour.

If the PLOT number is 97 or 101, the rectangle will be drawn and filled in the current foreground colour.

See also the RECTANGLE FILL statement.

**Horizontal line fill to foreground - PLOT 104 to 111**

The graphics cursor moves to the specified coordinate. It then moves to the right until the first foreground coloured pixel is encountered. It then moves left until the
first foreground coloured pixel is encountered. Finally, a line is drawn between these last two points. Note that in *BBC BASIC for Windows* the graphics cursor is moved to the specified X,Y position **not** the end of the drawn line. This is different from the behaviour of the BBC Micro and of BBC BASIC (86).

**Parallelogram plot and fill - PLOT 112 to 119**

Plot and fill a parallelogram whose vertices are defined by the previous two points visited and the point specified. For reference purposes, call the previous two points visited point1 and point2 in order, and the point specified in the PLOT command, point3. The x and y coordinates of the fourth point are calculated by:

\[
\text{point3} - \text{point2} + \text{point1}
\]

The order in which the points are visited is important.

The following example plots and fills the parallelogram illustrated in the current graphics foreground colour.

![Parallelogram example](image)

MOVE 150,200
MOVE 250,350
PLOT 117,500,300

If the PLOT number is 113 or 117, the parallelogram will be drawn and filled in the current foreground colour.

**Horizontal line fill to non-foreground right - PLOT 120 to 127**

The graphics cursor moves to the specified coordinate. It then moves to the right until the first non-foreground coloured pixel is encountered. Finally, a line is drawn between these two points. Note that in *BBC BASIC for Windows* the graphics cursor is moved to the specified X,Y position **not** the end of the drawn line. This is different from the behaviour of the BBC Micro and of BBC BASIC (86).

**Flood fill to non-background - PLOT 128 to 135**
Flood fill starting at the given point and continuing until non-background pixels are found. This may fail if the colour being used to fill can itself be filled, or if the background colour is not available as a 'solid' colour and has been approximated by 'dithering'.

See also the FILL statement.

**Flood fill to foreground - PLOT 136 to 143**

Flood fill starting at the given point and continuing until foreground pixels are found. This may fail if the colour being used to fill can itself be filled.

**Draw circle - PLOT 144 to 151**

Draw a circle on the screen with the centre at the current graphics cursor position. The supplied X,Y coordinates specify either a point on the circumference of the circle or its radius. If the 'absolute' plot codes (148 to 151) are used the point specified will be on the circumference of the circle. For example,

```
MOVE x1,y1
PLOT 149,x2,y2
```

will draw a circle centred on x1,y1 with the point x2,y2 in its circumference. If the 'relative' plot codes (144 to 147) are used either of the parameters may be used to specify the radius. For example, either

```
MOVE x,y
PLOT 145,R,0
```

or

```
MOVE x,y
PLOT 145,0,R
```

will draw a circle of radius R centred about the point x,y.

See also the CIRCLE statement.

**Draw solid disc - PLOT 152 to 159**
Draw a disc on the screen with its centre at the current graphics cursor position. A point on the circumference or the radius of the disc may be specified in a similar manner to a circle.

If plot codes 156 to 159 are used, the point specified will be on the circumference of the disc. If plot codes 152 to 155 are used, the non-zero parameter is interpreted as the radius of the disc.

See also the CIRCLE FILL statement.

**Draw circular arc - PLOT 160 to 167**

Draw an arc defined such that the previous two points visited correspond to the centre of the circle and the start of the arc respectively (i.e. the second point is on the circumference of the circle). The specified X,Y position lies anywhere on the line from the centre of the circle to the end of the arc. The arc is always drawn counterclockwise.

The following example draws the arc illustrated in the current graphics foreground colour:

```move 150,200
move 500,300
plot 165,250,350```

**Draw solid segment - PLOT 168 to 175**

Plot and fill a segment, i.e. the area between an arc and the chord joining the ends of the arc. The previous two points visited correspond to the centre of the circle and the start of the arc respectively (i.e. the second point is on the circumference of the circle). The specified X,Y position lies anywhere on the line from the centre of the circle to the end of the arc. The segment is always drawn counterclockwise.
The following example draws the segment illustrated in the current graphics foreground colour:

MOVE 150,200
MOVE 500,300
PLOT 173,250,350

Draw solid sector - PLOT 176 to 183

Plot and fill a sector, i.e. the area between an arc and the two radii to the ends of the arc. The previous two points visited correspond to the centre of the circle and the start of the arc respectively (i.e. the second point is on the circumference of the circle). The specified X,Y position lies anywhere on the line from the centre of the circle to the end of the arc. The sector is always drawn counterclockwise.

The following example draws the sector illustrated in the current graphics foreground colour:

MOVE 150,200
MOVE 500,300
PLOT 181,250,350

Move rectangular block - PLOT 185 and 189

Move the contents of a rectangular region. The previous two points visited define
diagonally-opposite corners of the source region. The specified X,Y position defines
the bottom-left corner of the destination region. The source region is cleared to the
current graphics background colour. The source and destination rectangles may
overlap.

See also the RECTANGLE FILL ... TO statement.

**Copy rectangular block - PLOT 187 and 191**

Copy the contents of a rectangular region. The previous two points visited define
diagonally-opposite corners of the source region. The specified X,Y position defines
the bottom-left corner of the destination region. The source region is left unmodified
(unless the source and destination rectangles overlap, in which case the source
region is overwritten).

See also the RECTANGLE ... TO statement.

**Swap rectangular block - PLOT 249 and 253**

Swap the contents of two rectangular regions. The previous two points visited define
diagonally-opposite corners of the first region. The specified X,Y position defines
the bottom-left corner of the second region. If the first and second regions overlap,
what was originally in the first region is preserved.

See also the RECTANGLE SWAP ... TO statement.

**Draw outline ellipse - PLOT 192 to 199**

Draw an outline axis-aligned ellipse such that the previous two points visited
correspond to the centre of the ellipse and a point on a vertical tangent to the ellipse
(i.e. the horizontal radius). The specified X,Y position lies anywhere on a horizontal
tangent to the ellipse (i.e. the vertical radius). *BBC BASIC for Windows* can draw
only axis-aligned ellipses; to draw an angled ellipse use the ELLIPSE library
routines.

The following example draws the ellipse illustrated in the current graphics
foreground colour:
MOVE 250,200
MOVE 500,300
PLOT 197,150,350

See also the ELLIPSE statement.

**Draw solid ellipse - PLOT 200 to 207**

Plot and fill an axis-aligned ellipse such that the previous two points visited correspond to the centre of the ellipse and a point on a vertical tangent to the ellipse (i.e. the horizontal radius). The specified X,Y position lies anywhere on a horizontal tangent to the ellipse (i.e. the vertical radius). *BBC BASIC for Windows* can draw only axis-aligned ellipses; to draw an angled ellipse use the ELLIPSE library routines.

The following example draws the filled ellipse illustrated in the current graphics foreground colour:

MOVE 250,200
MOVE 500,300
PLOT 205,150,350

See also the ELLIPSE FILL statement.

**Draw relative**

The 'draw relative' commands act relative to the last point plotted. Suppose, for example, the current X and Y coordinates were 200, 300. The command
would draw a line in the current graphics foreground colour from 200, 300 to 200,450.

**Logical inverse colour**

The meaning of *logical inverse colour* depends on the selected mode, as follows:

**Two colour modes**

<table>
<thead>
<tr>
<th>Logical</th>
<th>Inverse</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Four colour modes**

<table>
<thead>
<tr>
<th>Logical</th>
<th>Inverse</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

**Sixteen colour modes**
Reading a pixel's colour

The POINT function returns a number giving the logical colour of the screen at the coordinates specified. If the point is outside the graphics viewport, then -1 is returned.

There must not be a space between POINT and the opening bracket.

If two or more palette entries have the same (or nearly the same) physical colour, POINT may not return the expected value.

```
colour=POINT(X,Y)
IF POINT(X,Y)=3 THEN 300
```

You can use POINT to find out the colour of the screen at the specified point and take action accordingly. In an adventure game, for example, the swamps may be marked in green. If the explorer ventured into a green area he must be in the swamp and 'swamp type demons' would be activated.

See also the TINT function.
Text and graphics viewports

Introduction

Since the original BBC BASIC used the term 'window' differently from the sense in which it is used in the context of Microsoft Windows™ there is room for confusion. In this manual the word 'window' is used in its Windows™ sense and the word 'viewport' is used to indicate a rectangular region of the window within which text or graphics are displayed. For historical reasons the word 'screen' is sometimes used to mean BASIC's output window.

BBC BASIC for Windows writes text to the text viewport and graphics to the graphics viewport. The text and graphics viewports can overlap. By default, the text and graphics viewports are the full size of BASIC's output window, but their size and position may be changed.

The text and graphics viewports may be restored to their default state with the VDU 26 command (but see the important note there).

The CLS command clears the text viewport to the current text background colour. The CLG command clears the graphics viewport to the current graphics background colour with the current graphics background plotting action (set by GCOL ). If the text and graphics viewports overlap, the background colour of the overlapping area will be set by the last clear command issued.

Text viewport

The text origin is the top left-hand corner of the current text viewport. The reference point for defining a text viewport is the top left-hand corner of BASIC's output window.

Define text viewport

The VDU 28 command defines a text viewport. The four bytes following VDU 28 define the X and Y coordinates of the bottom left corner of the viewport and the X and Y coordinates of the top right corner of the viewport, in that order. The coordinates are measured in 'character positions'.
If the text cursor is outside the new viewport, it is moved to the new home position (top left of the viewport). If it is inside the new viewport, it is not moved.

The following example defines a text viewport with the bottom left corner at 0,15 (X across, Y down) and the top right corner at 30,3 (X across, Y down).

```
VDU 28,0,15,30,3
```

**Clear text viewport**

The CLS (or VDU 12 ) command clears the text viewport to the current text background colour. If you subsequently change the text background colour, the background of each character cell will be set to the new background colour as each new character is displayed. When the viewport scrolls, the 'empty' line will have the new background colour.

**Graphics viewport**

The graphics origin is initially the bottom left-hand corner of BASIC's output window. A graphics viewport is defined with reference to the current graphics origin. The graphics origin may be set with the VDU 29 command or the ORIGIN statement.

**Define graphics viewport**

In modes other than MODE 7, the VDU 24 command defines a graphics viewport. The four words (pairs of bytes) following the VDU 24 command are the X and Y coordinates of the bottom left corner of the viewport and the X and Y coordinates of the top right corner of the viewport in that order. The coordinates are with respect to the current graphics origin; they must be in the range -32768 to +32767.

It is particularly easy to select invalid viewport limits if the graphics origin has been moved. It is advisable, therefore, to precede a VDU 24 command with a VDU 29,0;0; or ORIGIN 0,0 command to reset the graphics origin.

The following example defines a graphics viewport with the bottom left corner at 200,100 (X,Y) and the top right corner at 500,300 (X,Y).

```
VDU 24,200;100;500;300;
```
Note the use of semicolons in this VDU command; their significance is explained under VDU in the keywords section. In particular, *be very careful not to omit the final semicolon*.

**Clear graphics viewport**

The CLG (or VDU 16) command clears the graphics viewport to the current graphics background colour using the current background plotting action (set by GCOL). If you change the graphics background colour, you will probably not notice the change until you next clear the graphics viewport. However, some of the PLOT modes use the current background colour.

**Changing the graphics origin**

In the graphics modes, ORIGIN (or VDU 29) moves the graphics origin to the specified coordinates. Subsequent graphics commands operate with respect to this origin. The coordinates must be in the range -32768 to +32767.

The following statements both set the graphics origin to X=640, Y=400:

```
ORIGIN 640,400
VDU 29,640;400;
```

Note the use of semicolons in the VDU command. Their significance is explained under VDU in the keywords section.

**Positioning the cursor**

**Text cursor**

The text cursor may be positioned within the text viewport with the TAB(X,Y) command. For example;

```
PRINT TAB(10,15)"Fred"
```

will print the word 'Fred' starting at column 15 of row 10. The top left hand corner of the text viewport is 0,0.
Graphics cursor

Within the graphics viewport, the graphics cursor is positioned by the MOVE, DRAW or PLOT commands.

When printing characters at the graphics cursor position, the top left-hand corner of the character cell is positioned at the graphics cursor position.

Homing the cursor

In VDU 4 mode, VDU 30 homes the text cursor to the top left corner of the text viewport. In VDU 5 mode, VDU 30 homes the graphics cursor to the top left corner of the graphics viewport.
Writing text to the screen

Text positioning

In all modes except MODE 7, text may be written at the text or graphics cursor position. In MODE 7 there is only a text cursor.

Text at text cursor

VDU 4 causes text to be written at the text cursor position in the normal way. This is the default mode.

The characters are 'opaque' and they can be positioned only at text coordinates within the text viewport. The text within the viewport scrolls as necessary.

Text at graphics cursor

VDU 5 causes text to be written at the graphics cursor position.

The characters have a 'transparent' background and they are plotted according to the current foreground GCOL rules and colour. VDU 127 (DEL) is an exception; it backspaces and deletes just as it does in the normal text mode using the current background GCOL colour and action.

In the graphics modes, characters may be positioned at any graphics coordinate within the graphics viewport. The top left of the character cell is the reference point. Characters are clipped to the limits of the graphics viewport if necessary. No scrolling takes place.

Changing the font

You can change the text font using the *FONT command, for example:

*FONT Courier New,16

Note that selecting a proportional spaced font may affect the operation of certain features, such as INPUT editing, positioning of text using TAB(X) and reading the current character with GET(X,Y).
Selecting the character set

You can select the required character set (ANSI, OEM or UTF-8) using the VDU 23,22 command. Selecting the UTF-8 (Unicode) character set may affect the operation of certain features, such as positioning of text using TAB(X).

User-defined characters

Characters from &20 to &FF (32 to 255) may be programmed using the VDU 23 command (programmable characters are not available in MODE 7).

Current character

The easiest way to discover the character located at a specific position in the output window is to use the GET(X,Y) or GET$(X,Y) function. However BBC BASIC for Windows also emulates the BBC Micro OSBYTE call which returns the ASCII code of the character at the current text cursor position:

\[
A\%=&87 \\
\text{char} = (\text{USR}(&FFF4) \text{ AND } &FF00) \text{ DIV } 256
\]

The 'Read Character' function is the only OSBYTE function currently available and it is not, therefore, strictly necessary to set A% to the function number. However, doing so makes the program compatible with any future enhancements of BBC BASIC for Windows.

Character dot pattern

BBC BASIC for Windows emulates the BBC Micro OSWORD call which reads the 'dot pattern' of a character. The pattern of both standard and user defined characters may be read using this call, but the 'standard' characters return the pattern of an internal 8x8 character font, not that of the actual characters which would be displayed.

In order to use the OSWORD call, you need to define a 9 byte area of memory in which the character code is passed and the dot pattern is returned. As with the BBC Micro, the OSWORD function number is passed in A% and the address of the parameter block is passed in X% and Y%. The least significant byte is passed in X%.
and the most significant in Y%

The following program segment:

- Defines a 9 byte parameter block called 'pattern'.
- Loads the character ASCII code into byte-0 of the parameter block.
- Sets A% to the OSWORD call number &A..
- Loads X% and Y% with the least and most significant bytes of the address of the parameter block.
- Calls the 'OSWORD' address &FFF1.

```
DIM pattern 8
?pattern=character
A%=&A
X%=pattern MOD 256:Y%=pattern DIV 256
CALL &FFF1
```

The character dot pattern is returned in the 8 bytes 'pattern?1' to 'pattern?8'.

The 'Get Character Pattern' function is the only OSWORD call available and it is not, therefore, strictly necessary to set A% to the function number. However, doing so makes the program compatible with any future enhancements of *BBC BASIC for Windows*. 
Capturing the screen contents

You can 'capture' (i.e. save to the clipboard) the contents of BASIC's output window - in other words what has been displayed by your program - in two ways:

- To save the entire contents of the window as **graphics** use the normal 'snapshot' facility provided by Windows™, i.e. hold down ALT and press the Print Screen key. You can view, save to disk or print the contents of the clipboard using almost any graphics display program, for example Microsoft™ Paint, which is supplied with Windows™. Normally, you simply have to select Paste from the Edit menu.

- To save the **textual** part of the window, i.e. that which was written as characters rather than graphics, hold down CTRL and press the TAB key. The window contents are saved to the clipboard as Unicode text, and may be Pasted into almost any Windows™ application. Note that this does not work reliably if the text was written using a proportional-spaced font.

*This facility does not work under Windows™ 95, 98 or Me.*
Pixel-perfect graphics

Because of differences in the underlying hardware and Operating System, the graphics statements of BBC BASIC for Windows may not have identical effects to those of other versions of BBC BASIC. In most cases the differences will be insignificant (usually no more than one pixel) but occasionally even such small differences may matter.

In earlier versions of BBC BASIC, the size and shape of the physical pixels varied with the selected screen mode. For example in MODE 1 the pixels were twice as wide as in MODE 0. This does not apply in the case of BBC BASIC for Windows: the size of the physical pixels is determined solely by the current Display Settings. Although changing BASIC's screen mode can affect the size of text characters, and the maximum number of colours available, the pixels remain the same size: one pixel always corresponds to two BASIC graphics units. For example, MODEs 0, 1 and 2 are all 640 pixels wide by 512 pixels high, or 1280 graphics units wide by 1024 graphics units high.

BBC BASIC for Windows could have attempted to emulate more closely (in software) the detailed characteristics of the BBC Microcomputer and Acorn Archimedes screens. However this would have adversely affected performance and made it difficult to mix the use of BASIC graphics statements with native Windows™ API functions. Instead, the graphics statements are mapped to Windows functions in the most straightforward way, consistent with a reasonable degree of compatibility.

When these small differences matter, the following sections describe in detail how BBC BASIC for Windows behaves, and therefore makes it possible to write programs which have precisely the desired effect on the screen.

General considerations

Because each physical pixel corresponds to two BASIC graphics units, when you need to determine precisely which pixels are affected it is best to ensure that only even-numbered coordinates are used. This applies both to the setting of the graphics origin (ORIGIN or VDU 29 statements) and to the coordinates passed to the various graphics statements. This is particularly important in the case of relative coordinates, where the use of odd-numbered values can result in truncation effects.
which accumulate to cause quite large errors.

**Straight lines**

Straight lines are drawn between the specified end-points, which are normally *inclusive*, i.e. the line includes both end-points. In general, it is difficult to predict exactly which pixels will be plotted to create the line, because this depends on the precise algorithm used by Windows™; it may even depend on the type of video card fitted in your computer and the version of its driver software. When it is essential to determine which pixels are 'lit', the line should be drawn as a series of individual points. However, in the case of horizontal and vertical lines the pixels lit should be predictable.

Whereas on earlier versions of BBC BASIC the thickness of a line may vary depending on the screen MODE in use, this is not the case with BBC BASIC for Windows where it will always be (about) one pixel thick. If you need to draw a thicker line you can use VDU 23,23.

**Outline shapes**

Outline shapes, such as triangles, rectangles and parallelograms, consist of a series of straight-line segments between the vertices. Each of these lines can be considered in isolation, and the same factors as described above for straight lines apply. Note in particular that although the RECTANGLE statement specifies the width and height of the rectangle, since the end-points of the lines forming the edges are *inclusive*, the actual dimensions (measured to the outer edges of the lit pixels) will be one pixel greater than those specified.

**Solid (filled) shapes**

Solid shapes, such as filled triangles, rectangles and parallelograms, are drawn using the Windows™ *Polygon* function. This function treats the supplied vertices in a special way: vertices to the *left* and *top* of the polygon are *inclusive* of the plotted shape whereas vertices to the *right* and *bottom* are *exclusive*.

For example, if plotting a rectangle with:

```
MOVE 100,300 : PLOT 101,300,100
```
the point 100,300 (the top-left corner) will be **inside** the rectangle but 300,100 (the bottom-right corner) will be **outside** the rectangle. Put another way, the width and height of the rectangle will both be 200 graphics units (100 pixels) with the bottom-left corner of the rectangle at coordinates 100,102 (*not* 100,100). This behaviour is independent of the order in which the vertices are specified, therefore the position and size of the rectangle will be same for all the following methods:

```
MOVE 300,300 : PLOT 101,100,100
MOVE 100,100 : PLOT 97,200,200
RECTANGLE FILL 100,100,200,200
```

**Plotting single points**

PLOT modes 64 to 71 plot a single 'dot' at the specified coordinates. Because these commands are commonly used to draw complex shapes, *BBC BASIC for Windows* (exceptionally) plots a different-sized 'dot' depending on the screen MODE in use. If it did not do so, programs written for earlier versions of BBC BASIC would be likely to plot a series of separated pixels rather than a solid shape. The 'dot' plotted is a rectangle whose width and height are one-eighth of the width and height, respectively, of a character cell in that particular screen mode. For example, in MODEs 2 and 5 it is 4 pixels wide by 2 pixels high (8 x 4 graphics units); if necessary you can change the size using the `@vdu.d.x%` and `@vdu.d.y%` system variables. The specified coordinates correspond to the top-left corner of this rectangle.

If you need to plot a single (hardware) pixel, you can do that by drawing a 'line' of zero length:

```
MOVE X,Y : DRAW X,Y
```

or

```
LINE X,Y,X,Y
```

**User-defined characters**

User-defined characters (created with the VDU 23 command) vary in size according
to the screen mode. For example, in MODE 0 they are 8 pixels wide by 16 pixels high, in MODE 1 they are 16 pixels by 16 pixels and in MODE 2 they are 32 pixels wide by 16 pixels high. When plotted as graphics (VDU 5 mode) the reference position is the top-left corner of the character cell.
Compatibility limitations

The characteristics of Windows™ displays impose a few limitations on the compatibility between BBC BASIC for Windows plotting modes and those of some earlier versions of BBC BASIC.

You can usually choose your display settings (either in the Control Panel or by right-clicking on the desktop) from a range of options. In particular, you can select the 'Color palette' (not to be confused with BBC BASIC's logical palette) from two or more of the following choices:

- '16 Color' (4 bits per pixel)
- '256 Color' (8 bits per pixel)
- 'High Color' (16 bits per pixel)
- 'True Color' (24 or 32 bits per pixel)

(With recent versions of Windows™ you can run BBC BASIC in a 256-colour paletted mode by right-clicking on the program's icon, selecting Properties... Compatibility and checking the 'Run in 256 colors' box.)

Depending on the selected mode, the 'logical' plotting operations (OR, AND, exclusive-OR and invert, as selected by GCOL) can behave in a way different from that which you might expect.

Paletted modes

In the 16 and 256 colour modes the physical display uses a hardware palette, otherwise known as a colour look-up table. In these modes each pixel is represented by a number (4 bits or 8 bits) which is used to 'address' a table of colours. The actual colour displayed depends on the contents of the table. It is therefore impossible to have more than 16 or 256 different colours, respectively, displayed at any one time. The advantage of these modes is that the amount of memory required to hold the screen contents is reduced, and the speed of access is increased. They also permit 'palette animation', the changing of what is displayed by altering the palette contents rather than the screen memory itself.

These modes work in a very similar way to those of the BBC Micro, and of BBC BASIC (86), but with one very important difference: the hardware palette must be shared between the desktop and all the applications which are running (i.e. those
whose windows are displayed). As a result, the logical colour numbers used by BBC BASIC, which with the earlier versions would have corresponded directly to physical palette addresses, are 'mapped' by Windows™ to different physical addresses.

This is not apparent when you are simply 'plotting' a colour (GCOL 0,n or GCOL n) but can affect the behaviour of the 'logical' plotting modes (OR, AND and so on). This is best illustrated by example. Suppose you are using BBC BASIC's default palette, in which colour 1 is red, colour 2 is green and colour 3 is yellow. If you firstly use GCOL 0,1 to plot in red, then use GCOL 1,2 to combine the existing colour with colour 2 using the OR plotting mode, you would expect the end result to be colour 3 (1 OR 2) i.e. yellow. If, however, Windows has mapped BASIC's colour 1 to physical colour 4, and BASIC's colour 2 to physical colour 5, then the OR operation will result in colour 5 (4 OR 5) which is green, not yellow.

Non-paletted modes

In the 'high color' and 'true color' modes there is no physical palette; the screen memory holds the RGB (Red, Green, Blue) colour directly. The range of colours which may be displayed at any one time is greatly increased, but the amount of memory used is more than in the paletted modes, and the speed is reduced. Palette animation is not available.

BBC BASIC still uses a 'logical' palette of 16 colours, but this is purely for software convenience and to maximise compatibility between paletted and non-paletted displays. The actual number of colours which can be displayed in BASIC's output window at any one time is limited only by the display settings. Changing the contents of BASIC's palette affects only things subsequently plotted; the existing screen contents are unaffected.

In these modes the 'logical' plotting operations (OR, AND etc.) operate on the physical RGB colour, not on the logical colour. Suppose you have selected (using VDU 19 or COLOUR ) colour 1 to be red, colour 2 to be green and colour 3 to be blue. If you firstly use GCOL 0,1 to plot in red, then use GCOL 1,2 to combine the existing colour with colour 2 using the OR plotting mode, you would expect the end result to be colour 3 (1 OR 2) i.e. blue. However, what will actually happen is that the RGB value for red will be logically ORed with the RGB value for green, resulting in yellow, not blue.
Compatibility summary

In general you should be aware that the 'logical' plotting modes (OR, AND, exclusive-OR and invert) operate on the number stored in the physical screen memory (whether hardware palette index or RGB colour value) and not on BASIC's colour number. Combinations of logical operations which restore a colour to its original value (for example a double inversion) will work, but otherwise the result may not be what you intended.
Introduction to keywords

The BBC BASIC for Windows keywords are listed alphabetically for ease of reference. Where appropriate, the abbreviated form is also given (acceptance of abbreviated keywords is controlled by the Customize menu command).

The syntax and any associated keywords are listed at the end of each section.

Syntax

Syntax definitions are given at the end of each keyword section. Rather than use formal Backus-Naur Form (BNF) definitions, which many people find somewhat confusing, we have attempted to use descriptions which, whilst being reasonably precise, are readable by the majority of users. To those who would have preferred 'the real thing' - we apologise.

Symbols

The following symbols have special meaning in the syntax definitions.

- `{}` The enclosed item may be repeated zero or more times.
- `[ ]` The items enclosed are optional, they may occur zero or one time.
- `|` Indicates alternatives; one of which must be used.
- `<stmt>` Means a BBC BASIC for Windows statement.
- `<var>` Means a numeric or string variable.
- `<exp>` Means an expression like PI*radius*height+2 or name$+"FRED"+CHR$(&0D). It can also be a single variable or constant like 23 or "FRED".
- `<label>` Means a label identifying a program line.
- `<l-num>` Means a line number in a BBC BASIC for Windows program.
- `<k-num>` Means the number of one of the programmable keys.
- `<n-const>` Means a numeric constant like '26.4' or '256'.
- `<n-var>` Means a numeric variable like 'size' or 'weight'.
<numeric>  Means a <n-const> or a <n-var> or an expression combining them. For example: PI*radius+2.66

<s-const>  Means a string constant like "FRED".

<s-var>  Means a string variable like 'address$'.

<string>  Means a <s-const> or a <s-var> or an expression combining them. For example: name$+add$+"Phone".

<array()>  Means an entire array. For example: alpha$().

<struct{}>  Means an entire structure. For example: my_struct{}.

<t-cond>  Means a 'testable condition'. In other words, something which is tested in an IF, UNTIL or WHILE clause. Since BBC BASIC does not have true Boolean variables, a <numeric> can be used anywhere a <t-cond> is specified.

<name>  Means a valid variable name.

<member>  Means a valid structure member.

<d:>  Means a disk drive name (A: to P:).

<dirtry>  Means a directory path-name.

<afsp>  Means an ambiguous file specifier.

<ufsp>  Means an unambiguous file specifier.

<nchr>  Means a character valid for use in a name. 0 to 9, A to Z, a to z and underline.
**ABS**

A function which returns the absolute value of its numeric argument.

\[
X = \text{ABS}(\text{deficit}) \\
\text{length} = \text{ABS}(X_1-X_2)
\]

This function converts negative numbers into positive ones. It can be used to give the difference between two numbers without regard to the sign of the answer.

It is particularly useful when you want to know the difference between two values, but you don't know which is the larger. For instance, if \(X=6\) and \(Y=10\) then the following examples would give the same result.

\[
difference = \text{ABS}(X-Y) \\
difference = \text{ABS}(Y-X)
\]

You can use this function to check that a calculated answer is within certain limits of a specified value. For example, suppose you wanted to check that 'answer' was equal to 'ideal' plus or minus (up to) 0.5. One way would be:

\[
\text{IF } \text{answer}>\text{ideal}-0.5 \text{ AND } \text{answer}<\text{ideal}+0.5 \text{ THEN}....
\]

However, the following example would be a more elegant solution.

\[
\text{IF } \text{ABS(}\text{answer-ideal}\text{)}<0.5 \text{ THEN}....
\]

**Syntax**

\[
<n-var>=\text{ABS}(<numeric>)
\]

**Associated Keywords**

SGN
ACS

A function which returns, in radians, the arc-cosine of its numeric argument. The permitted range of the argument is -1 to +1.

If you know the cosine of the angle, this function will tell you the angle (in radians). Unfortunately, you cannot do this with complete certainty because two angles within the range +/- PI (+/- 180 degrees) can have the same cosine. This means that one cosine has two associated angles.

The following diagram illustrates the problem:

![Diagram showing the relationship between cosine, sine, and tangent in different quadrants.]

Within the four quadrants, there are two angles which have the same cosine, two with the same sine and two with the same tangent. When you are working back from the cosine, sine or tangent you don't know which of the two possible angles is correct.

By convention, ACS gives a result in the top two quadrants (0 to PI; 0 to 180 degrees) and ASN and ATN in the right-hand two quadrants (-PI/2 to +PI/2; -90 to +90 degrees).

In the example below, 'radian_angle' becomes equal to the angle (in radians) whose cosine is 'y'.

\[
\text{radian\_angle} = \text{ACS}(y)
\]
You can convert the answer to degrees by using the DEG function (or multiplying by 180/PI).

\[
\text{degree\_angle=DEG(ACS(y))}
\]

**Syntax**

\[
<n\text{-var}>=\text{ACS(<numeric>)}
\]

**Associated Keywords**

ASN, ATN, SIN, COS, TAN, RAD, DEG
ADVAL

A function which returns information about the joystick, if fitted, or the number of free spaces in a buffer. The information returned depends upon the argument passed to ADVAL.

**ADVAL positive argument**

If the argument is positive (1 to 3), ADVAL returns a value related to the position of the joystick, if any:

ADVAL(1) returns the joystick X-position.
ADVAL(2) returns the joystick Y-position.
ADVAL(3) returns the joystick Z-position.

```ml
joyx=ADVAL(1)
level=ADVAL(chnumber)
```

**ADVAL zero argument**

ADVAL(0) returns a value in the range 0 to 15 corresponding to the state of the four joystick buttons. Zero is returned if no buttons are pressed and 15 is returned if all four buttons are pressed.

**ADVAL negative argument**

ADVAL(-1) returns the number of free spaces in the keyboard buffer. If the keyboard buffer is empty, ADVAL(-1) will return 255. If the keyboard buffer is full, ADVAL(-1) will return 0.

ADVAL(-5) returns the number of free bytes in the channel 0 SOUND queue. If the sound queue is empty, ADVAL(-5) will return 16. If the queue is full, ADVAL(-5) will return 0. Each sound event uses 4 bytes.

ADVAL(-6) returns the number of free bytes in the channel 1 SOUND queue. If the sound queue is empty, ADVAL(-6) will return 16. If the queue is full, ADVAL(-6) will return 0. Each sound event uses 4 bytes.
ADVAL(-7) returns the number of free bytes in the channel 2 SOUND queue. If the sound queue is empty, ADVAL(-7) will return 16. If the queue is full, ADVAL(-7) will return 0. Each sound event uses 4 bytes.

ADVAL(-8) returns the number of free bytes in the channel 3 SOUND queue. If the sound queue is empty, ADVAL(-8) will return 16. If the queue is full, ADVAL(-8) will return 0. Each sound event uses 4 bytes.

Syntax

<n-var>=ADVAL(<numeric>)

Associated Keywords

SOUND , INPUT , GET
AND

An operator which performs a bitwise or logical AND between two items. The two operands are internally converted to 32-bit integers before the AND operation. For each of the 32 bits AND uses the following truth-table:

<table>
<thead>
<tr>
<th>Input A</th>
<th>Input B</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

You can use AND as a logical operator or as a 'bit-by-bit' (bitwise) operator. The operands can be boolean (logical) or numeric.

```plaintext
answer=num1 AND num2
char=byte AND &7F
IF (num AND &F0)
test=(count=3 AND total=5)
```

In the following example program segment, AND is used as a bitwise operator to remove the most significant bit of a byte read from a file before writing it to another file. This is useful for converting some word-processor files into standard ASCII format.

```plaintext
byte=BGET#infile AND &7F
BPUT#outfile,byte
```

BBC BASIC does not have true boolean variables; this can lead to confusion at times (see NOT for more details).

In the example below, the operands are boolean (logical). In other words, the result of the tests (IF) A=2 and (IF) B=3 is either TRUE or FALSE. The result of this example will be TRUE if A=2 and B=3.
answer=(A=2 AND B=3)

The brackets are not necessary, they have been included to make the example easier to follow.

The second example is similar to the first, but in the more familiar surroundings of an IF statement.

IF A=2 AND B=3 THEN 110

or

answer= A=2 AND B=3 (without brackets this time)
IF answer THEN 110

The final example, uses the AND in a similar fashion to the numeric operators (+, -, etc).

A=X AND 11

Suppose X was -20, the AND operation would be:

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]

= 8

Syntax

<n-var>=<numeric> AND <numeric>
<n-var>AND=<numeric>

Associated Keywords

EOR , OR , FALSE , TRUE , NOT
ASC

A function which returns the ASCII value of the first character of the argument string. If the string is empty then -1 will be returned.

A computer only understands numbers. In order to deal with characters, each character is assigned a code number. For example (in the ASCII code table) the character 'A' is given the code number 65 (decimal).

Different types of computer use different numbers for the characters. The codes used for PC Compatible computers are those defined by the American Standard Code for Information Interchange (ASCII). See the Table of ASCII codes section.

You could use this function to convert ASCII codes to some other coding scheme.

```
    ascii_code=ASC("H")  Result would be 72
    X=ASC("HELLO")      Result would be 72
    name$="FRED"
    ascii_code=ASC(name$) Result would be 70
    X=ASC"e"            Result would be 101
    X=ASC(MID$(A$,position)) Result depends on A$ and position.
```

ASC is the complement of CHR$.

Syntax

```
<n-var>=ASC(<string>)
```

Associated Keywords

CHRS, STRS, VAL
ASN

A function which returns, in radians, the arc sine of its numeric argument. The permitted range of the argument is -1 to +1.

By convention, the result will be in the range -PI/2 to +PI/2 (-90 to +90 degrees).

If you know the sine of the angle, this function will tell you the angle (in radians). Unfortunately, you cannot do this with complete certainty because one sine has two associated angles (see ACS for details).

In the example below, 'radian_angle' becomes equal to the angle (in radians) whose sine is 'y'.

\[
radian\_angle = \text{ASN}(y)
\]

You can convert the answer to degrees by using the DEG function (the DEG function is equivalent to multiplying by 180/PI). The example below is similar to the first one, but the angle is in degrees.

\[
degree\_angle = \text{DEG}(\text{ASN}(y))
\]

Syntax

<var>=ASN(<numeric>)

Associated Keywords

ACS, ATN, SIN, COS, TAN, RAD, DEG
**ATN**

A function which returns, in radians, the arc tangent of its numeric argument. The permitted range of the argument is from minus infinity to plus infinity.

By convention, the result will be in the range -PI/2 to +PI/2 (-90 to +90 degrees).

If you know the tangent of the angle, this function will tell you the angle (in radians).

As the magnitude of the argument (tangent) becomes very large (approaches + or - infinity) the accuracy diminishes.

In the example below, 'radian_angle' becomes equal to the angle (in radians) whose tangent is 'y'.

\[
\text{radian\_angle} = \text{ATN}(y)
\]

You can convert the answer to degrees by using the DEG function (the DEG function is equivalent to multiplying by 180/PI). The example below is similar to the first one, but the angle is in degrees.

\[
\text{degree\_angle} = \text{DEG}(\text{ATN}(y))
\]

**Syntax**

\[
<n\text{-var}>=\text{ATN}(<\text{numeric}>)
\]

**Associated Keywords**

ACS, ASN, SIN, COS, TAN, RAD, DEG
A function which reads a byte from the file whose channel number is its argument. The file pointer is incremented after the byte has been read.

E=BGET#n
aux=BGET#3

You must normally have opened a file using OPENOUT, OPENIN or OPENUP before you use this statement. (See these keywords and the BBC BASIC for Windows Disk files section for details).

You can use BGET# to read single bytes from a disk file. This enables you to read back small integers which have been 'packed' into fewer than 5 bytes (see BPUT# ). It is also very useful if you need to perform some conversion operation on a file. Each byte read is numeric, but you can use CHR$ (BGET#n) to convert it to a string.

The input file in the example below is a text file produced by a word-processor. Words to be underlined are 'bracketed' with ^S. The program produces an output file suitable for a printer which expects such words to be bracketed by ^Y. You could, of course, perform several such translations in one program.

REM Open i/p and o/p files. End if error.
infile=OPENIN "WSFILE.DOC"
IF infile=0 THEN END
outfile=OPENOUT "BROTH.DOC"
IF outfile=0 THEN END
REM Process file, converting ^S to ^Y
REPEAT
    temp=BGET#infile :REM Read byte
    IF temp=&13 THEN temp=&19 :REM Convert ^S
    BPUT#outfile,temp :REM Write byte
UNTIL temp=&1A OR EOF#infile :REM ^Z
CLOSE#0 :REM Close all files
END

To make the program more useful, it could ask for the names of the input and output files at 'run time':
INPUT "Enter name of INPUT file " infile$
INPUT "Enter name of OUTPUT file " outfile$
REM Open i/p and o/p files. End if error.
infile=OPENIN(infile$)
IF infile=0 THEN END
outfile=OPENOUT(outfile$)
IF outfile=0 THEN END :
REM Process file, converting ^S to ^Y
REPEAT
temp=BGET#infile :REM Read byte
  IF temp=&13 THEN temp=&19 :REM Convert ^S
  BPUT#outfile,temp :REM Write byte
UNTIL temp=&1A OR EOF#infile :REM ^Z
CLOSE#0 :REM Close all files
END

Syntax
<n-var>=BGET#<numeric>

Associated Keywords

OPENIN, OPENUP, OPENOUT, CLOSE#, PRINT#, INPUT#, BGET#, EXT#, PTR#, EOF#, GET$#
A statement which writes a byte or string to the data file whose channel number is the first argument. If the second argument is numeric, its least significant byte is written to the file. If the second argument is a string, the contents of the string are written to the file, followed by a line-feed character (CHR$10); adding a terminating semicolon suppresses the line-feed. The file pointer is incremented after the bytes have been written.

```
BPUT#E,32
BPUT#staff_file,A/256
BPUT#chn,A$
BPUT#N,"a string";
```

Before you use this statement you must normally have opened a file for output using OPENOUT or OPENUP (see these keywords and the Disk files section for details).

You can use this statement to write single bytes to a disk file. The number that is sent to the file is in the range 0 to 255. Real numbers are converted internally to integers and the top three bytes are 'masked off'. Each byte written is numeric, but you can use ASC (character$) to convert (the first character of) 'character$' to a number.

Alternatively BPUT# can be used to write a character string to a file. The difference between PRINT#file,string$ and BPUT#file,string$ is that PRINT# appends a carriage-return character (CHR$13) whereas BPUT# appends a line-feed character (CHR$10). Adding a terminating semicolon (;) causes the contents of the string to be written to the file with nothing appended.

The example below is a program segment that 'packs' an integer number between 0 and 65535 (&FFFF) into two bytes, least significant byte first. The file must have already been opened for output and the channel number stored in 'fnum'. The integer variable number% contains the value to be written to the file.

```
BPUT#fnum,number% MOD 256
BPUT#fnum,number% DIV 256
```
Syntax

BPUT#<numeric>,<numeric>
BPUT#<numeric>,<string>[;]

Associated Keywords

OPENIN , OPENUP , OPENOUT , CLOSE# , PRINT# , INPUT# , BGET# ,
EXT# , PTR# , EOF# , GET$#
**BY**

Used in conjunction with DRAW, FILL, MOVE or PLOT, indicates that the supplied X and Y coordinates are *relative* (offsets from the current point) rather than *absolute*.

BY is also used with the GET$#channel function to specify the number of bytes to be read from the file.

**Syntax**

```
DRAW BY <numeric>,<numeric>
FILL BY <numeric>,<numeric>
MOVE BY <numeric>,<numeric>
PLOT BY <numeric>,<numeric>
```

**Associated Keywords**

```
DRAW, FILL, GET$, MOVE, PLOT
```
CALL

A statement to call a machine code (assembly language) subroutine, or to run BBC BASIC code in a separate file.

CALL Muldiv,A,B,C,D
CALL &FFE3
CALL 12340,A$,M,J$

The processor's EAX, EBX, ECX and EDX registers are initialised to the contents of A%, B%, C% and D% respectively (see also USR). The processor's flags register is initialised to the contents of F%; however, you cannot disable interrupts nor enter single-step mode by setting F% to an appropriate value because this could affect the operation of Windows.

Your machine-code routine should return to BBC BASIC for Windows with a near return instruction (RET) but for compatibility with BBC BASIC (86) a far return instruction (RETF) will also work.

**Operating system interface**

CALL and USR operate differently when addresses in the range &FF00 to &FFFF are used. See the Operating System Interface section for more details and an example of the use of CALL.

**Parameter table**

CALL sets up a table in RAM containing details of the parameters. The base pointer (EBP register) is set to the memory address of this parameter table.

Variables included in the parameter list need not have been declared before the CALL statement.

The parameter types are:
<table>
<thead>
<tr>
<th>Code</th>
<th>Parameter type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>byte (unsigned 8 bits)</td>
<td>?A%</td>
</tr>
<tr>
<td>1</td>
<td>byte (unsigned 8 bits)</td>
<td>A&amp;</td>
</tr>
<tr>
<td>4</td>
<td>32-bit signed integer</td>
<td>!A% or A%</td>
</tr>
<tr>
<td>40</td>
<td>64-bit signed integer</td>
<td>A%%</td>
</tr>
<tr>
<td>5</td>
<td>40-bit variant</td>
<td>A^1</td>
</tr>
<tr>
<td>8</td>
<td>64-bit variant</td>
<td>A^2 or A#</td>
</tr>
<tr>
<td>10</td>
<td>80-bit variant</td>
<td>A^3</td>
</tr>
<tr>
<td>24</td>
<td>structure</td>
<td>A{}</td>
</tr>
<tr>
<td>36</td>
<td>function or procedure</td>
<td>FNa or PROCa</td>
</tr>
<tr>
<td>65</td>
<td>byte array</td>
<td>A&amp;(</td>
</tr>
<tr>
<td>68</td>
<td>32-bit integer array</td>
<td>A%(</td>
</tr>
<tr>
<td>104</td>
<td>64-bit integer array</td>
<td>A%%(</td>
</tr>
<tr>
<td>69</td>
<td>40-bit variant array</td>
<td>A()^1</td>
</tr>
<tr>
<td>72</td>
<td>64-bit variant array</td>
<td>A()^2 or A##()</td>
</tr>
<tr>
<td>74</td>
<td>80-bit variant array</td>
<td>A()^3</td>
</tr>
<tr>
<td>88</td>
<td>structure array</td>
<td>A(){}</td>
</tr>
<tr>
<td>100</td>
<td>function or procedure</td>
<td>FNa() or PROCa()</td>
</tr>
<tr>
<td>128</td>
<td>fixed string (CR terminated)</td>
<td>$A%</td>
</tr>
<tr>
<td>129</td>
<td>movable string, 6-byte descriptor</td>
<td>A$^1.2</td>
</tr>
<tr>
<td>136</td>
<td>movable string, 8-byte descriptor</td>
<td>A$^3</td>
</tr>
<tr>
<td>130</td>
<td>fixed string (NUL terminated)</td>
<td>$$A%</td>
</tr>
<tr>
<td>193</td>
<td>string array, 6-byte descriptors</td>
<td>A$(^1.2</td>
</tr>
<tr>
<td>200</td>
<td>string array, 8-byte descriptors</td>
<td>A$(^3</td>
</tr>
</tbody>
</table>

1. *BBC BASIC for Windows version 5.95a or earlier* in *FLOAT 40 mode.*
2. *BBC BASIC for Windows version 5.95a or earlier* in *FLOAT 64 mode.*
3. *BBC BASIC for Windows version 6.00a or later*.
On entry to the subroutine the parameter table contains the following values:

- **Number of parameters**: 1 byte (at EBP)
- **Parameter type**: 1 byte (at EBP+1)
- **Parameter address**: 4 bytes (at EBP+2 to EBP+5, LSB first)
- **Parameter type**: repeated as often
- **Parameter address**: as necessary.

The parameter address given is the absolute address at which the item is stored. In the case of a movable string (type 129 or 136), it is the address of the 'string descriptor' (see below). In the case of an array it is the address of a pointer to the array's parameter block (see the Format of data in memory section).

**Parameter formats**

- Integer variables are stored in twos complement format with their least significant byte first.
- Byte variables are stored as a single unsigned byte.
- Fixed strings are stored as the characters of the string followed by a carriage return (&0D) or NUL (&00).
- Real variables are stored in 5-byte, 8-byte or 10-byte binary floating point format with their least significant byte first. See the Format of data in memory section for more details and examples.

In the case of a movable string (normal string variable), the parameter address points to a 6-byte or 8-byte 'string descriptor'. This descriptor gives the address of the start of the string (4 bytes, LSB first) and the current length of the string (2 bytes or 4 bytes, LSB first) in that order.

**Running BASIC code in a separate file**

You can execute BBC BASIC code in an external file using the following syntax:
The contents of the file are loaded into memory, executed, and then discarded. As with the INSTALL statement the file must be in BBC BASIC internal (tokenised) format, and line numbers and labels are not recognised. Whereas INSTALLed modules should only contain procedure and function definitions, CALLeled modules should normally not contain any procedure or function definitions (nor should they INSTALL modules which do). Attempting to execute such a procedure or function after the CALLeled module has returned could easily crash BASIC.

One use for this facility is to keep 'constant' and structure definitions (such as might be used, for example, by calls to Windows™ API functions ) in separate files rather than including them in every program that uses them. In this way it is easier to maintain and update the definitions, and your program size is reduced. This is similar to the use of library files for packaging functions which you might want to call from many programs.

You should normally prefix the filename with either @lib$ (for standard definitions) or @dir$ (for definitions specific to your program) so that the file will be loaded from the correct location irrespective of the current directory setting:

```
CALL @lib$+"windefs"
CALL @dir$+"mydefs"
```

This also makes it easier to embed the files within a compiled executable. If you crunch your compiled program, using the **abbreviate names** option, it is important that any CALLeled files are also crunched in such a way that any variable and structure names declared in the files agree with those used in your main program. You can achieve that using the crunch embedded files option of the Compile command.

The CALLeled program will return control to the calling program when execution 'drops off the end' of the code, or if a RETURN statement is executed. CALL may be nested, that is program files executed using CALL may themselves include CALL statements.

**Syntax**
CALL <numeric>{,<n-var>|<s-var>|<array>()|<struct()>>
CALL <string>

**Associated Keywords**

INSTALL, RETURN, USR
CASE

A statement which results in different actions depending on the value of a numeric or string variable. The value is compared with each of the alternatives given; if it matches then the appropriate statements are executed. If the value does not match any of the alternatives, the statements following the OTHERWISE statement (if any) are executed. CASE ... ENDCASE clauses can be nested.

WHEN and ENDCASE must be the first thing on the program line and OF must be the last thing on the line (it cannot even be followed by a REMark).

```
CASE toss% OF
   WHEN 0 : coin$ = "tails"
   WHEN 1 : coin$ = "heads"
   OTHERWISE coin$ = "cheat"
ENDCASE
CASE direction$ OF
   WHEN "left","LEFT" :
      PRINT "turn left"
   WHEN "right","RIGHT" :
      PRINT "Turn right"
   OTHERWISE
      PRINT "Straight on"
ENDCASE
```

Syntax

```
CASE <numeric>|<string> OF
   WHEN <numeric>|<string>{,<numeric>|<string>} : {<stmt>}
   {<stmt>}
   WHEN <numeric>|<string>{,<numeric>|<string>} : {<stmt>}
   {<stmt>}
   OTHERWISE {<stmt>}
   {<stmt>}
ENDCASE
```

Associated Keywords

ENDCASE , IF , OF , ON , OTHERWISE , WHEN
A statement which loads and runs the program whose name is specified in the argument.

CHAIN "GAME1"
CHAIN A$

The program file must be in *BBC BASIC for Windows* internal (tokenised) format.

All but the static variables @% to Z% are CLEAR ed.

CHAIN sets ON ERROR OFF before chaining the specified program.

RUN may be used as an alternative to CHAIN.

You can use CHAIN (or RUN) to link program modules together. This allows you to write modular programs which would, if written in one piece, be too large for the memory available or too unwieldy.

Passing data between CHAINed programs can be a bit of a problem because all but the static variables are cleared by CHAIN.

If you wish to pass large amounts of data between CHAINed programs, you should use a data file. However, if the amount of data to be passed is small and you do not wish to suffer the time penalty of using a data file, you can pass data to the CHAINed program by using the indirection operators to store them at known addresses. The safest way to do this is to move HIMEM down and store common data at the top of memory.

The following sample program segment moves HIMEM down 1000 bytes and stores the input and output file names in the memory above HIMEM. There is, of course, still plenty of room for other data in this area.

```
HIMEM=HIMEM-1000
$HIMEM=in_file$
$(HIMEM+256)=out_file$
CHAIN "NEXTPROG"
```
Syntax

CHAIN <string>

Associated Keywords

CALL, INSTALL, RUN
**CHR$**

A function which returns a string of length 1 containing the ASCII character specified by the least significant byte of the numeric argument.

\[
\begin{align*}
A$ &= \text{CHR$(72)$} \\
B$ &= \text{CHR$(12)$} \\
C$ &= \text{CHR$(A/200)$}
\end{align*}
\]

CHR$ generates an ASCII character (symbol, letter, number character, control character, etc) from the number given. The number specifies the position of the generated character in the ASCII table. For example:

\[
\text{char$}$ = \text{CHR$(65)$}
\]

will set char$ equal to the character 'A'. You can use CHR$ to send a special character to the terminal or printer (generally, VDU is better for sending characters to the screen). For example,

\[
\text{CHR$(7)$}
\]

will generate the ASCII character ^G. So,

\[
\text{PRINT "ERROR"+CHR$(7)$}
\]

will print the message 'ERROR' and sound the PC's 'bell'.

CHR$ is the complement of ASC.

**Syntax**

\[
<s-var> = \text{CHR$(<\text{numeric})$}
\]

**Associated Keywords**
CIRCLE

A statement which draws a circle or disc (filled circle) in all screen modes except MODE 7. CIRCLE is followed by the X and Y coordinates of the centre of the circle and the radius. To draw a filled (solid) circle rather than an outline circle, use CIRCLE FILL .

The graphics origin (X=0, Y=0) is normally the bottom left of the 'screen' (BASIC's output window). The origin can be changed using the VDU29 command or the ORIGIN statement. See the Graphics and colours section for more details.

The circle or disc is drawn in the current graphics foreground colour. This colour can be changed using the GCOL statement.

CIRCLE x,y,r is equivalent to MOVE x,y : PLOT 145,r,0
CIRCLE FILL x,y,r is equivalent to MOVE x,y : PLOT 153,r,0

CIRCLE 200,300,40
CIRCLE FILL 300,400,100

Syntax

CIRCLE [FILL] <numeric>,<numeric>,<numeric>

Associated Keywords

ELLIPSE, FILL, GCOL, MOVE, PLOT
CLEAR

A statement which clears all the dynamically declared variables, including strings. CLEAR does not affect the static variables.

The CLEAR command tells *BBC BASIC for Windows* to 'forget' about ALL the dynamic variables used so far. This includes strings, arrays and structures, but the static variables (@% to Z%) are not altered.

You can use the indirection operators to store integers and strings at known addresses and these will not be affected by CLEAR. However, you will need to 'protect' the area of memory used. One way to do this is to move HIMEM down. See CHAIN for an example.

CLEAR sets LOMEM equal to TOP.

Syntax

```
CLEAR
```

Associated Keywords

None
CLOSE# A statement used to close a data file. CLOSE #0 will close all data files.

CLOSE#file_num
CLOSE#0

You use CLOSE# to tell BBC BASIC for Windows that you have completely finished with a data file for this phase of the program. Any data still in the file buffer is written to the file before the file is closed.

You can open and close a file several times within one program, but it is generally considered 'better form' not to close a file until you have finally finished with it. However, if you wish to CLEAR the variables, it is simpler if you close the data files first.

You should also close data files before chaining another program. CHAIN does not automatically close data files, but it does clear the variables in which the channel numbers were stored. You can still access the open file if you have used one of the static variables (A% to Z%) to store the channel number. Alternatively, you could reserve an area of memory (by moving HIMEM down for example) and use the byte indirection operator to store the channel number. See the keyword CHAIN for more details.

END or 'dropping off' the end of a program will also close all open data files. However, QUIT and STOP do not close data files.

CLOSE is also used in the ON CLOSE statement.

Syntax

CLOSE#<numeric>

Associated Keywords

OPENIN , OPENUP , OPENOUT , PRINT# , INPUT# , BPUT# , BGET# ,
EXT# , PTR# , EOF# , ON CLOSE
CLG

A statement which clears the graphics viewport and sets it to the currently selected graphics background colour using the current background plotting action (set by GCOL). The position of the graphics cursor is unchanged.

See the Graphics and colours section or GCOL for more information on graphics colours.

Syntax

```
CLG
```

Associated Keywords

```
CLS, GCOL
```
CLS

A statement which clears the text viewport and sets it to the currently selected text background colour. The text cursor is moved to the 'home' position (0,0) at the top left-hand corner of the text viewport.

See the Graphics and colours section or COLOUR for more information on text colours.

Syntax

CLS

Associated Keywords

CLG, COLOUR
A statement which sets the text foreground and background colour or modifies the colour palette.

When COLOUR is followed by one value the text foreground or background colour is set. If the value is less than 128, the text foreground colour is set to that value. If the number is 128 or greater, the text background colour is set to value-128.

There are up to 16 logical colours (numbered 0 to 15), but the number available varies with the mode; see the section on text colours for details.

```
COLOUR 1 : REM sets the text foreground colour to 1
COLOUR 130 : REM sets the text background colour to 2
```

COLOUR n is equivalent to VDU 17,n.

When COLOUR is followed by two values the palette is modified using a physical colour number. The first value specifies the logical colour to be changed (0-15) and the second value specifies the physical colour to which it should be mapped (0-15).

```
COLOUR 1,4 : REM set logical colour 1 to blue
```

COLOUR l,p is equivalent to VDU 19,l,p,0,0,0

When COLOUR is followed by four values the palette is modified using an RGB (Red, Green, Blue) colour. The first value specifies the logical colour to be changed (0-15) and the second, third and fourth values specify the red, green and blue components of the colour to which it should be mapped. In each case the value should be in the range 0 (none) to 255 (maximum).

```
COLOUR 1,128,0,128 : REM set logical colour 1 to dark purple
```

COLOUR l,r,g,b is equivalent to VDU 19,l,16,r,g,b

**Syntax**
COLOUR <numeric>
COLOUR <numeric>,<numeric>
COLOUR <numeric>,<numeric>,<numeric>,<numeric>

Associated Keywords

VDU, GCOL, MODE
COS

A function giving the cosine of its radian argument.

\[ X = \cos(\text{angle}) \]

This function returns the cosine of an angle. The angle must be expressed in radians, not degrees.

Whilst the computer is quite happy dealing with angles expressed in radians, you may prefer to express angles in degrees. You can use the RAD function to convert an angle from degrees to radians.

The example below sets Y to the cosine of the angle 'degree_angle' expressed in degrees.

\[ Y = \cos(\text{RAD(degree_angle)}) \]

Syntax

\[ <n\text{-var}> = \cos(<\text{numeric}>) \]

Associated Keywords

\[ \text{SIN, TAN, ACS, ASN, ATN, DEG, RAD} \]
COUNT

A function returning the number of text characters written to the screen since the last new line.

```
char_count=COUNT
```

Characters with an ASCII value of less than 13 (carriage return/new-line/enter) have no effect on COUNT.

Because control characters above 13 are included in COUNT, you cannot reliably use it to find the position of the cursor on the screen. If you need to know the cursor's horizontal position use the POS function.

The example below prints strings from the string array 'words$'. The strings are printed on the same line until the line length exceeds 65. When the line length is in excess of 65, a new-line is printed.

```
PRINT
FOR i=1 TO 1000
    PRINT words$(i);
    IF COUNT>65 THEN PRINT
NEXT
```

Note that COUNT may not return the expected value when UTF-8 mode is enabled (it counts bytes rather than characters).

Syntax

```
<n-var>=COUNT
```

Associated Keywords

POS
DATA

A program object which must precede all lists of data for use by the READ statement. DATA must be used at the start of a program line.

As for INPUT, string values may be quoted or unquoted. However, quotes need to be used if the string contains commas or leading spaces.

Numeric values may include calculation so long as there are no keywords.

Data items in the list should be separated by a comma.

```
DATA 10.7,2,HELLO," THIS IS A COMMA","1/3,PRINT
DATA " This is a string with leading spaces."
```

You can use DATA in conjunction with READ to include data in your program which you may need to change from time to time, but which does not need to be different every time you run the program.

The following example program segment reads through a list of names looking for the name in 'name$'. If the name is found, the name and age are printed. If not, an error message is printed.

```
DATA FRED,17,BILL,21,ALLISON,21,NOEL,32
DATA JOAN,26,JOHN,19,WENDY,35,ZZZZ,0
REPEAT
READ list$,age
IF list$=name$ THEN PRINT name$,age
UNTIL list$=name$ OR list$="ZZZZ"
IF list$="ZZZZ" PRINT "Name not in list"
```

**Syntax**

```
DATA <s-const>|<n-const>{,<s-const>|<n-const>}
```

**Associated Keywords**
DEF

A program object which must precede declaration of a user defined function (FN ) or procedure (PROC ). DEF must be used at the start of a program line.

If DEF is encountered during execution, the rest of the line is ignored. As a consequence, single line definitions can be put anywhere in the program.

Multi-line definitions must not be executed. The safest place to put multi-line definitions is at the end of the main program after the END statement.

There is no speed advantage to be gained by placing function or procedure definitions at the start of the program.

```
DEF FNMEAN ....
DEF PROCJIM ....
```

The function or procedure name may start with an underline:

```
DEF FN_mean ....
DEF PROC_Jim$ ....
```

Function and procedure names may end with a '$'. This is not compulsory for functions which return strings.

A procedure definition is terminated by the statement ENDPROC .

A function definition is terminated by a statement which starts with an equals (=) sign. The function returns the value of the expression to the right of the equals sign.

For examples of function and procedure declarations, see FN and PROC . For a general explanation refer to the Procedures and functions section.

Syntax

```
DEF PROC<name>[([RETURN]<s-var>|<n-var>{[,]([RETURN]<s-var>|<n-var>)}])
DEF FN<name>[([RETURN]<s-var>|<n-var>{[,]([RETURN]<s-var>|<n-var>)}])
```
Associated Keywords

ENDPROC, FN, PROC, RETURN
DEG

A function which converts radians to degrees.

\[
\text{degree\_angle} = \DEG(\frac{\pi}{2}) \\
X = \DEG(\ATN(1))
\]

You can use this function to convert an angle expressed in radians to degrees. One radian is approximately 57 degrees (actually \(\frac{180}{\pi}\)). \(\pi/2\) radians is 90 degrees and \(\pi\) radians is 180 degrees.

Using DEG is equivalent to multiplying the radian value by \(\frac{180}{\pi}\), but the result is calculated internally to a greater accuracy.

See ACS, ASN and ATN for further examples of the use of DEG.

Syntax

\[
<n\text{-var}> = \DEG(<\text{numeric}>)
\]

Associated Keywords

RAD, SIN, COS, TAN, ACS, ASN, ATN, PI
DIM can be used as a statement or as a function. When used as a statement, DIM has three different forms: the first declares an array, the second declares a structure and the third reserves an area of memory for special applications. When used as a function, DIM returns the number of dimensions in an array, the size of a particular dimension or the size of a structure.

Dimensioning arrays

The DIM statement is used to declare arrays. Arrays must be pre-declared before use and once declared their dimensions cannot be changed (with the exception of LOCAL and PRIVATE arrays). Arrays may be integer numeric, byte numeric, real numeric or string and they may be multi-dimensional. However, you cannot mix integers, bytes, reals and/or strings in the same array.

```plaintext
DIM A(2), Ab(2,3), A$(2,3,4), A%(3,4,5,6)
```

All elements in the array are initialised to zero (for numeric arrays) or to empty strings (for string arrays). Note that you can repeat an array declaration so long as the dimensions are identical to those specified when it was first declared. In this case the contents of the array remain unchanged: they are *not* initialised.

The subscript base is 0, so `DIM X(12)` defines an array of 13 elements whose subscript value can be from 0 to 12 inclusive.

Arrays are like lists or tables. A list of names is a single dimension array. In other words, there is only one column - the names. Its single dimension in a DIM statement would be the maximum number of names you expected in the table less 1.

If you wanted to describe the position of the pieces on a chess board you could use a two dimensional array. The two dimensions would represent the row (numbered 0 to 7) and the column (also numbered 0 to 7). The contents of each 'cell' of the array would indicate the presence (if any) of a piece and its value.

```plaintext
DIM chess_board(7,7)
```
Such an array would only represent the chess board at one moment of play. If you wanted to represent a series of board positions you would need to use a three-dimensional array. The third dimension would represent the 'move number'. Each move would use about 320 bytes of memory, so you could record 40 moves in about 12.5k bytes.

```
DIM chess_game(7,7,40)
```

The 'DIM space' error will occur if you attempt to declare an array for which there is insufficient memory. In this case you can (usually) increase the value of HIMEM to make more memory available.

### Declaring structures

The DIM statement is used to declare structures and arrays of structures. Structures must be pre-declared before use and once declared their format cannot be changed (with the exception of LOCAL and PRIVATE structures). Structures may contain integer numeric, byte numeric, real numeric, string, array or sub-structure members, and arrays of structures can be one-dimensional or multi-dimensional. For more details see the Structures section.

```
DIM mystruct{one,two%,three$,four(3),five%(1,2),six$(9),seven{a,b%})
DIM structarray{(10)one,two%,three$}
DIM newstruct{}=protostruct{}
DIM multistruct{(3,2)}=protostruct{}
```

All members of the structure are initialised to zero (for numeric members) or to empty strings (for string members). Note that you can repeat a structure declaration so long as the format is identical to that specified when it was first declared. In this case the contents of the structure remain unchanged: they are *not* initialised.

The 'DIM space' error will occur if you attempt to declare a structure or structure array for which there is insufficient memory. In this case you can (usually) increase the value of HIMEM to make more memory available.

### Reserving an area of memory

The DIM statement can be used to reserve an area of memory which the interpreter will not then use. The variable specified in the DIM statement is set to the start
address of this memory area. This reserved area can then be used by the indirection operators, assembly language code, etc.

The example below reserves 68 bytes of memory from the heap and sets A% equal to the address of the first byte. Thus A%?0 to A%?67 are free for use by the program (68 bytes in all):

```
DIM A% 67
```

The amount of memory reserved is one byte greater than the value specified. DIM P% -1 is a special case; it reserves zero bytes of memory. This is of more use than you might think, since it tells you the limit of the dynamic variable allocation (the heap) so far. Thus,

```
DIM P% -1
PRINT HIMEM-P%
```

is the equivalent of PRINT HIMEM-END (or PRINT FREE(0) in some other dialects of BASIC).

Memory reserved in this way causes irreversible loss of heap space, therefore you must be very careful not to reserve the memory multiple times when once will suffice. This is a particular danger when DIM is used within a function or procedure.

See the Assembler section for a more detailed description of the use of DIM for reserving memory for machine code programs.

**Reserving a temporary area of memory**

If you want to reserve a temporary block of memory for use within a function or procedure you can use use the following construct:

```
DIM A% LOCAL 67
```

This reserves 68 bytes from the stack rather than from the heap, and the memory is automatically freed on exit from the function or procedure. This is very useful if, for example, you need to create a temporary parameter block for a Windows API call.
Note that the variable (A% in this case) is not automatically made LOCAL to the function or procedure. In most cases you will want to do that as well:

```plaintext
LOCAL A%
DIM A% LOCAL 67
```

The amount of memory reserved is one byte greater than the value specified. DIM P% LOCAL -1 is a special case; it reserves zero bytes of memory. This is of more use than you might think, since it tells you the current size of the stack:

```plaintext
DIM P% LOCAL -1
PRINT HIMEM-P%
```

**DIM as a function**

You can use DIM as a function to discover the number of dimensions in an array, the size of an array dimension or the size of a structure. The function DIM(array()) returns the number of dimensions in the array (note the use of opening and closing brackets with nothing in between). The function DIM(array(),n) returns the size of dimension 'n', where n is in the range 1 to the number of dimensions. The function DIM(structure{}) returns the size of the structure in bytes.

If an array is declared as follows:

```plaintext
DIM cuboid(4,5,7)
```

the function DIM(cuboid()) will return the value 3, the function DIM(cuboid(),1) will return the value 4, the function DIM(cuboid(),2) will return the value 5 and the function DIM(cuboid(),3) will return the value 7.

Discovering the size of an array is of particular interest when a whole array is passed as a parameter to a function or procedure. For example, if you write a general-purpose function to calculate the geometric mean of the values in a numeric array, the function must know the total number of elements in the array. See the section on passing arrays to functions/procedures for an example.

If a structure is declared as follows:
the function $\text{DIM(mystruct{}}))$ would return the value 12 (4 + 6 + 2).

**Syntax**

\[
\begin{align*}
\text{DIM } & \text{<n-var>|<s-var>{(<numeric>},{(<numeric>)})} \\
\text{DIM } & \text{<name>{<member>{(<member>)}}} \\
\text{DIM } & \text{<struct{}>=<struct{}}} \\
\text{DIM } & \text{<n-var> [LOCAL] <numeric>} \\
\text{<n-var> = DIM(<array()>[,<numeric>])} \\
\text{<n-var> = DIM(<struct()>)}
\end{align*}
\]

**Associated Keywords**

CLEAR, HIMEM, LOCAL, LOMEM
DIV

An operator returning the integer quotient of two items. The result is always an integer.

\[ X = A \div B \]
\[ y = (\text{top} + \text{bottom} + 1) \div 2 \]

You can use this function to give the 'whole number' part of the answer to a division. For example,

\[ 21 \div 4 \]

would give 5 (with a 'remainder' of 1).

Whilst it is possible to use DIV with real numbers, it is really intended for use with integers. If you do use real numbers, *BBC BASIC for Windows* converts them to integers by truncation before DIViding them.

**Syntax**

\[ <\text{n-var}> = <\text{numeric}> \div <\text{numeric}> \]
\[ <\text{n-var}> \div = <\text{numeric}> \]

**Associated Keywords**

MOD
DRAW

A statement which draws a line on the screen (in all modes except MODE 7). DRAW is followed by the X and Y coordinates of the end of the line; the optional qualifier BY indicates that the coordinates are relative (offsets from the start of the line) rather than absolute. The coordinates must be in the range -32768 to +32767.

The start of the line is the current graphics cursor position, i.e. either the last point 'visited' (for example the end of the last line drawn) or a point explicitly specified by a MOVE statement.

The graphics origin (X=0, Y=0) is normally the bottom left of the 'screen' (BASIC's output window). The origin can be changed using the ORIGIN statement or the VDU 29 command.

The line is drawn in the current graphics foreground colour. This colour can be changed using the GCOL statement. DRAW only draws solid lines; you can draw dotted or dashed lines by using the PLOT statement.

```
DRAW x,y
MOVE 200,300
DRAW 640,800
```

DRAW x,y is equivalent to PLOT 5,x,y.
DRAW BY x,y is equivalent to PLOT 1,x,y.

Syntax

```
DRAW <numeric>,<numeric>
DRAW BY <numeric>,<numeric>
```

Associated Keywords

```
BY , MODE , PLOT , MOVE , CLG , VDU , GCOL
```
ELLIPSE

A statement which draws an outline ellipse or filled ellipse, in all screen modes except MODE 7. ELLIPSE is followed by the X and Y coordinates of the centre of the ellipse, the horizontal radius and the vertical radius, in that order. To draw a filled ellipse use ELLIPSE FILL.

*BBC BASIC for Windows* can draw only axis-aligned ellipses. To draw an angled ellipse use the ELLIPSE library routines.

The graphics origin (X=0, Y=0) is normally the bottom left of the 'screen' (BASIC's output window). The origin can be changed using the VDU29 command or the ORIGIN statement. See the Graphics and colours section for more details.

The ellipse is drawn in the current graphics foreground colour. This colour can be changed using the GCOL statement.

ELLIPSE x,y,a,b is equivalent to MOVE x,y : PLOT 0,a,0 : PLOT 193,0,b
ELLIPSE FILL x,y,a,b is equivalent to MOVE x,y : PLOT 0,a,0 : PLOT 201,0,b

ELLIPSE 200,300,40,50
ELLIPSE FILL 300,400,100,80

Syntax

```
ELLIPSE [FILL] <numeric>,<numeric>,<numeric>,<numeric>
```

Associated Keywords

```
CIRCLE , FILL , GCOL , MOVE , PLOT
```
A statement delimiter which provides an alternative course of action in IF ...THEN, ON...GOSUB, ON...GOTO and ON...PROC statements.

In an IF statement, if the conditional expression evaluates to zero, the statements after ELSE will be executed. This makes the following work:

```
IF A=B THEN B=C ELSE B=D
IF A=B THEN B=C:PRINT"WWW" ELSE B=D:PRINT"QQQ"
IF A=B THEN B=C ELSE IF A=C THEN............
```

In a multi statement line containing more than one IF, the statement(s) after the ELSE delimiter will be actioned if ANY of the tests fail. For instance, the example below would print the error message 'er$' if 'x' did not equal 3 OR if 'a' did not equal 'b'.

```
IF x=3 THEN IF a=b THEN PRINT a$ ELSE PRINT er$
```

If you want to 'nest' the tests, you should use a multi-line IF ... ENDIF statement. The following example would print "Bad" ONLY if x was equal to 3 AND 'a' was not equal to 'b'.

```
IF x=3 THEN
    IF a=b THEN PRINT a$ ELSE PRINT "Bad"
ENDIF
```

ELSE can be used to indicate an alternative course of action in a multi-line IF ... THEN ... ENDIF statement:

```
IF x=3 THEN
    PRINT "x is 3"
ELSE
    PRINT "x is not 3"
ENDIF
```

In this case ELSE must be the first thing on the line. You can cascade multiple IF ...
ENDIF statements to test a number of different alternatives:

```plaintext
IF x=3 THEN
    PRINT "x is 3"
ELSE IF x=4 THEN
    PRINT "x is 4"
ELSE
    PRINT "x is neither 3 nor 4"
ENDIF
ENDIF
```

but in this situation a CASE statement would be a better choice.

You can use ELSE with `ON...GOSUB` , `ON...GOTO` and `ON...PROC` statements to prevent an out of range control variable causing an 'ON range ' error.

```plaintext
ON action GOTO 100, 200, 300 ELSE PRINT "Error"
ON number GOSUB 100,200,300 ELSE PRINT "Error"
ON value PROCa,PROCb,PROCc ELSE PRINT "Error"
```

**Syntax**

```plaintext
IF <t-cond> THEN <stmt> ELSE <stmt>
ELSE [<stmt>]
ON <n-var> GOTO <l-num>{,<l-num>} ELSE <stmt>
ON <n-var> GOSUB <l-num>{,<l-num>} ELSE <stmt>
ON <n-var> PROC<name>{,PROC<name>} ELSE <stmt>
```

**Associated Keywords:**

```plaintext
IF , THEN , ON , ENDIF
```
END

A statement causing the interpreter to return to immediate mode. There can be any number (>=0) of END statements anywhere in a program. END closes all open data files.

END tells *BBC BASIC for Windows* that it has reached the end of the program. You don't have to use END, just 'running out of program' will have the same effect, but it's a bit messy.

You can use END within, for instance, an IF ...THEN ...ELSE statement to stop your program if certain conditions are satisfied. You should also use END to stop *BBC BASIC for Windows* 'running into' any procedure or function definitions at the end of your program.

END can also be used as a function. It returns the address of the first byte above BASIC's dynamic variable area (the heap). These two statements are equivalent:

```plaintext
    E% = END
    DIM E% -1
```

**Syntax**

```plaintext
    END
    <n-var> = END
```

**Associated Keywords**

QUIT , STOP , CLOSE#
ENDCASE

A keyword which terminates a CASE ...ENDCASE clause. ENDCASE must be the first item on the program line.

```
CASE die% OF
  WHEN 1,2,3 : bet$ = "lose"
  WHEN 4,5,6 : bet$ = "win"
  OTHERWISE bet$ = "cheat"
ENDCASE
```

Syntax

```
ENDCASE
```

Associated Keywords

```
CASE , OF , OTHERWISE , WHEN
```
ENDIF

A keyword which terminates a multi-line IF clause. ENDIF must be the first thing on the program line.

Syntax

ENDIF

Associated Keywords

ELSE, IF, THEN
ENDPROC

A statement denoting the end of a procedure.

All LOCAL or PRIVATE variables and the dummy arguments are restored at ENDPROC and the program returns to the statement after the calling statement.

Syntax

    ENDPROC

Associated Keywords

    DEF, FN, LOCAL, PRIVATE, PROC
ENDWHILE

A keyword which terminates a WHILE ... ENDWHILE clause. If the ENDWHILE statement is executed, BBC BASIC jumps to the matching WHILE statement. If the WHILE condition is not met, control is transferred to the statement after the matching ENDWHILE statement.

```basic
WHILE LEFT$(A$,1) = " " A$ = MID$(A$,2) : ENDWHILE
```

Syntax

```
ENDWHILE
```

Associated Keywords

```
REPEAT , UNTIL , WHILE
```
A statement which is used, in conjunction with the SOUND statement, to control the pitch and/or amplitude of a sound whilst it is playing.

During its 'life' the pitch and amplitude (volume) of a sound may change. The variation of pitch with time is called the pitch envelope and the variation of amplitude with time is called the amplitude envelope. In *BBC BASIC for Windows*, there are four phases of amplitude (the attack phase, the decay phase, the sustain phase and the release phase) and three phases of pitch. The four amplitude phases and the three pitch phases are defined separately based on a common 'step' duration.

You may use the ENVELOPE statement to define up to 16 different envelopes. The envelopes are referred to by number in the SOUND statement.

The ENVELOPE statement has 14 parameters. The syntax of the ENVELOPE statement is:

```
ENVELOPE <parameter list>
```

The parameter list has the format shown below; all the parameters must be supplied.

```
N, T, PI1, PI2, PI3, PN1, PN2, PN3, AA, AD, AS, AR, ALA, ALD
```

<table>
<thead>
<tr>
<th>Param</th>
<th>Range</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>1 to 16</td>
<td>Envelope number.</td>
</tr>
<tr>
<td>T (bits 0-6)</td>
<td>0 to 127</td>
<td>Length of each step in hundredths of a second.</td>
</tr>
<tr>
<td>T (bit 7)</td>
<td>0 or 1</td>
<td>0=Auto repeat the pitch envelope. 0=Auto repeat.</td>
</tr>
<tr>
<td>PI1</td>
<td>-128 to 127</td>
<td>Change of pitch per step in section 1.</td>
</tr>
<tr>
<td>PI2</td>
<td>-128 to 127</td>
<td>Change of pitch per step in section 2.</td>
</tr>
<tr>
<td>PI3</td>
<td>-128 to 127</td>
<td>Change of pitch per step in section 3.</td>
</tr>
<tr>
<td>PN1</td>
<td>0 to 255</td>
<td>Number of steps in section 1.</td>
</tr>
<tr>
<td>PN2</td>
<td>0 to 255</td>
<td>Number of steps in section 2.</td>
</tr>
<tr>
<td>PN3</td>
<td>0 to 255</td>
<td>Number of steps in section 3.</td>
</tr>
</tbody>
</table>
The durations of the three phases of the pitch envelope are explicitly specified (PN1, PN2 and PN3 respectively) but the durations of the phases of the amplitude envelope are determined indirectly.

The duration of the 'attack' phase is determined by the target level (ALA) and the rate of attack (AA), i.e. the duration is equal to ALA/AA or the total duration specified in the SOUND statement, whichever is less.

The duration of the 'decay' phase is determined by the target level (ALD) and the rate of decay (AD), i.e. the duration is equal to (ALD-ALA)/AD or what remains of the total duration specified in the SOUND statement, whichever is less.

The duration of the 'sustain' phase is determined by what remains of the total duration specified in the SOUND statement. For example if the total duration is two seconds, and the attack and decay phases lasted a total of one second, the sustain phase will last one second.

The 'release' phase is terminated as soon as another note is ready to play on that channel, otherwise it continues indefinitely or until the amplitude has reduced to zero. A SOUND statement with the 'hold' bit set can be used to force some or all of the release phase to sound even if there are more notes ready to be played.

**Syntax**

```
ENVELOPE <numeric>, <numeric>, <numeric>, <numeric>, <numeric>,
        <numeric>, <numeric>, <numeric>, <numeric>, <numeric>,
        <numeric>, <numeric>, <numeric>, <numeric>
```

**Associated Keywords**

`SOUND`
**EOF#**

A function which will return -1 (TRUE) if the data file whose channel number is the argument is at, or beyond, its end. In other words, when PTR# points beyond the current end of the file. When reading a serial file, EOF# would go true when the last byte of the file had been read.

EOF# is only true if PTR# is set beyond the last byte written to the file. It will NOT be true if an attempt has been made to read from an empty block of a sparse random access file. Because of this, it is difficult to tell which records of a random access file have had data written to them. These files need to be initialised and the unused records marked as empty.

Writing to a byte beyond the current end of file updates the file length immediately, whether the record is physically written to the disk at that time or not. However, the file must be closed in order to ensure that all the data written to it is physically written to the disk.

**Syntax**

<n-var>=EOF#(<numeric>)

**Associated Keywords**

OPENIN, OPENUP, OPENOUT, CLOSE#, INPUT#, READ#, BGET#, GET$#, EXT#, PTR#
EOR

An operator returning the bitwise or logical exclusive-or between two items. The two operands are internally converted to 4 byte integers before the EOR operation. For each of the 32 bits EOR uses the following truth-table:

<table>
<thead>
<tr>
<th>Input A</th>
<th>Input B</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

You can use EOR as a logical operator or as a 'bit-by-bit' (bitwise) operator. The operands can be boolean (logical) or numeric.

```
X=B EOR 4
IF A=2 EOR B=3 THEN 110
```

BBC BASIC does not have true boolean variables; this can lead to confusion at times (see NOT for more details).

In the example below, the operands are boolean (logical) and the result of the tests (IF) A=2 and (IF) B=3 is either TRUE or FALSE.

The result of this example will be FALSE if A=2 and B=3 or A<>2 and B<>3. In other words, the answer will only be TRUE if the results of the two tests are different.

```
answer=(A=2 EOR B=3)
```

The brackets are not necessary, they have been included to make the example easier to follow.

The last example uses EOR in a similar fashion to the numeric operators (+, -, etc).
Suppose X was -20, the EOR operation would be:

\[
\begin{align*}
00000000 & \quad 00000000 & \quad 00000000 & \quad 00001011 \\
11111111 & \quad 11111111 & \quad 11111111 & \quad 11100111 & = -25
\end{align*}
\]

Syntax

\[
<n\text{-var}>=<\text{numeric}> \text{ EOR } <\text{numeric}>
\]

\[
<n\text{-var}>\text{EOR=<numeric>}
\]

Associated Keywords

NOT , AND , OR
**ERL**

A function returning the line number of the line where the last error occurred.

```plaintext
x=ERL
```

If there was an error in a procedure call, the line number of the calling line would be returned, not the line number of the definition.

The number returned by ERL is the line number printed out when *BBC BASIC for Windows* reports an error.

See the Error Handling sub-section for more information on error handling and correction.

**Syntax**

```plaintext
<n-var>=ERL
```

**Associated Keywords**

```
ON ERROR , REPORT , ERR
```
**ERR**

A function returning the error code number of the last error which occurred (see the section entitled Error messages and codes).

\[ \text{x(ERR)} \]

Once you have assumed responsibility for error handling using the ON ERROR statement, you can use this function to discover which error occurred.

See the Error Handling sub-section for more information on error handling and correction.

**Syntax**

\[ \text{<n-var> = ERR} \]

**Associated Keywords**

ON ERROR, ERL, REPORT
ERROR

A statement which can 'force' an error to occur. This can be useful for testing or for adding 'user-defined' errors. ERROR is followed by the error number and the error string:

```
200 ERROR 100,"Fault"
```

Unless errors are trapped using ON ERROR this will result in the message:

```
Fault at line 200
```

and ERR will be set to 100. User-defined errors should normally be given error numbers in the range 100 to 179 so that they don't conflict with the built-in error codes. However, if you specify an error number of zero BBC BASIC for Windows will treat it as a fatal error, i.e. it will not be trapped by the ON ERROR statement and will cause the program to end immediately.

Note that the error string is held in a temporary buffer, which will be overwritten if an INPUT, CALL or READ statement is executed, or if BBC BASIC for Windows returns to immediate mode. This means you should be careful when accessing the error string using REPORT or REPORT$; if necessary copy it into a conventional string variable first.

ERROR is also used in the ON ERROR, ON ERROR LOCAL and RESTORE ERROR statements.

Syntax

```
ERROR <numeric>,<string>
ON ERROR <stmt>{:<stmt>}
ON ERROR LOCAL <stmt>{:<stmt>}
ON ERROR OFF
RESTORE ERROR
```

Associated Keywords
ON ERROR, ON ERROR LOCAL, ERL, ERR, REPORT, RESTORE
A function which applies the interpreter's expression evaluation program to the characters held in the argument string.

```plaintext
X=EVAL("X^Q+Y^P")
X=EVAL"A$+B$
X$=EVAL(A$)
```

In effect, you pass the string to BBC BASIC for Windows's evaluation program and say 'work this out'.

You can use this function to accept and evaluate an expression, such as a mathematical equation, whilst the program is running. You could, for instance, use it in a 'calculator' program to accept and evaluate the calculation you wished to perform. Another use would be in a graph plotting program to accept the mathematical equation you wished to plot.

The example below is a 'bare bones' calculator program which evaluates the expression typed in by the user.

```plaintext
PRINT "This program evaluates the expression"
PRINT "you type in and prints the answer"
REPEAT
  INPUT "Enter an expression" exp$
  IF exp$<"END" PRINT EVAL exp$
UNTIL exp$="END"
END
```

You can only use EVAL to work out functions (like SIN, COS, etc). It won't execute statements like MODE 0, PRINT, etc. Note that EVAL may fail if you use it to evaluate a structure member, since the 'dot' after the structure name may cause it to be interpreted as an abbreviated keyword.

**Syntax**

```plaintext
<n-var>=EVAL(<string>)
<s-var>=EVAL(<string>)
```
Associated Keywords

STR$, VAL
A statement which causes a premature exit from a FOR ...NEXT, REPEAT ...UNTIL or WHILE ...ENDWHILE loop.

EXIT FOR causes execution to continue after the matching NEXT statement, EXIT REPEAT causes execution to continue after the matching UNTIL statement and EXIT WHILE causes execution to continue after the matching ENDWHILE statement. Typically you would use EXIT when a situation occurs which necessitates leaving the loop 'part way through':

```
FOR I% = start% TO finish%
  ...
  IF bool% THEN EXIT FOR
  ...
NEXT I%
REPEAT
  ...
  IF bool% THEN EXIT REPEAT
  ...
UNTIL condition%
WHILE condition%
  ...
  IF bool% THEN EXIT WHILE
  ...
ENDWHILE
```

In the case of EXIT FOR an optional loop variable can be specified, causing execution to continue after the NEXT statement which matches that variable:

```
FOR I% = start% TO finish%
  FOR J% = first% TO last%
    ...
    IF bool% THEN EXIT FOR I%
    ...
  NEXT
  NEXT
REM Execution continues here
```

You can EXIT from a loop even when inside a nested loop of a different kind:

```
REPEAT
```
EXIT is not compatible with the use of non-standard FOR...NEXT constructs such as 'popping' an inner loop or listing multiple loop variables after NEXT. When EXIT is used, every FOR must have a matching NEXT.

Syntax

EXIT FOR [<n-var>]
EXIT REPEAT
EXIT WHILE

Associated Keywords

FOR , REPEAT , WHILE
**EXP**

A function returning 'e' to the power of the argument. The argument must be < 88.7228392 (in *FLOAT 40 mode) or <709.782713 (in *FLOAT 64 mode). The 'natural' number, 'e', is approximately 2.718281828.

\[ Y = \text{EXP}(Z) \]

This function can be used as the 'anti-log' of a natural logarithm. Logarithms are 'traditionally' used for multiplication (by adding the logarithms) and division (by subtracting the logarithms). For example,

\[
\begin{align*}
\log_1 &= \text{LN}(2.5) \\
\log_2 &= \text{LN}(2) \\
\log_3 &= \log_1 + \log_2 \\
\text{answer} &= \text{EXP}(\log_3) \\
\text{PRINT} \ & \text{answer}
\end{align*}
\]

will calculate 2.5*2 by adding their natural logarithms and print the answer.

**Syntax**

\[ <n-var>=\text{EXP}(<numeric>) \]

**Associated Keywords**

LN, LOG
EXT#

A function which returns the total length of the file whose channel number is its argument.

```
length=EXT#f_num
```

In the case of a sparse random-access file, the value returned is the complete file length from byte zero to the last byte written. This may well be greater than the actual amount of data on the disk, but it is the amount of disk space allocated to the file by Windows™.

If you want to write extra data at the end of an existing file, you should open the file with OPENUP and then set the file pointer to the end of the existing data:

```
PTR#channel = EXT#channel
```

The file must have been opened before EXT# can be used to find its length.

EXT# is also used when accessing a communications port (e.g. a serial port). It returns the number of characters which have been received, but not yet read. To prevent blocking you should attempt to read (with BGET# ) no more than this number of characters.

EXT# may also be used on the left of the equals sign to set the file's length:

```
EXT#channel = newlength
```

Syntax

```
<n-var>=EXT#(<numeric>)
EXT#(<numeric>)=<numeric>
```

Associated Keywords

OPENIN , OPENUP , OPENOUT , CLOSE# , PRINT# , INPUT# , READ# ,
FALSE

A function returning the value zero.

flag=FALSE
...
IF flag ...

You can use FALSE in a REPEAT ....UNTIL loop to make the loop repeat for ever. Consider the following example.

    terminator=10
    REPEAT
      PRINT "An endless loop"
    UNTIL terminator=0

Since 'terminator' will never be zero, the result of the test 'terminator=0' will always be FALSE. Thus, the following example has the same effect as the previous one.

    REPEAT
      PRINT "An endless loop"
    UNTIL FALSE

Similarly, since FALSE=0, the following example will also have the same effect, but its meaning is less clear.

    REPEAT
      PRINT "An endless loop"
    UNTIL 0

See the keyword AND for logical tests and their results.

Syntax

<n-var>=FALSE
Associated Keywords

TRUE, EOR, OR, AND, NOT
FILL

A statement which performs a 'flood fill' in the current graphics foreground colour, starting at the specified point and continuing until non-background-colour pixels are found. FILL is followed by the X and Y coordinates of the starting point; the optional qualifier BY indicates that the coordinates are relative (offsets from the current graphics cursor position) rather than absolute.

The operation may appear to fail if the graphics background colour is not available as a 'solid' colour, with the current display settings, and has therefore been approximated by 'dithering'.

FILL x,y is equivalent to PLOT 133,x,y
FILL BY x,y is equivalent to PLOT 129,x,y

FILL is also used with the CIRCLE, ELLIPSE and RECTANGLE statements to indicate that a filled (solid) shape should be drawn, or that a block move rather than a block copy is required.

FILL 400,400

Syntax

FILL <numeric>,<numeric>
FILL BY <numeric>,<numeric>
CIRCLE FILL <numeric>,<numeric>,<numeric>
ELLIPSE FILL <numeric>,<numeric>,<numeric>,<numeric>
RECTANGLE FILL <numeric>,<numeric>,<numeric>,<numeric> [TO <numeric>,<numeric>]

Associated Keywords

BY, CIRCLE, ELLIPSE, GCOL, RECTANGLE
FN

A keyword used to identify a user-defined function. FN is followed immediately by the function name; the first character of the function name may be an underline (or a digit).

A function may take any number of parameters of any type (or none), and may return a string or numeric result. It does not have to be defined before it is used.

A function definition is terminated by '=' used at the beginning of a statement.

The first two examples below are single line function definitions and the third is a multi-line definition:

```
DEF FN_mean(Q1,Q2,Q3,Q4)=(Q1+Q2+Q3+Q4)/4
DEF FN_fact(N):IF N<2 THEN =1 ELSE =N*FN_fact(N-1)
DEF FN_reverse(A$)
   LOCAL B$,Z%
   FOR Z%=1 TO LEN(A$)
      B$=MID$(A$,Z%,1)+B$
   NEXT
   =B$
```

Functions are re-entrant and the parameters (arguments) are normally passed by value. The keyword RETURN can be used in the function definition to specify that a parameter should instead be passed by reference. Arrays and structures are always passed by reference.

The following function sets the print control variable to the parameter passed and returns a null string. It may be used in a PRINT command to change the print control variable (@%) within a print list.

```
DEF FN_pformat(N):@%=N:=""
```

Functions have to return a result, but the value returned by this function is a null string. Consequently, its only effect is to change the print control variable. Thus the PRINT statement

```
PRINT FN_pformat(&90A) X FN_pformat(&2020A) Y
```
will print X in G9z10 format and Y in F2z10 format. See the keyword PRINT for print format details.

Single-line functions can be placed anywhere within your program. Multi-line functions must be placed where they will not be executed 'out of sequence', this usually means at the end of the program after the END statement. See the Procedures and functions sub-section for more information.

FN may alternatively be followed by a numeric value contained in parentheses. This causes the function pointed to by the specified value to be called (an indirect call):

```
fptr% = ^FN_mean()
mean = FN(fptr%)(a, b, c, d)
```

See the Indirect procedure and function calls sub-section for more information.

**Syntax**

```
<n-var>|<s-var>=FN<name>[(<exp>{,<exp>})]
<n-var>|<s-var>=FN(<numeric>)[(<exp>{,<exp>})]
DEF FN<name>[(<n-var>|<s-var>{,<n-var>|<s-var>})]
```

**Associated Keywords**

DEF , ENDPROC , LOCAL , PRIVATE , PROC , RETURN
A statement which initialises a FOR...NEXT loop. The loop is always executed at least once.

```
FOR temperature%=0 TO 9
    FOR A(2,3,1)=9 TO 1 STEP -0.3
```

The FOR...NEXT loop is a way of repeating a section of program a set number of times. For example, the two program segments below perform identically, but the second is easier to understand.

```
10 start=4: end=20: step=2
20 counter=start
30 PRINT counter," ",counter^2
40 counter=counter+step
50 IF counter<=end THEN 30
60 ... 
10 start=4: end=20: step=2
20 FOR counter=start TO end STEP step
30    PRINT counter," ",counter^2
40 NEXT
50 ... 
```

You must not exit a FOR...NEXT loop with a GOTO (see the sub-section on Program Flow Control for more details). You can force a premature end to the loop with the EXIT statement:

```
FOR I=1 TO 20
    X=A^I
    IF X>1000 THEN EXIT FOR
    PRINT I,X
NEXT
```

It is not necessary to declare the loop variable as an integer type in order to take advantage of fast integer arithmetic. If it is an integer, then fast integer arithmetic is used automatically.

Any numeric assignable item may be used as the control variable. In particular, a byte variable (?X) may act as the control variable and only one byte of memory will
be used. See the Indirection sub-section for details of the indirection operators.

```
FOR ?X=0 TO 16: PRINT ~?X: NEXT
FOR !X=0 TO 16 STEP 4: PRINT ~!X: NEXT
```

Because a single stack is used, you cannot use a FOR...NEXT loop to set array elements to LOCAL in a procedure or function.

**Syntax**

```
FOR <n-var>=<numeric> TO <numeric> [STEP <numeric>]
```

**Associated Keywords**

```
TO , STEP , NEXT , EXIT
```
A statement which sets the graphics foreground or background logical colour to be used in all subsequent graphics operations.

```
GCOL colour
GCOL mode,colour
```

GCOL sets the graphics foreground or background colour, and optionally specifies how the colour is to be plotted on the screen. It can be plotted directly, ANDed, ORed or Exclusive-ORed with the colour already there, or the existing colour can be inverted.

When GCOL is followed by one value the graphics foreground or background colour is set, and the plotting mode is set to 'plot'. If the value is less than 128, the graphics foreground colour is set to that value. If the number is 128 or greater, the graphics background colour is set to value-128. There are up to 16 logical colours (numbered 0 to 15), but the number available varies with the mode; see the section on graphics colours for details.

When GCOL is followed by two values the first determines the plotting mode and the second the graphics foreground or background colour. The plotting modes are as follows:

- **mode=0**: Plot the colour specified.
- **mode=1**: OR the specified colour with the colour that is already there.
- **mode=2**: AND the specified colour with the colour that is already there.
- **mode=3**: Exclusive-OR the specified colour with the colour that is already there.
- **mode=4**: Invert the colour that is already there.

```
GCOL 1 : REM sets the graphics foreground colour to 1
GCOL 130 : REM sets the graphics background colour to 2
GCOL 3,4 : REM sets the mode to EOR and the colour to 4
```
See the Compatibility limitations section for important notes on the use of these plotting modes.

GCOL n is equivalent to VDU 18,0,n
GCOL m,n is equivalent to VDU 18,m,n.

Generally, you will not notice a change in the graphics background colour until you clear the graphics screen with a CLG command.

**Syntax**

GCOL [<numeric>,]<numeric>

**Associated Keywords**

CLS, CLG, MODE, COLOUR, PLOT
GET/GET$

A function and compatible string function that read the next character from the keyboard buffer (they wait for the character), or read a character from the screen. GET$ is also used to read a string from a file.

**Reading the keyboard**

N=GET
N$=GET$

GET and GET$ wait for a 'key' (character) to be present in the keyboard buffer and then return the ASCII value of that key or a string (length=1) containing the character of the key. If there are any characters in the keyboard buffer when a GET or GET$ is issued, then the function will return immediately. See the keyword INKEY for a way of emptying the keyboard buffer before issuing a GET.

GET and GET$ do not echo the pressed key to the screen. If you want to display the character for the pressed key, you must PRINT or VDU it.

If the function and cursor keys have been redefined using the *KEY command, pressing them will load the keyboard buffer with their currently defined character string. Since GET and GET$ remove characters from the keyboard buffer, one character will be returned every time a GET is issued. A single GET will return the first character and leave the rest in the keyboard buffer.

You can use GET and GET$ whenever you want your program to wait for a reply before continuing. For example, you may wish to display several screens of instructions and allow the user to decide when he has read each screen.

```
REM First screen of instructions
CLS
PRINT .......
PRINT .......
PRINT "Press any key to continue ";
temp=GET
REM Second screen of instructions
CLS
PRINT ....... etc
```
Note that using GET prevents any events, for example ON CLOSE, from being processed until you press a key. For that reason it is often better to use INKEY instead:

```plaintext
PRINT "Press any key to continue ";
REPEAT temp=INKEY(1):UNTIL temp>0
```

The default values returned by the function and cursor control keys are listed below:

<table>
<thead>
<tr>
<th>Code</th>
<th>Key Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>127</td>
<td>Ctrl/backspace</td>
</tr>
<tr>
<td>128</td>
<td>Ctrl/←</td>
</tr>
<tr>
<td>129</td>
<td>Ctrl/→</td>
</tr>
<tr>
<td>130</td>
<td>Home</td>
</tr>
<tr>
<td>131</td>
<td>End</td>
</tr>
<tr>
<td>132</td>
<td>PgUp</td>
</tr>
<tr>
<td>133</td>
<td>PgDn</td>
</tr>
<tr>
<td>134</td>
<td>Ins</td>
</tr>
<tr>
<td>135</td>
<td>Del</td>
</tr>
<tr>
<td>136</td>
<td>←</td>
</tr>
<tr>
<td>137</td>
<td>→</td>
</tr>
<tr>
<td>138</td>
<td>↓</td>
</tr>
<tr>
<td>139</td>
<td>↑</td>
</tr>
<tr>
<td>140</td>
<td>Mouse wheel down</td>
</tr>
<tr>
<td>141</td>
<td>Mouse wheel up</td>
</tr>
<tr>
<td>145</td>
<td>F1</td>
</tr>
<tr>
<td>146</td>
<td>F2</td>
</tr>
<tr>
<td>147</td>
<td>F3</td>
</tr>
<tr>
<td>148</td>
<td>F4</td>
</tr>
<tr>
<td>149</td>
<td>F5</td>
</tr>
<tr>
<td>150</td>
<td>F6</td>
</tr>
<tr>
<td>151</td>
<td>F7</td>
</tr>
<tr>
<td>152</td>
<td>F8</td>
</tr>
<tr>
<td>153</td>
<td>F9</td>
</tr>
<tr>
<td>154</td>
<td>F10</td>
</tr>
<tr>
<td>155</td>
<td>F11*</td>
</tr>
<tr>
<td>156</td>
<td>F12*</td>
</tr>
<tr>
<td>157</td>
<td>F13*</td>
</tr>
<tr>
<td>158</td>
<td>F14*</td>
</tr>
<tr>
<td>159</td>
<td>F15*</td>
</tr>
</tbody>
</table>

* These codes are also generated from the following key combinations:
+note that Alt/F4 normally closes the window so does not actually return the code listed. You can make Alt/F4 return code 196 by using the ON CLOSE statement, but beware that this will also prevent you closing the window by clicking on the close button.

The mouse wheel codes (140 and 141) require Windows™ 98 or later.

The ASCII code for DELete (127) can be generated by holding down the Ctrl key and pressing backspace. If you would prefer the DEL key to produce this code instead of its default value (135) you can achieve that using *KEY as follows:

    *KEY 23 |?

(that is, the vertical line or pipe character followed by a question mark).

**Reading the screen**

```
N=GET(X,Y)
N$=GET$(X,Y)
```

GET(X,Y) returns the character at text coordinates X,Y as an ASCII code. GET$(X,Y) returns the character at text coordinates X,Y as a string. The coordinates are with respect to the current text viewport. If the coordinates are outside the viewport GET(X,Y) returns -1 and GET$(X,Y) returns an empty string.

In UTF-8 mode GET$(X,Y) returns a string of length 1, 2 or 3 bytes, depending on the position of the character within the Basic Multilingual Plane.

Note that GET(X,Y) and GET$(X,Y) do not work reliably if the text was written using a proportional-spaced font, or in a mixture of different fonts, or if the font has changed since the text was written.
Reading from a file

A$=GET$#file_channel

GET$ can also be used to read a string from a file. GET$ differs from INPUT# and READ# in that it reads a string terminated by carriage-return (&0D), line-feed (&0A) or NUL (&00) - you cannot tell what the terminator actually was. It also differs in that if a terminator character is not found within 65535 bytes, it returns a string of length 65535 containing the data read up to this point (BBC BASIC for Windows v5.95a only). By contrast, INPUT# and READ# return an empty string in these circumstances, and the data read from the file is discarded. GET$# is therefore more useful if the format of the file is uncertain.

GET$ may optionally be used with the qualifier BY (to specify how many bytes to read from the file) or TO (to specify the terminator character). These override the normal CR, LF or NUL terminators:

dest$=GET$#file_channel BY bytecount%
dest$=GET$#file_channel TO termchar%

The terminator character is specified as its ASCII code, for example to specify a comma terminator use TO 44, TO &2C or TO ASC(",") . If you add &100 to the value (e.g. TO &12C ) the specified terminator will be recognised in addition to (rather than instead of) the normal CR and LF terminators.

If you add &8000 to the value (e.g. TO &802C ) CR (carriage return) will be recognised as a terminator in addition to the specified character (in this example a comma). However LF will not be recognised as a terminator.

Syntax

<n-var> = GET
<s-var> = GET$
<n-var> = GET(<numeric>,<numeric>)
<s-var> = GET$((numeric),<numeric>)
<s-var> = GET$#<numeric>
<s-var> = GET$#<numeric> BY <numeric>
<s-var> = GET$#<numeric> TO <numeric>
Associated Keywords

INKEY, INKEY$, INPUT#, READ#, OPENIN, OPENOUT, OPENUP
**GOSUB**

A statement which calls a section of a program (a subroutine) at a specified line number or label. One subroutine may call another subroutine (or itself).

```
GOSUB 400
GOSUB (4*answer+6)
GOSUB (mysub)
```

The only limit placed on the depth of nesting is the room available for the stack.

You may calculate the line number. However, if you do, the program should not be renumbered. A calculated value must be placed in brackets.

Very often you need to use the same group of program instructions at several different places within your program. It is tedious and wasteful to repeat this group of instructions every time you wish to use them. You can separate this group of instructions into a small sub-program. This sub-program is called a subroutine. The subroutine can be 'called' by the main program every time it is needed by using the GOSUB statement. At the end of the subroutine, the RETURN statement causes the program to return to the statement after the GOSUB statement.

**Syntax**

```
GOSUB <l-num>
GOSUB (<numeric>)
```

**Associated Keywords**

```
RETURN , ON , PROC
```
GOTO

A statement which makes BBC BASIC jump directly to a specified line number or label rather than continuing with the next statement in the program.

```
GOTO 100
GOTO (X*10)
GOTO (mydest)
```

You should never GOTO a line which is outside the current FOR ...NEXT, REPEAT ...UNTIL or WHILE ...ENDWHILE loop.

If a calculated value is used, the program should not be renumbered. A calculated value must be placed in brackets.

The use of GOTO is deprecated. Uninhibited use will make your programs almost impossible to understand (and hence, debug). It is always possible to find an alternative to using GOTO, and invariably your programs will be improved as a result.

Syntax

```
GOTO <l-num>
GOTO (<numeric>)
```

Associated Keywords

```
GOSUB , ON
```
HIMEM

A pseudo-variable which contains the address of the first byte above BBC BASIC for Windows' program, data and stack memory. The user's program and data grow upwards from LOMEM and the stack grows downwards from HIMEM. If the two meet, an untrappable 'No room' error results. See the Format of data in memory section for more details.

By default, HIMEM is set at slightly less than 1 Mbyte above LOMEM (2 Mbytes in BBC BASIC for Windows version 6.00a or later); this is determined by the initial user memory setting. If this is insufficient for your program and data, you can raise HIMEM (subject to there being sufficient memory in your computer). This is an important difference from other versions of BBC BASIC, in which you must never raise HIMEM above its initial value. Note that INSTALL ed libraries are loaded immediately above HIMEM, so if you do raise HIMEM above the value it had when the first library was loaded, all libraries will be removed.

If you need to raise HIMEM don't be tempted to set it to an absolute value (memory addresses are allocated by Windows™ and vary from session to session). Instead always set it relative to the value of PAGE, LOMEM or TOP. For best performance, set HIMEM to an exact multiple of four; you can achieve that by ANDing it with -4 (&FFFFFFFC).

If you need to reserve an area of memory which will not be affected by CHAIN or CLEAR (for example to pass data between programs) you can lower HIMEM from its initial value. The area of memory between the new value of HIMEM and the old value of HIMEM (-1) will not be used by BBC BASIC for Windows (except when the INSTALL statement loads the first library).

HIMEM must not be changed within a subroutine, procedure or function; in a FOR ... NEXT, REPEAT ... UNTIL, or WHILE ... ENDWHILE loop nor within a multi-line IF ... ENDIF clause. If you want to change HIMEM, it is better to do so early in your program.

```
HIMEM = PAGE+10000000
HIMEM = (LOMEM+10000000) AND -4
HIMEM = HIMEM-40
```
Syntax

HIMEM=<numeric>
<n-var>=HIMEM

Associated Keywords

LOMEM, PAGE, TOP
**IF**

A statement which tests a condition and controls the subsequent flow of the program depending on the result. It is part of the IF...THEN ....ELSE structure. IF statements may be **single line** or **multi-line**.

**Single-line IF statement**

```
IF length=5 THEN 110
IF A<C OR A>D GOTO 110
IF A>C AND C>=D THEN GOTO 110 ELSE PRINT "BBC"
IF A>Q PRINT"IT IS GREATER":A=1:GOTO 120
```

The IF statement is the primary decision making statement. The testable condition (A<C, etc) is evaluated, and converted to an integer if necessary. If the value is **non-zero** the rest of the line (up to the ELSE clause if there is one) is executed. If the value is **zero**, the rest of the line is ignored, unless there is an ELSE clause in which case execution continues after the ELSE.

```
IF age<21 THEN PRINT "Under 21"
flag=age<21
IF flag THEN PRINT "Under 21"
```

The above examples will print "Under 21" if the value of 'age' is less than 21, and do nothing otherwise.

```
IF age<21 THEN PRINT "Under 21" ELSE PRINT "21 or over"
flag=age<21
IF flag THEN PRINT "Under 21" ELSE PRINT "21 or over"
```

The above examples will print "Under 21" if the value of 'age' is less than 21, and "21 or over" otherwise.

The keyword THEN is optional in most examples of the single-line IF statement. The exceptions are when THEN is followed immediately by a destination line number, by a pseudo-variable (e.g. TIME ) or by an end-of-function (=). However, THEN is mandatory for the multi-line IF statement.
**Multi-line IF statement**

```plaintext
IF length=5 THEN
    PRINT "length is 5"
ELSE
    PRINT "length is not 5"
ENDIF
```

A multi-line IF clause is begun by an IF statement which has the keyword THEN as the very last thing on the line (it cannot even be followed by a REM ark). It is ended by an ENDIF statement, which must be the first thing on the line. If the integer numeric expression after IF is **non-zero**, the statements up to the ENDIF (or ELSE clause if present) are executed. If the expression evaluates to **zero** the statements after ELSE are executed, or if there is no ELSE clause execution continues after the ENDIF statement.

There may be any number of lines between the IF ... THEN statement and the ENDIF statement. Multi-line IF ... ENDIF clauses may be nested.

You can cascade multiple IF ... ENDIF clauses to test a number of different alternatives:

```plaintext
IF x=3 THEN
    PRINT "x is 3"
ELSE IF x=4 THEN
    PRINT "x is 4"
ELSE IF x=5 THEN
    PRINT "x is 5"
ELSE
    PRINT "x is not 3 or 4 or 5"
ENDIF
ENDIF
```

but in this situation a CASE statement would be a better choice.

**Syntax**

```plaintext
IF <t-cond> THEN <stmt>{:<stmt>} [ELSE <stmt>{:<stmt>}]  
IF <t-cond> GOTO <l-num> [ELSE <l-num>]  
IF <t-cond> THEN <l-num> [ELSE <l-num>]  
IF <t-cond> THEN
```
Associated Keywords

THEN, ELSE, ENDIF
INKEY/INKEY$

A function and compatible string function which does a GET/GET$ , but waits for a specified maximum time rather than indefinitely. INKEY can also be used to test the current state of a specific key (pressed or not pressed).

INKEY or INKEY$ is followed by a numeric argument. If this is zero or positive it indicates the maximum time to wait for a keypress (in units of 10ms). If it is negative it indicates which key should be tested.

INKEY positive or zero argument

INKEY/INKEY$ waits for the specified maximum number of clock ticks of 10ms each. If no key is pressed in the time limit, INKEY will return -1 and INKEY$ will return a null (empty) string. If there is one or more characters waiting in the keyboard buffer, INKEY/INKEY$ will behave exactly the same as GET/GET$. If the specified time limit is zero, INKEY/INKEY$ will return immediately.

The INKEY function returns the ASCII value of the key pressed. The INKEY$ function returns a string (length=1) containing the character corresponding to the key pressed. The values returned by the function and other special keys are tabulated under GET/GET$.

```plaintext
key=INKEY(num)
N=INKEY(0)
N$=INKEY$(100)
```

If the function and cursor keys have been redefined using the *KEY command, pressing them will load the keyboard buffer with their currently defined character string. Since INKEY and INKEY$ remove characters from the keyboard buffer, one character will be returned every time an INKEY is issued. A single INKEY will return the first character and leave the rest in the keyboard buffer.

INKEY can be used to 'empty' the keyboard buffer. You can do this with the following program line:

```plaintext
REPEAT UNTIL INKEY(0)=-1
```
**INKEY negative argument**

If INKEY is followed by a negative argument, the state of a particular key is tested. If the key is currently pressed INKEY will return the value -1 (TRUE ); if the key is not pressed INKEY will return the value 0 (FALSE ).

The numeric values corresponding to the different keys are as follows (note that these values are as compatible as possible with the BBC Micro and the Acorn Archimedes, allowing for the differences between their keyboards and the PC's keyboard).

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>-66</td>
</tr>
<tr>
<td>B</td>
<td>-101</td>
</tr>
<tr>
<td>C</td>
<td>-83</td>
</tr>
<tr>
<td>D</td>
<td>-51</td>
</tr>
<tr>
<td>E</td>
<td>-35</td>
</tr>
<tr>
<td>F</td>
<td>-68</td>
</tr>
<tr>
<td>G</td>
<td>-84</td>
</tr>
<tr>
<td>H</td>
<td>-85</td>
</tr>
<tr>
<td>I</td>
<td>-38</td>
</tr>
<tr>
<td>J</td>
<td>-70</td>
</tr>
<tr>
<td>K</td>
<td>-71</td>
</tr>
<tr>
<td>L</td>
<td>-87</td>
</tr>
<tr>
<td>M</td>
<td>-102</td>
</tr>
<tr>
<td>N</td>
<td>-86</td>
</tr>
<tr>
<td>O</td>
<td>-55</td>
</tr>
<tr>
<td>P</td>
<td>-56</td>
</tr>
<tr>
<td>Q</td>
<td>-17</td>
</tr>
<tr>
<td>R</td>
<td>-52</td>
</tr>
<tr>
<td>S</td>
<td>-82</td>
</tr>
<tr>
<td>T</td>
<td>-36</td>
</tr>
<tr>
<td>U</td>
<td>-54</td>
</tr>
<tr>
<td>V</td>
<td>-100</td>
</tr>
<tr>
<td>W</td>
<td>-34</td>
</tr>
<tr>
<td>X</td>
<td>-67</td>
</tr>
<tr>
<td>Y</td>
<td>-69</td>
</tr>
<tr>
<td>Z</td>
<td>-98</td>
</tr>
<tr>
<td>[</td>
<td>-49</td>
</tr>
<tr>
<td>]</td>
<td>-50</td>
</tr>
<tr>
<td>;</td>
<td>-18</td>
</tr>
<tr>
<td>'</td>
<td>-19</td>
</tr>
<tr>
<td>#</td>
<td>-20</td>
</tr>
<tr>
<td>,</td>
<td>-53</td>
</tr>
<tr>
<td>.</td>
<td>-37</td>
</tr>
<tr>
<td>/</td>
<td>-22</td>
</tr>
<tr>
<td>\</td>
<td>-39</td>
</tr>
<tr>
<td>Tab</td>
<td>-40</td>
</tr>
<tr>
<td>Bksp</td>
<td>-24</td>
</tr>
<tr>
<td>Enter</td>
<td>-94</td>
</tr>
<tr>
<td>Space</td>
<td>-46</td>
</tr>
<tr>
<td>Space</td>
<td>-99</td>
</tr>
<tr>
<td>Esc</td>
<td>-113</td>
</tr>
<tr>
<td>Ins</td>
<td>-62</td>
</tr>
<tr>
<td>Home</td>
<td>-63</td>
</tr>
<tr>
<td>PgUp</td>
<td>-64</td>
</tr>
<tr>
<td>Del</td>
<td>-90</td>
</tr>
<tr>
<td>End</td>
<td>-106</td>
</tr>
<tr>
<td>PgDn</td>
<td>-79</td>
</tr>
<tr>
<td>Shift</td>
<td>-1</td>
</tr>
<tr>
<td>Ctrl</td>
<td>-2</td>
</tr>
<tr>
<td>Alt</td>
<td>-3</td>
</tr>
<tr>
<td>Caps</td>
<td>-65</td>
</tr>
<tr>
<td>Print</td>
<td>-33</td>
</tr>
</tbody>
</table>

**Numeric keypad:**

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>.</td>
</tr>
</tbody>
</table>

Enter -61 Num -78
Note that the above table corresponds to the standard UK keyboard layout. The US keyboard differs slightly:

```
  `   -80   '   -91
```

The mouse buttons can be tested with -10 (left) -11 (middle) and -12 (right)

```
IF INKEY(-99) PRINT "Space bar pressed"
```

**Important note:** INKEY with a negative argument works whether or not your program has the *input focus*. This can occasionally be useful, but often you will prefer to detect a key being pressed only when the user is working with your program. In that case you can use the Windows™ API to check for input focus.

In *BBC BASIC for Windows* INKEY(-256) returns the value ASC("W"), i.e. 87.

**Syntax**

```
<n-var>=INKEY(<numeric>)
<s-var>=INKEY$(<numeric>)
```

**Associated Keywords**

```
GET, GET$, MOUSE, WAIT
```
A statement to input values from the keyboard.

```
INPUT A, B, C, D$, "WHO ARE YOU", W$, "NAME" R$
```

If the items are not immediately preceded by a printable prompt string (even if null) then a '? ' will be printed as a prompt. If the variable is not separated from the prompt string by a comma, the '?' is not printed. In other words: no comma - no question mark.

Items A, B, C, DS in the above example can have their answers returned on one to four lines, separate items being separated by commas. Extra items will be ignored.

Then WHO ARE YOU? is printed (the question mark comes from the comma) and W$ is input, then NAME is printed and R$ is input (no comma - no '? ').

When the <Enter> key is pressed to complete an entry, a new-line is generated. BBC BASIC has no facility for suppressing this new-line, but the TAB function can be used to reposition the cursor. For example,

```
INPUT TAB(0, 5) "Name ? " N$, TAB(20, 5) "Age ? " A
```

will position the cursor at column 0 of line 5 and print the prompt Name ?. After the name has been entered the cursor will be positioned at column 20 on the same line and Age ? will be printed. When the age has been entered the cursor will move to the next line.

The statement

```
INPUT A
```

is exactly equivalent to

```
INPUT A$: A=VAL(A$)
```
Leading spaces will be removed from the input line, but not trailing spaces. If the input string is not completely numeric, it will make the best it can of what it is given. If the first character is not numeric, 0 will be returned. Neither of these two cases will produce an error indication. Consequently, your program will not abort back to the command mode if a bad number is input. You may use the EVAL function to convert a string input to a numeric and report an error if the string is not a proper number or you can include your own validation checks.

```
INPUT A$
A=EVAL(A$)
```

Strings in quoted form are taken as they are, with a possible error occurring for a missing closing quote.

A semicolon following a prompt string is an acceptable alternative to a comma.

See the input editing section for details of the editing facilities available when entering data in response to an INPUT statement.

**Syntax**

```
INPUT \{TAB(X[,Y])\}[SPC(<numeric>)][')]<s-const>[,|;]
<n-var>|<s-var>{,<n-var>|<s-var>}
```

**Associated Keywords**

```
INPUT LINE , INPUT# , GET , INKEY
```
INPUT LINE

A statement of identical syntax to INPUT which uses a new line for each item to be input. The item input is taken as is, including commas, quotes and leading spaces.

INPUT LINE A$

Syntax

INPUT LINE[TAB(X[,Y])][SPC(<numeric>)][.SOCK(<s-const>[,|;])]
\hspace{1em} <s-var>{,<s-var>}

Associated Keywords

INPUT
INPUT#

A statement which reads data in internal format from a file and puts them in the specified variable(s). INPUT# is used with a file or device opened with OPENIN, OPENUP or OPENOUT.

\[
\text{INPUT } #E,A,B,C,D$,E$,F$
\]

\[
\text{INPUT } #\text{chnl},aux$
\]

It is possible to read past the end-of-file without an error being reported. You should usually include some form of check for the end of the file.

READ# is synonymous with INPUT#. When INPUT# and READ# are used to read strings, they look for a carriage-return terminator (&0D), which is what is written with PRINT#. If the string was written by another program and is terminated with a line-feed character (&0A), then use GET$# instead.

The format of data files differs according to the *FLOAT mode in effect at the time (in *FLOAT 40 mode numeric values are written as 5 bytes, in *FLOAT 64 mode they are written as 8 bytes and - in BBC BASIC for Windows version 6.00a or later only - in *FLOAT 80 mode they are written as 10 bytes). You must ensure that when numeric values are read with INPUT# (or READ#) the same *FLOAT mode is in effect as when the values were written.

The format of BBC BASIC for Windows data files is different from files created by Acorn versions of BBC BASIC. If you want to read from an Acorn format data file you can use the following function, which takes the file channel number as a parameter and returns a numeric or string value as appropriate (this works only in the default *FLOAT 40 mode):

```
DEF FNacorn(F%):LOCAL I%:I%=BGET#F%
IF I%=&40 ?(^I%+3)=BGET#F%:?(^I%+2)=BGET#F%:?(^I%+1)=BGET#F%:?(^I%)=BGET#F%:=I%
IF I%=0 I%=BGET#F%:LOCAL A$:WHILE I% A$=CHR$BGET#F%+A$:I% -= 1:ENDWHILE:=A$
LOCAL N:INPUT #F%,N:IF ?(^N+4)=0 THEN =0 ELSE =N/2
```

See the Disk files section for more details and numerous examples of the use of INPUT#.
Syntax

\[ \text{INPUT } \#<\text{numeric}>,<\text{n-var}>|<\text{s-var}>\{|,<\text{n-var}>|<\text{s-var}> \}\]

Associated Keywords

\[ \text{INPUT, OPENIN, OPENUP, OPENOUT, CLOSE#, PRINT#, READ#, BPUT#, BGET#, GET$#, EXT#, PTR#, EOF#} \]
INSTALL

A statement which allows you to load a library file containing one or more pre-defined functions and procedures. These functions and procedures can be called by name from within your program, but they do not appear in the program listing. If you, or somebody else, has written a number of useful functions which you might want to call from several different programs, this provides a convenient way of packaging and distributing them.

Because the procedures and functions do not form part of your program the size of its file is reduced, and if a bug is found in one of the functions it can be corrected by updating the library file without having to modify each of the programs from which it is called.

You can INSTALL as many library files at the same time as you like, subject to there being sufficient memory. These libraries remain resident until the BASIC program's window is closed (they are not cleared by a CHAIN statement). If two or more libraries contain a procedure or function of the same name, the one used is that in the last library loaded.

The libraries are loaded above HIMEM so if you raise HIMEM above the value it had when the first library was INSTALLed, all the libraries will be removed from memory.

A library file is a standard BBC BASIC internal-format (tokenised) program file except that it should normally contain only procedure and function definitions. Line numbers and labels are not recognised in library files, therefore the GOTO and GOSUB statements cannot be used, and the only variant of the RESTORE statement which is useful is the RESTORE +n form used in conjunction with LOCAL DATA and RESTORE DATA.

You should normally prefix the filename with either @lib$ (for standard libraries) or @dir$ (for libraries specific to your program) so that the file will be loaded from the correct location irrespective of the current directory setting. See the Library routines section for details of the supplied libraries.

```
INSTALL @lib$+"arraylib"
INSTALL @dir$+"mylib"
```
INSTALL checks to see if the library is already loaded, and if so it does nothing. Note that a case-sensitive comparison of the supplied path/filename is performed, so the same library being specified differently will not be detected.

**Syntax**

```
INSTALL <string>
```

**Associated Keywords**

HIMEM, FN, PROC
INSTR

A function which returns the position of a sub-string within a string, optionally starting the search at a specified place in the string. The leftmost character position is 1. If the sub-string is not found, 0 is returned.

The first string is searched for any occurrence of the second string.

There must not be any spaces between INSTR and the opening bracket.

```
X=INSTR(A$,B$)
positional=INSTR(find_in$,look_for$)
Y=INSTR(A$,B$,Z%) :REM START AT POSITION Z%
```

You can use this function for validation purposes. If you wished to test A$ to see if was one of the set 'FRED BERT JIM JOHN', you could use the following:

```
set$="FRED BERT JIM JOHN"
IF INSTR(set$,A$) PROC_valid ELSE PROC_invalid
```

The character used to separate the items in the set must be excluded from the characters possible in A$. One way to do this is to make the separator an unusual character, say CHRS$(127)$.

```
z$=CHR$(127)
set$="FRED"+z$+"BERT"+z$+"JIM"+z$+"JOHN"
```

Syntax

```
<n-var>=INSTR(<string>,<string>[,<numeric>])
```

Associated Keywords

```
LEFT$, MID$ , RIGHT$ , LEN
```
A function truncating a real number to the lower integer.

\[
X=\text{INT}(Y)
\]

\[
\text{INT}(99.8) = 99
\]
\[
\text{INT}(-12) = -12
\]
\[
\text{INT}(-12.1) = -13
\]

This function converts a real number (one with a decimal part) to the nearest integer (whole number) less than or equal to the number supplied. Thus,

\[
\text{INT}(14.56)
\]
gives 14, whereas

\[
\text{INT}(-14.5)
\]
gives -15.

**Note:** INT truncates towards minus infinity but assignment to an integer variable truncates towards zero. Thus:

\[
x\% = \text{INT}(-14.5)
\]
\[
\text{PRINT } x\%
\]
\[
x\% = -14.5
\]
\[
\text{PRINT } x\%
\]

prints -15 and -14 respectively.

**Syntax**

\[
<n\text{-var}>=\text{INT}<\text{numeric}>
\]

**Associated Keywords**
**LEFT$**

A string function which returns the left 'num' characters of the string. If there are insufficient characters in the source string, all the characters are returned. LEFT$ may also be used to the left of an equals sign to change the leftmost part of a string whilst leaving the rest alone.

There must not be any spaces between LEFT$ and the opening bracket.

```
newstring$=LEFT$(A$,num)
A$=LEFT$(A$,2)
A$=LEFT$(RIGHT$(A$,3),2)
LEFT$(A$,3) = B$
```

For example,

```
name$="BBC BASIC for Windows"
FOR i=3 TO 13
   PRINT LEFT$(name$,i)
NEXT
END
```

would print

```
BBC
BBC
BBC B
BBC BA
BBC BAS
BBC BASI
BBC BASIC
BBC BASIC
BBC BASIC f
BBC BASIC fo
BBC BASIC for
BBC BASIC for
```

LEFT$(A$,0) returns an empty string; LEFT$(A$,−1) returns the original string unmodified.

When using LEFT$ on the left of an equals sign, and the expression to the right of the equals sign evaluates to a string with fewer characters than the specified sub-
string length, only that number of characters is changed. For example:

```
A$ = "BBC BASIC"
LEFT$(A$, 3) = "ZZ"
```

will set A$ equal to "ZZC BASIC". Although the sub-string length is set to three, only two characters are actually modified since that is the length of the string "ZZ".

LEFT$(A$) is shorthand for LEFT$(A$, LENA$-1)$, in other words it returns or sets all but the last character of A$.

**Syntax**

```
<s-var>=LEFT$(<string>[,<numeric>])
LEFT$(<s-var>[,<numeric>])=<string>
```

**Associated Keywords**

`RIGHT$`, `MID$`, `LEN`, `INSTR`
LEN

A function which returns the length of the argument string.

\[ X = \text{LEN} \left( \text{"fred"} \right) \]
\[ X = \text{LENT} \]
\[ X = \text{LEN} \left( A+\text{B} \right) \]

This function 'counts' the number of characters in a string. For example,

\[ \text{length} = \text{LEN} \left( \text{"BBC BASIC for Windows"} \right) \]

would set 'length' to 21.

LEN is often used with a FOR ....NEXT loop to 'work down' a string doing something with each letter in the string. For example, the following program looks at each character in a string and checks that it is a valid hexadecimal digit.

valid$=\text{"0123456789ABCDEF"}
\text{REPEAT}
\text{INPUT } "\text{Type in a HEX number" } \text{hex$}
\text{flag=TRUE}
\text{FOR } i=1 \text{ TO } \text{LEN(hex$)}\]
\hspace{1cm} \text{IF INSTR(valid$, \text{MID$(hex$,i,1)})=0 flag=FALSE}
\hspace{1cm} \text{NEXT}
\hspace{1cm} \text{IF NOT flag THEN PRINT } "\text{Bad HEX"}
\text{UNTIL flag}

Syntax

\[ <\text{n-var}> = \text{LEN} \left( <\text{string}> \right) \]

Associated Keywords

LEFT$ , MID$ , RIGHT$ , INSTR
LET

A statement indicating an assignment to a string or numeric variable (or an array element or structure member). LET can normally be omitted, and must be omitted in the case of assignment to the pseudo-variables LOMEM, HIMEM, PAGE, PTR#, and TIME.

LET was mandatory in early versions of BASIC. Its use emphasised that when we write

\[ X = X + 4 \]

we don't mean to state that X equals X+4 - it can't be, but rather 'let X become equal to what it was plus 4':

\[ \text{LET } X = X + 4 \]

Most modern versions of BASIC allow you to drop the 'LET' statement. However, if you are writing a program for a novice, the use of LET makes it more understandable.

Syntax

\[ [\text{LET}] \ <\var> = <\exp> \]

Associated Keywords

None
LINE

A statement which draws a straight line between two points in the current graphics foreground colour. LINE is followed by the X and Y coordinates of the ends of the line.

LINE 100,200,300,400

This will draw a straight line from the point 100,200 to the point 300,400.

LINE x1,y1,x2,y2 is equivalent to MOVE x1,y1 : DRAW x2,y2

The keyword LINE is also used in the INPUT LINE statement. LINE INPUT is synonymous with INPUT LINE.

Syntax

```
LINE <numeric>,<numeric>,<numeric>,<numeric>
LINE INPUT [TAB(X[,Y])][SPC(<numeric>)]['][<s-const>[,|;]]
    <s-var>{,<s-var>}
```

Associated Keywords

DRAW , INPUT , MOVE , PLOT
LN

A function giving the natural logarithm of its argument.

\[ X = \text{LN}(Y) \]

This function gives the logarithm to the base 'e' of its argument. The 'natural' number, 'e', is approximately 2.71828183.

Logarithms are 'traditionally' used for multiplication (by adding the logarithms) and division (by subtracting the logarithms). For example,

\[
\begin{align*}
\log_1 &= \text{LN}(2.5) \\
\log_2 &= \text{LN}(2) \\
\log_3 &= \log_1 + \log_2 \\
\text{answer} &= \text{EXP} (\log_3) \\
\text{PRINT} &\text{ answer}
\end{align*}
\]

will calculate 2.5*2 by adding their natural logarithms and print the answer.

Syntax

\[ \text{<n-var>}=\text{LN}<\text{numeric}> \]

Associated Keywords

LOG, EXP
LOCAL

A statement to declare variables for local use inside a function (FN) or procedure (PROC).

LOCAL saves the value(s) of the specified variable(s) on the stack, and initialises the variables to zero (in the case of numeric variables) or null (in the case of string variables). The original values are restored from the stack on exit from the function or procedure (i.e. at '=' or ENDPROC). The variables need not have been previously declared.

An entire array may be made LOCAL, following which it is in an undimensioned state. Before the local array can be used within the function or procedure it must be re-dimensioned using a DIM statement. The new dimensions can be the same as or different from those of the original array. LOCAL arrays are allocated on the stack, and are freed when the function/procedure is exited.

Similarly an entire structure may be made LOCAL, following which it is in an undeclared state. Before the local structure can be used within the function or procedure it must be re-declared using a DIM statement. The new format (number and type of members) can be the same as or different from that of the original structure. LOCAL structures are allocated on the stack, and are freed when the function/procedure is exited.

See also PRIVATE which is similar to LOCAL except that the variables hold their values from one call of the function/procedure to the next, rather than being initialised to zero/null each time.

LOCAL A$,X,Y%,items()

If a function or a procedure is used recursively, the LOCAL variables will be preserved at each level of recursion.

LOCAL DATA

LOCAL DATA saves the current DATA pointer on the stack (but leaves its value unchanged). When used inside a FOR ...NEXT, REPEAT ...UNTIL or WHILE ...ENDWHILE loop, or inside a user-defined function, procedure or subroutine, the
data pointer is automatically restored to its original value on exit. Otherwise it can be restored using the RESTORE DATA statement.

LOCAL DATA
RESTORE +1
DATA "Here", "is", "some", "local", "data"
READ A$, B$, C$, D$, E$
RESTORE DATA

LOCAL is also used in the ON CLOSE LOCAL, ON ERROR LOCAL, ON MOUSE LOCAL, ON MOVE LOCAL, ON SYS LOCAL, ON TIME LOCAL, DIM LOCAL and RESTORE LOCAL statements.

Syntax

LOCAL <n-var>|<s-var>|<array()>){,<n-var>|<s-var>|<array()>}
LOCAL <struct{}>{,<struct{}>}
LOCAL DATA

Associated Keywords

DEF , DIM , ENDPROC , FN , ON , PRIVATE , PROC
LOG

A function giving the base-10 logarithm of its argument.

\[ X = \text{LOG}(Y) \]

This function calculates the common (base 10) logarithm of its argument. Inverse logarithms (anti-logs) can be calculated by raising 10 to the power of the logarithm. For example, if \( x = \text{LOG}(y) \) then \( y = 10^x \).

Logarithms are 'traditionally' used for multiplication (by adding the logarithms) and division (by subtracting the logarithms). For example,

```plaintext
log1 = \text{LOG}(2.5)
log2 = \text{LOG}(2)
log3 = \text{LOG}(2.5 + \text{LOG}(2))
answer = 10^{log3}
PRINT answer
```

Syntax

\[ <n\text{-var}> = \text{LOG}<\text{numeric}> \]

Associated Keywords

LN, EXP
LOMEM

A pseudo-variable which controls where in memory the dynamic data structures are to be placed. The default is TOP, the first free address after the end of the user's BASIC program.

\[
\text{LOMEM = LOMEM+100} \\
\text{PRINT ~LOMEM :REM The ~ makes it print in HEX.}
\]

Normally, dynamic variables are stored in memory immediately after your program (see the section entitled Format of data in memory). You can change the address where **BBC BASIC for Windows** starts to store these variables by changing LOMEM.

USE WITH CARE. Changing LOMEM in the middle of a program causes **BBC BASIC for Windows** to lose track of all the variables you are using.

**Syntax**

\[
\text{LOMEM=<numeric>} \\
\text{<n-var>=LOMEM}
\]

**Associated Keywords**

HIMEM , TOP , PAGE
MID$

A function which returns a part of a string starting at a specified position and with a given length. If there are insufficient characters in the string, or the length is not specified, then all the characters from the specified start position onwards are returned. MID$ may also be used to the left of an equals sign to change part of a string whilst leaving the rest alone.

\[ \text{C$} = \text{MID$(A$,start$_\text{posn}$,num) } \]
\[ \text{C$} = \text{MID$(A$,Z) } \]
\[ \text{MID$(A$,4,2) = B$} \]

You can use this function to select any part of a string. For instance, if

name$="\text{BBC BASIC for Windows}"$

then

part$=\text{MID$(name$,5,3) }$

would set part$ to "BAS". If the third parameter is omitted or there are insufficient characters to the right of the specified position, MID$ returns the right hand part of the string starting at the specified position. Thus,

part$=\text{MID$(name$,15) }$

would set part$ to "Windows".

For example,

\begin{verbatim}
name$="\text{BBC BASIC for Windows}"$
FOR i=5 TO 15
   PRINT MID$(name$,i,11)
NEXT
\end{verbatim}

would print
When using MID$ on the left of an equals sign, and the expression to the right of the equals sign evaluates to a string with fewer characters than the specified sub-string length, only that number of characters is changed. For example:

```
A$ = "BBC BASIC"
MID$(A$, 5, 4) = "ZZ"
```

will set A$ equal to "BBC ZZSIC". Although the sub-string length is set to four, only two characters are actually modified since that is the length of the string "ZZ".

**Syntax**

```
<s-var>=MID$(<string>,<numeric>[,<numeric>])
MID$(<s-var>,<numeric>[,<numeric>])=<string>
```

**Associated Keywords**

LEFT$, RIGHT$, LEN, INSTR
MOD

An operator returning the signed remainder after an integer division. The result is always an integer.

\[ X = A \mod B \]

MOD is defined such that,

\[ A \mod B = A - (A \div B) \times B. \]

If you are doing integer division (DIV) of whole numbers it is often desirable to know the remainder (a 'teach children to divide' program for instance). For example, 23 divided by 3 is 7, remainder 2. Thus,

```
PRINT 23 DIV 3
PRINT 23 MOD 3
```

would print

```
7
2
```

You can use real numbers in these calculations, but they are truncated to their integer part before *BBC BASIC for Windows* calculates the result. Thus,

```
PRINT 23.1 DIV 3.9
PRINT 23.1 MOD 3.9
```

would give exactly the same results as the previous example.

MOD can also be used as a function which operates on an entire numeric array. It returns the *modulus* (the square-root of the sum of the squares of all the elements) in the array. See the array arithmetic section.

**Syntax**
\(<n\text{-}var>\equiv<\text{numeric}> \mod<\text{numeric}>\)
\(<n\text{-}var>\mod<\text{numeric}>\)
\(<n\text{-}var>\equiv\mod(<\text{array()}>)\)

**Associated Keywords**

DIV
A statement to set the screen display mode. The screen (i.e. BASIC's output window) is cleared and all the graphics and text parameters (colours, origin, etc) are reset to their default values.

The following example sets the display to mode 3:

```
MODE 3
```

*BBC BASIC for Windows* starts in a default Windows™ display mode, with black text on a white background. When a MODE statement is executed the window dimensions are changed to those appropriate to that mode, with (initially) white text on a black background.

### Available modes

The available display modes and the resolution of which they are capable are listed below. MODEs 0 to 7 are compatible with MODEs 0 to 7 on the BBC Micro (with some minor differences) and MODEs 8 to 33 are unique to *BBC BASIC for Windows*. No attempt has been made to emulate the screen modes of the Acorn Archimedes, other than MODEs 0 to 7.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Text (chars)</th>
<th>Graphics (pixels)</th>
<th>Graphics units</th>
<th>Logical Colours</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>80x32</td>
<td>640x512</td>
<td>1280x1024</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>40x32</td>
<td>640x512</td>
<td>1280x1024</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>20x32</td>
<td>640x512</td>
<td>1280x1024</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>80x25</td>
<td>640x500</td>
<td>1280x1000</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>40x32</td>
<td>640x512</td>
<td>1280x1024</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>20x32</td>
<td>640x512</td>
<td>1280x1024</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>40x25</td>
<td>640x500</td>
<td>1280x1000</td>
<td>16</td>
</tr>
<tr>
<td>7</td>
<td>40x25</td>
<td>teletext</td>
<td>teletext</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>80x32</td>
<td>640x512</td>
<td>1280x1024</td>
<td>16</td>
</tr>
<tr>
<td>9</td>
<td>40x32</td>
<td>640x512</td>
<td>1280x1024</td>
<td>16</td>
</tr>
</tbody>
</table>
Graphics are available in all modes except MODE 7.

As well as these predefined modes you can also select a 'custom' mode with VDU 23,22.

MODE n is equivalent to VDU 22,n.

MODE can also be used as a function. It returns the current mode number (or -1 if no MODE statement has been executed, or a 'custom' mode is in use).
Syntax

MODE <numeric>
<n-var> = MODE

Associated Keywords

CLS, CLG, VDU
MOUSE

A multi-purpose statement which controls, or returns information about, the mouse. The different variants of the statement are as follows:

MOUSE x,y,b

This returns the current position of the mouse pointer and the status of the mouse buttons; 'x', 'y' and 'b' are the names of numeric variables (which need not have previously been defined). The variables 'x' and 'y' are set to the position of the mouse pointer in BBC BASIC graphics units, and are affected by the current graphics origin. The variable 'b' is set to a value between 0 and 7 depending on which mouse buttons are pressed: the value 1 indicates that the right button is pressed, 2 that the middle button (if any) is pressed and 4 that the left button is pressed. The values may be combined.

Important note: MOUSE works whether or not your program has the input focus. This can occasionally be useful, but often you will prefer to detect a mouse button being pressed only when the user is working with your program. In that case you can use the Windows™ API to check for input focus. Alternatively you may find ON MOUSE better suited to your needs.

The state of the mouse buttons can also be tested using INKEY with the parameters -10 (left), -11 (middle) and -12 (right). The mouse wheel, if any, cannot be tested directly but rotation of the wheel causes characters 140 or 141 to be inserted into the keyboard buffer; these can be read using GET or INKEY in the usual way (this requires Windows™ 98 or later).

MOUSE ON n

This causes the mouse pointer to be displayed, and determines the shape of the pointer depending on the value of 'n'. The standard pointer shapes available are as follows:
MOUSE ON is equivalent to MOUSE ON 0. Note that MOUSE ON 137 may not work on all versions of Windows.

**MOUSE OFF**

This causes the mouse pointer to be hidden (whilst it is within BASIC's output window).

**MOUSE TO x,y**

This moves the mouse pointer to the specified coordinates, in BBC BASIC graphics units. Note that if you move the mouse pointer this will affect all open applications and the desktop, not just BBC BASIC.

If your program is running on Windows 2000™ Professional or Windows XP™
Professional it is possible that you won't be able to move the mouse pointer in this way. This is because the mouse pointer is a shared resource and, on those systems, it is possible for the administrator to withhold privileges which could adversely affect other programs.

**MOUSE RECTANGLE l,b,w,h**

This confines the mouse pointer to remain within the specified bounding rectangle, in BBC BASIC graphics units (left, bottom, width, height in that order). Note that if you confine the mouse pointer this will affect all open applications and the desktop, not just BBC BASIC. If you exit your program with the rectangle still set, you may find it impossible to move the mouse where you need to!

Because the mouse pointer is a shared resource, and confining it could adversely affect other programs, this statement only works if you have the appropriate administrative privileges.

**MOUSE RECTANGLE OFF**

This clears the bounding rectangle so that the mouse pointer is no longer confined.

MOUSE is also used in the ON MOUSE statement.

**Syntax**

```
MOUSE <n-var>,<n-var>,<n-var>
MOUSE ON [<numeric>]
MOUSE OFF
MOUSE TO <n-var>,<n-var>
MOUSE RECTANGLE <n-var>,<n-var>,<n-var>,<n-var>
MOUSE RECTANGLE OFF
```

**Associated Keywords**

INKEY, ON, OFF, ON MOUSE
MOVE

A statement which moves the graphics cursor without drawing a line. MOVE is followed by the X and Y coordinates to which the cursor should be moved; the optional qualifier BY indicates that the coordinates are relative (offsets from the current graphics cursor position) rather than absolute. The coordinates must be in the range -32768 to +32767.

The graphics origin (X=0, Y=0) is normally the bottom left of the 'screen' (BASIC's output window). The origin can be changed using the ORIGIN statement or the VDU 29 command.

MOVE X,Y
MOVE 124,327

MOVE X,Y is equivalent to PLOT 4,X,Y.
MOVE BY X,Y is equivalent to PLOT 0,X,Y.

MOVE is also used in the ON MOVE statement.

Syntax

MOVE <numeric>,<numeric>
MOVE BY <numeric>,<numeric>

Associated Keywords

BY, DRAW, MODE, GCOL, PLOT, ON MOVE
A statement delimiting a FOR ...NEXT loop. NEXT takes an optional control variable.

```
NEXT
NEXT J
```

If the control variable is present then FOR....NEXT loops may be 'popped' automatically in an attempt to match the correct FOR statement (this should not be necessary). If a matching FOR statement cannot be found, a 'Can't match FOR ' error will be reported.

Leaving out the control variable will make the program run quicker, but this is not to be encouraged.

See the keyword FOR for more details about the structure of FOR....NEXT loops.

**Syntax**

```
NEXT [<n-var>{,<n-var>}]  
```

**Associated Keywords**

FOR, TO, STEP
NOT

A function returning the bitwise or logical inversion of its argument. The argument is converted to a 4-byte integer before the operation.

```
A=NOT 3
flag=NOT flag
flag=NOT(A=B)
```

NOT is most commonly used in an IF ....THEN ....ELSE statement to reverse the effect of the test.

```
IF NOT(rate>5 AND TIME<100) THEN ..... 
IF NOT flag THEN ..... 
```

*BBC BASIC for Windows* does not have true boolean variables; it makes do with numeric variables. This can lead to confusion because the testable condition in an IF....THEN....ELSE statement is evaluated numerically and can result in something other than -1 (TRUE ) or 0 (FALSE ).

If you wish to use NOT to reverse the action of an IF statement it is important to ensure that the testable condition does actually evaluate to either 0 (FALSE) or -1 (TRUE).

If the testable condition evaluates to 1, for example, the test will be considered to have succeeded and the THEN part of the IF....THEN....ELSE statement would be carried out. However, using NOT in front of the testable condition would not reverse the action. NOT 1 evaluates to -2, which would also be considered to be success.

**Syntax**

```
<n-var>=NOT<numeric>
```

**Associated Keywords**

AND , EOR , OR
OF

A keyword which is part of the CASE ... ENDCASE clause. OF follows the name of the CASE variable, and must be the last item on the program line (not even followed by a REMark).

```plaintext
CASE die% OF
  WHEN 1,2,3 : bet$ = "lose"
  WHEN 4,5,6 : bet$ = "win"
  OTHERWISE bet$ = "cheat"
ENDCASE
```

Syntax

```plaintext
CASE <var> OF
```

Associated Keywords

```plaintext
CASE , ENDCASE , OTHERWISE , WHEN
```
OFF

A statement which hides the text cursor (which is normally a flashing underscore). The cursor can be switched on again with ON. OFF is equivalent to VDU 23,1,0;0;0;0;.

OFF is also used in the MOUSE OFF, MOUSE RECTANGLE OFF, ON CLOSE OFF, ON ERROR OFF, ON MOUSE OFF, ON MOVE OFF, ON SYS OFF, ON TIME OFF and TRACE OFF statements.

Syntax

OFF

Associated Keywords

MOUSE, ON, TRACE
ON

A statement controlling a multi-way switch. The destinations in the list may be line numbers (constants or calculated), labels or procedures, and the 'unwanted' ones are skipped without calculation. The ON statement is used in conjunction with three other keywords: GOTO, GOSUB and PROC (ON CLOSE, ON ERROR, ON MOUSE, ON MOVE, ON SYS and ON TIME are explained separately).

ON option GOTO 1000,2000,3000,4000
ON action GOSUB label1,label2,label3,label4
ON choice PROC_add,PROC_find(a$),PROC_delete

The ON statement alters the path through your program by transferring control to one of a selection of line numbers, labels or procedures depending on the value of a variable. For example,

ON number GOTO 1000,2000,500,100

would send your program to line 1000 if 'number' was 1, to line 2000 if 'number' was 2, to line 500 if 'number' was 3 and to line 100 if 'number' was 4.

If 'number' was less than 1 or greater than 4 the ON range error would result. You can trap this condition by using the ELSE statement delimiter:

ON action GOTO 100,300,120 ELSE PRINT"Illegal"

If there is no statement after the ELSE, the program will 'drop through' to the following line if an exception occurs. In the following example, the program would drop through to the error handling part of the program if 'B-46' was less than one or more than 3.

ON B-46 GOTO 100,200,(C/200) ELSE
PRINT "Illegal choice - try again"

You can use ON...GOTO, ON...GOSUB, and ON...PROC to execute the appropriate part of your program as the result of a menu selection. The following skeleton
example offers a menu with three choices.

CLS
PRINT "Select the action you wish to take:"
PRINT "1 Open a new data file"
PRINT "2 Add data to the file"
PRINT "3 Close the file and end"
REPEAT
    INPUT TAB(10,20)"What is your choice",choice
UNTIL choice>0 AND choice<4
ON choice PROC_open,PROC_add,PROC_close
.....etc

Enabling the text cursor

ON can also be used on its own to enable the text cursor (caret). ON is equivalent to VDU 23,1,1;0;0; .

Syntax

ON <numeric> GOTO <l-num>{,<l-num>}
    [ELSE <stmt>{:<stmt>}]}
ON <numeric> GOSUB <l-num>{,<l-num>}
    [ELSE <stmt>{:<stmt>}]}
ON <numeric> PROC<name>[((exp){,<exp>})]
    {,PROC<name>[(<exp>{,<exp>})]]
    [ELSE <stmt>{:<stmt>}]}

Associated Keywords

ON CLOSE , ON ERROR , ON MOUSE , ON MOVE , ON SYS , ON TIME ,
ON ERROR LOCAL , GOTO , GOSUB , PROC
ON CLOSE

A statement which allows you to 'trap' the window being closed by the user. Normally, the window can be closed by clicking on the close button in the top right-hand corner or by pressing Alt-F4. However, in some circumstances it may be desirable to prevent the user doing this (for example, you may wish to prompt him to save some data first).

Once ON CLOSE has been executed, any attempt to close the window (other than by QUIT) will cause control to be transferred to the statement following ON CLOSE. The current program pointer will be saved on the stack, so that you can resume execution from where it left off using RETURN. In this respect, attempting to close the window acts as a sort of 'interrupt' to the BASIC program.

For example, the following program segment asks the user to confirm whether he or she really wants to quit:

```
ON CLOSE PROCclose : RETURN
REPEAT
   REM Do something useful here!
UNTIL FALSE ;
DEF PROCclose : LOCAL ans$
REPEAT UNTIL INKEY(0) = -1
INPUT '"Do you really want to exit",ans$
IF LEFT$(ans$,1) = "y" OR LEFT$(ans$,1) = "Y" THEN QUIT
ENDPROC
```

Note that the ON CLOSE 'interrupt' can only take place in between the execution of one BASIC statement and the next. If BBC BASIC for Windows is waiting in an INKEY, INPUT, GET, SOUND or WAIT statement, then the ON CLOSE procedure will not be called until the current statement has completed. If appropriate you can use the replacement routines in the NOWAIT library.

Be careful not to change anything within the ON CLOSE processing which could affect the operation of the rest of the program if the user chooses to continue. For example, PROCclose above ought really to save the current text cursor position and restore it before returning, and to ensure that the prompt and the user's response do not upset the display. See the notes on the use of interrupts for more details.

If you simply wish to disable the close button (and Alt-F4) you can use:
Trapping the close operation can be cancelled using ON CLOSE OFF, although it is possible for a queued event to be processed by a previous ON CLOSE procedure even after the ON CLOSE OFF statement has been executed. Note that returning to immediate mode automatically cancels event trapping.

ON CLOSE LOCAL can be used inside a function or procedure. It has the same effect as ON CLOSE except that trapping set up by a previous ON CLOSE (if any) is saved, and restored on exit from the function or procedure.

Syntax

```plaintext
ON CLOSE [LOCAL] <stmt>{:<stmt>}:RETURN
ON CLOSE [LOCAL] OFF
```

Associated Keywords

```plaintext
LOCAL , ON MOUSE , ON MOVE , ON SYS , ON TIME , RETURN , PROC
```
ON ERROR

A statement controlling error trapping. If an ON ERROR statement has been encountered, *BBC BASIC for Windows* will transfer control to it (without taking any reporting action) when an error is detected. This allows error reporting/recovery to be controlled by the program. However, the program control stack is still cleared when the error is detected and it is not possible to RETURN to the point where the error occurred.

ON ERROR OFF returns the control of error handling to *BBC BASIC for Windows*.

```
ON ERROR PRINT"Suicide":END
ON ERROR GOTO 100
ON ERROR OFF
```

Error handling is explained more fully in the General Information section.

Syntax

```
ON ERROR <stmt>{:<stmt>}
ON ERROR OFF
```

Associated Keywords

ON, ON ERROR LOCAL, GOTO, GOSUB, PROC
ON ERROR LOCAL

A statement controlling error trapping. If an ON ERROR LOCAL statement has been encountered, *BBC BASIC for Windows* will transfer control to it (without taking any reporting action) when an error is detected. This allows error reporting/recovery to be controlled by the program.

Unlike the ON ERROR statement, ON ERROR LOCAL prevents *BBC BASIC for Windows* clearing the program stack. By using this statement, you can trap errors within a FOR ... NEXT, REPEAT ... UNTIL or WHILE ... ENDWHILE, loop or a subroutine, function or procedure without *BBC BASIC for Windows* losing its place within the program structure.

ON ERROR OFF returns the control of error handling to *BBC BASIC for Windows*.

```
ON ERROR LOCAL PRINT"Suicide":END
ON ERROR LOCAL ENDPROC
ON ERROR OFF
```

The following example program will continue after the inevitable 'Division by zero' error.

```
FOR n=-5 TO 5
ok% = TRUE
ON ERROR LOCAL PRINT "Infinity" : ok% = FALSE
IF ok% PRINT "The reciprocal of ";n;" is ";1/n
NEXT n
```

If ON ERROR LOCAL is used within a procedure, function, or loop structure then the previous error-trapping status (which might be an earlier ON ERROR LOCAL) is automatically restored on exit from the structure. However, ON ERROR LOCAL can be used anywhere within a *BBC BASIC for Windows* program, not just in these structures. In this case you must explicitly restore the previous error-trapping status using RESTORE ERROR.

Error handling is explained more fully in the General Information section.

ON ERROR LOCAL OFF temporarily disables error trapping; the default error reporting mechanism will be used until the previous error-trapping status is restored.
Syntax

ON ERROR LOCAL <stmt>{:<stmt>}  
ON ERROR LOCAL OFF  
ON ERROR OFF

Associated Keywords

ON , ON ERROR , GOTO , GOSUB , PROC , restore
ON MOUSE

A statement which allows you to 'trap' the pressing of a mouse button. Although you can 'poll' the state of the mouse buttons using INKEY or MOUSE, it is sometimes more convenient if the pressing of a button causes an event which will 'interrupt' the program.

If the ON MOUSE statement has been executed, pressing any of the three mouse buttons will cause a transfer of control to the statement after ON MOUSE. The current program pointer will be saved on the stack, so that you can resume execution from where it left off using RETURN.

Once the interrupt has occurred, you can test which button(s) are pressed by examining the @wparam% system variable. Its value depends on the button(s) pressed as follows:

<table>
<thead>
<tr>
<th>Button/key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left button</td>
<td>1</td>
</tr>
<tr>
<td>Right button</td>
<td>2</td>
</tr>
<tr>
<td>SHIFT key</td>
<td>4</td>
</tr>
<tr>
<td>CTRL key</td>
<td>8</td>
</tr>
<tr>
<td>Middle button</td>
<td>16</td>
</tr>
</tbody>
</table>

(the values may be combined).

The position of the mouse pointer when the button was pressed can be discovered from the value of @lparam% as follows:

\[
\begin{align*}
\text{xpos} &= \text{@lparam} \text{ AND } &FFFF \\
\text{ypos} &= \text{@lparam} \text{ >>> 16}
\end{align*}
\]

The coordinates are returned in pixels from the top left-hand corner of the window. If you need the coordinates in BBC BASIC graphics units then use the following code:

\[
\begin{align*}
\text{xpos} &= (\text{@lparam \text{ AND } } &FFFF) \times 2 - \text{@vdu.o.x} \\
\text{ypos} &= (\text{@vdu!212-1-(@lparam >> 16)}) \times 2 - \text{@vdu.o.y}
\end{align*}
\]
You should use the values of @wparam% and/or @lparam% in the statement immediately following ON MOUSE (e.g. by making them parameters of a procedure call), otherwise they might have changed as the result of a subsequent interrupt:

```
ON MOUSE PROCmouse(@wparam%,@lparam%) : RETURN
```

Note that the ON MOUSE 'interrupt' can only take place in between the execution of one BASIC statement and the next. If *BBC BASIC for Windows* is waiting in an INKEY, INPUT, GET, SOUND or WAIT statement, then the ON MOUSE procedure will not be called until the current statement has completed. If appropriate you can use the replacement routines in the NOWAIT library.

Be careful not to change anything within the ON MOUSE processing which could affect the operation of the rest of the program. See the notes on the use of interrupts for more details.

ON MOUSE OFF causes mouse-button events to be ignored, although it is possible for one or more queued events to be processed by a previous ON MOUSE procedure even after the ON MOUSE OFF statement has been executed. Note that returning to immediate mode automatically cancels event trapping.

ON MOUSE LOCAL can be used inside a function or procedure. It has the same effect as ON MOUSE except that trapping set up by a previous ON MOUSE (if any) is saved, and restored on exit from the function or procedure.

**Syntax**

```
ON MOUSE [LOCAL] <stmt>{:<stmt>} :RETURN
ON MOUSE [LOCAL] OFF
```

**Associated Keywords**

- ON CLOSE
- ON MOVE
- ON SYS
- ON TIME
- RETURN
- MOUSE
- PROC
ON MOVE

A statement which allows you to 'trap' manual moving or resizing of BASIC's output window. When a MODE statement is executed, the output window is set to a size dependent on the specified MODE. However, there is nothing to stop the user re-sizing the window, for example by dragging the corner to a different position. ON MOVE causes the BASIC program to be alerted to this change.

If the ON MOVE statement has been executed, moving or re-sizing the window will cause a transfer of control to the statement after ON MOVE. The current program pointer will be saved on the stack, so that you can resume execution from where it left off using RETURN.

Once the interrupt has occurred, you can take any action necessary to deal with the change in window size. For example, by issuing a VDU 26 command you can reset BASIC's text and graphics windows to the new size:

```
ON MOVE VDU 26 : RETURN
```

If you need to distinguish between moving or re-sizing the window you can examine the value of the @msg% system variable. This will have the value 3 for a move and 5 for a re-size. ON MOVE also lets you intercept scroll messages if a scroll bar has been created. In this case @msg% will have the value &114 for a horizontal scroll message and &115 for a vertical scroll message.

Following a change in the window size you might want to know what the new size actually is. Once you have checked that @msg% is 5 (signifying a re-size) you can discover the new size (in pixels) as follows:

```
Width% = @lparam% AND &FFFF
Height% = @lparam% >>> 16
```

You should use the values of @msg% and @lparam%, as required, in the statement immediately following ON MOVE (e.g. by making them parameters of a procedure call), otherwise they might have changed as the result of a subsequent interrupt:

```
ON MOVE PROCmove(@msg%,@lparam%) : RETURN
......
```
DEF PROC move(M%, L%)
IF M%=5.Width%=L% AND &FFFF AND L% >> 16
ENDPROC

Note that the ON MOVE 'interrupt' can only take place in between the execution of one BASIC statement and the next. If *BBC BASIC for Windows* is waiting in an INKEY, INPUT, GET, SOUND or WAIT statement, then the ON MOVE procedure will not be called until the current statement has completed. If appropriate you can use the replacement routines in the NOWAIT library.

Be careful not to change anything within the ON MOVE processing which could affect the operation of the rest of the program. See the notes on the use of interrupts for more details.

ON MOVE OFF causes window-resizing or moving events to be ignored, although it is possible for one or more queued events to be processed by a previous ON MOVE procedure even after the ON MOVE OFF statement has been executed. Note that returning to immediate mode automatically cancels event trapping.

ON MOVE LOCAL can be used inside a function or procedure. It has the same effect as ON MOVE except that trapping set up by a previous ON MOVE (if any) is saved, and restored on exit from the function or procedure.

**Syntax**

```
ON MOVE [LOCAL] <stmt>{:<stmt>}:RETURN
ON MOVE [LOCAL] OFF
```

**Associated Keywords**

```
ON CLOSE, ON MOUSE, ON SYS, ON TIME, RETURN, PROC
```
ON SYS

A statement which allows you to react to a menu item being selected. The *BBC BASIC for Windows* output window does not, ordinarily, include any menus but it is possible to add them using Windows™ API calls (see the section Accessing the Windows API).

If the ON SYS statement has been executed, selecting a menu item will cause a transfer of control to the statement after ON SYS. The current program pointer will be saved on the stack, so that you can resume execution from where it left off using RETURN.

When the interrupt occurs, the WPARAM and LPARAM values associated with the menu selection can be recovered from the system variables @wparam% and @lparam%. These should be immediately passed as parameters to a procedure or function call, to ensure that they are copied to conventional variables before another interrupt can occur:

```
ON SYS PROCmenu(@wparam%,@lparam%) : RETURN
```

Note that the ON SYS 'interrupt' can only take place in between the execution of one BASIC statement and the next. If *BBC BASIC for Windows* is waiting in an INKEY, INPUT, GET, SOUND or WAIT statement, then the ON SYS procedure will not be called until the current statement has completed. If appropriate you can use the replacement routines in the NOWAIT library.

Be careful not to change anything within the ON SYS processing which could affect the operation of the rest of the program. See the notes on the use of interrupts for more details.

ON SYS OFF causes menu events to be ignored, although it is possible for one or more queued events to be processed by a previous ON SYS procedure even after the ON SYS OFF statement has been executed. Note that returning to immediate mode automatically cancels event trapping.

ON SYS is also used when you have created a Windows™ dialogue box; it allows you to handle events such as the user clicking on a button or closing the dialogue.

ON SYS LOCAL can be used inside a function or procedure. It has the same effect.
as ON SYS except that trapping set up by a previous ON SYS (if any) is saved, and restored on exit from the function or procedure.

By default ON SYS and ON SYS LOCAL intercept only Windows™ WM_COMMAND messages. It is possible to intercept other messages using the *SYS command.

Syntax

```
ON SYS [LOCAL] <stmt>{:<stmt>}:RETURN
ON SYS [LOCAL] OFF
```

Associated Keywords

```
ON CLOSE , ON MOUSE , ON MOVE , ON TIME , RETURN , PROC
```
ON TIME

A statement which allows you to generate a periodic 'timer interrupt'. If the ON TIME statement has been executed, control will be transferred to the statement after ON TIME approximately four times every second (to change the rate see below). The current program pointer will be saved on the stack, so that you can resume execution from where it left off using RETURN.

ON TIME allows you to perform a 'background task' without the complication of having to 'poll' the timer from (potentially) many different points in your program. If necessary you can have more than one timer, either using the code listed below or by means of the TIMERLIB library.

Note that the ON TIME 'interrupt' can only take place in between the execution of one BASIC statement and the next. If BBC BASIC for Windows is waiting in an INKEY, INPUT, GET, SOUND or WAIT statement, then the ON TIME procedure will not be called until the current statement has completed. If appropriate you can use the replacement routines in the NOWAIT library.

Be careful not to change anything within the ON TIME processing which could affect the operation of the rest of the program. See the notes on the use of interrupts for more details.

ON TIME OFF causes timer interrupt events to be cancelled, although it is possible for one or more queued events to be processed by a previous ON TIME procedure even after the ON TIME OFF statement has been executed. Note that returning to immediate mode automatically cancels event trapping.

ON TIME LOCAL can be used inside a function or procedure. It has the same effect as ON TIME except that trapping set up by a previous ON TIME (if any) is saved, and restored on exit from the function or procedure.

Syntax

```
ON TIME [LOCAL] <stmt>{:<stmt>} RETURN
ON TIME [LOCAL] OFF
```

Associated Keywords
Changing the timer period

You can change the periodicity of the ON TIME interrupt using the *TIMER command:

```
OSCLI "TIMER " + STR$(period%)
```

where `period%` is the required timer period in milliseconds. Don't expect the rate to be very accurate, and note that the shortest period you can use is 10 ms. If you need a faster timer you may be able to utilise the TIMERLIB library.

Note that the timer period determines the flash rate of flashing characters in MODE 7. You are recommended not to change it when using that mode.

Creating additional timers

You can create additional timers, which also result in ON TIME interrupts, as follows:

```
SYS "SetTimer", @hwnd%, timerid%, period%, 0
```

where `timerid%` is a value other than 101 and `period%` is the required timer period in milliseconds. Don't expect the rate to be very accurate, and note that the shortest period you can use is 10 ms. If you need a faster timer you may be able to utilise the TIMERLIB library.

To determine which timer was responsible for the ON TIME interrupt you can check the value of `@wparam%` which will be 101 for the normal timer and whatever ID(s) you specified for the additional timer(s):

```
ON TIME PROCtimer(@wparam%) : RETURN
```

When you have finished with the additional timers you should cancel them as follows:
Notes on the use of interrupts

The statements ON CLOSE, ON MOUSE, ON MOVE, ON SYS and ON TIME all have the effect of interrupting your program when the associated event occurs. Special care needs to be taken if unwanted side-effects are not to occur as a result.

The problem with interrupts is that they occur at unpredictable times, therefore the part of your program being executed at the time is also unpredictable. If this part of your program can be affected by some change made by the interrupt service routine, the result may be a malfunction. Since only a small part of your program may be affected in this way, it will fail only rarely and in an apparently random fashion. Such faults are very hard to trace.

Ideally the interrupt service routine should be arranged to have no effects on the rest of the program. The following techniques can be valuable in achieving this:

- Any 'private' variables used by the routine should be declared as LOCAL or PRIVATE, and where possible any other system parameters modified by the routine should be saved on entry and restored on exit.

- If the routine uses or alters the value of a global variable, but that same variable is declared as LOCAL or PRIVATE (or is used as a formal parameter) in a function or procedure, then a problem will arise if the interrupt occurs during execution of the function or procedure. In that case the local variable will be accessed instead of the global variable. To avoid this, ensure that any global variables used by interrupt routines are never declared as LOCAL or PRIVATE (or used as formal parameters of a procedure or function) anywhere in your program or libraries that it uses. The use of a strict naming convention (e.g. local variables being in lower-case and global variables starting with a capital letter) can be very helpful.

- If the routine outputs text you can save and restore the text cursor position as follows:

```assembly
DEF PROCinterrupt
LOCAL X%, Y%
X%=POS : Y%=VPOS
REM. Output some text here
```
If the routine performs a graphics operation you can save and restore the current 'VDU state' as follows:

```basic
DEF PROC interrupt
LOCAL vdu save{}
DIM vdu save{} = @vdu{}
vdu save{} = @vdu{}
REM. Output some text or graphics here
@vdu{} = vdu save{}
ENDPROC
```

If you don't do this the interrupt may adversely affect text or graphics output from the rest of the program. For example, when plotting a triangle, PLOT 85 takes the previous two points 'visited' as vertices of the triangle, and any interrupt which moves the graphics pointer in between will cause the triangle to be incorrectly plotted.

Beware of changing the current font and/or graphics viewport whilst the VDU state has been saved; special precautions are necessary in such cases.

If there is a danger that your interrupt service routine could be called *re-entrantly*, that is a second or subsequent interrupt occurs before the previous one has been completely processed, you must either write your code to be tolerant of that possibility or add a **semaphore** to cause the re-entrant interrupts to be ignored (if that is acceptable):

```basic
DEF PROC service routine(parameters)
PRIVATE semaphore%
IF semaphore% THEN ENDPROC
semaphore% = TRUE
REM. Do the interrupt processing here
semaphore% = FALSE
ENDPROC
```

The likelihood of re-entrant calls is greatly increased if your program performs any operations which can *stall* interrupts (for example a WAIT statement with a non-zero parameter). To minimise this possibility you can use the replacement functions in the NOWAIT library.
In general you should try to do **as little as possible** in the interrupt service routine, to minimise the chance of side-effects. Setting a flag within the interrupt routine, and performing the necessary action within your main code when the flag is found to be set, is generally preferable to performing the action within the interrupt routine itself. Ideally you should **test** and **reset** the interrupt flag within one non-interruptible unit, which can be done using the SWAP statement:

```plaintext
interrupt_flag% = FALSE
......
ON TIME interrupt_flag% = TRUE : RETURN
......
REM. within polling loop:
temp% = FALSE
SWAP interrupt_flag%,temp%
IF temp% PROC_take_action
```

The SWAP statement both resets the interrupt flag and copies its previous value into **temp%** so it can be tested at leisure. If you need to know not only that the interrupt occurred, but also the associated **@msg%**, **@wparam%** and/or **@lparam%** values, you can extend the principle by using an array rather than a boolean flag:

```plaintext
DIM click%(2), temp%(2)
......
ON SYS click%() = @msg%,@wparam%,@lparam% : RETURN
......
REM. within polling loop:
temp%() = FALSE
SWAP click%(),temp%()
IF temp%(0) PROC_click(temp%(0),temp%(1),temp%(2))
```

The SWAP both zeroes **click%()** and copies the old data into **temp%()**, so there is no danger of the data from two interrupts being mixed up. If you can guarantee to poll sufficiently often, this technique can eliminate many of the troublesome aspects of dealing with interrupts.

The LOCAL forms of event trapping (ON CLOSE LOCAL, ON MOUSE LOCAL, ON MOVE LOCAL, ON SYS LOCAL and ON TIME LOCAL) need special care. Because of the way events are **queued** it is possible that an event has **occurred** but has not yet been **processed** when the ON <event> LOCAL statement is executed. In that case the event may be processed by the previous ON <event> statement, if any. Similarly, a queued event may be processed by the ON <event> LOCAL statement even **after** the function or procedure in which it is contained has been exited. You
should write your interrupt service routines to be tolerant of these possibilities.
OPENIN

A function which opens a file for reading and returns the 'channel number' of the file. This number must be used in subsequent references to the file with BGET#, INPUT#, EXT#, PTR#, EOF#, CLOSE# or GET$#.

If the extension is omitted, .BBC is assumed. To specify a filename with no extension, add a final full-stop. A returned value of zero signifies that the specified file was not found on the disk, or could not be opened for some other reason.

OPENIN may also be used to open a channel from an input device, such as COM1 (the first serial port) or COM2 (the second serial port). See the section Serial I/O for details.

X=OPENIN "jim"
X=OPENIN A$
X=OPENIN (A$)
X=OPENIN ("FILE1")

The example below reads data from disk into an array. If the data file does not exist, an error message is printed and the program ends.

DIM posn(10), name$(10)
fnum=OPENIN(@usr$+"TOPTEN.DAT")
IF fnum=0 THEN PRINT "No TOPTEN data": END
FOR i=1 TO 10
   INPUT#fnum,posn(i),name$(i)
NEXT
CLOSE#fnum

If the supplied filename is an empty string, or begins with a * (wildcard) character, an Open File dialogue box is displayed.

If the supplied filename contains a wildcard (?) or * character anywhere, an Open File dialogue box is displayed.

Syntax

<n-var>=OPENIN(<string>)
Associated Keywords

OPENOUT, OPENUP, CLOSE#, PTR#, PRINT#, INPUT#, READ#, BGET#, BPUT#, EOF#, GET$#
OPENOUT

A function which opens a file for writing and returns the 'channel number' of the file. This number must be used in subsequent references to the file with BPUT# , PRINT# , EXT# , PTR# or CLOSE#. If the specified file does not exist it is created. If the specified file already exists it is truncated to zero length and all the data in the file is lost.

If the extension is omitted, .BBC is assumed. To specify a filename with no extension, add a final full-stop. A returned value of zero indicates that the specified file could not be created.

OPENOUT may also be used to open a channel to an output device, such as COM1 (the first serial port) or COM2 (the second serial port). See the section Serial I/O for details.

```
X=OPENOUT(A$)
X=OPENOUT("DATAFILE")
```

You can also read from a file which has been opened using OPENOUT. This is of little use until you have written some data to it. However, once you have done so, you can move around the file using PTR# and read back previously written data.

Data is not written to the file at the time it is opened. Consequently, it is possible to successfully open a file on a full disk. Under these circumstances, a 'Disk full' error would be reported when you tried to write data to the file for the first time.

The example below writes the contents of two arrays (tables) to a file called 'TOPTEN.DAT'.

```
A=OPENOUT(@usr$+"TOPTEN.DAT")
FOR Z=1 TO 10
  PRINT#A,N(Z),N$(Z)
NEXT
CLOSE#A
END
```

If the supplied filename is an empty string, or begins with a * (wildcard) character, an Open File dialogue box is displayed.
If the supplied filename contains a wildcard (?) or (*) character anywhere, an Open File dialogue box is displayed.

**Syntax**

\[ <n\text{-}var> = \text{OPENOUT}\{<\text{string}>\} \]

**Associated Keywords**

OPENIN, OPENUP, CLOSE#, PTR#, PRINT#, INPUT#, BGET#, BPUT#, EOF#
OPENUP

A function which opens a disk data file for update (reading and writing) and returns the 'channel number' of the file. This number must be used in subsequent references to the file with BGET#, BPUT#, INPUT#, PRINT#, EXT#, PTR#, EOF#, CLOSE# or GET$#.

If the extension is omitted, .BBC is assumed. To specify a filename with no extension, add a final full-stop. A returned value of zero signifies that the specified file was not found on the disk, or could not be opened for some other reason.

```
X=OPENUP "jim"
X=OPENUP A$
X=OPENUP (A$
X=OPENUP ("FILE1")
```

See the random file examples in the Disk files section for examples of the use of OPENUP.

OPENUP is also used to open a communications port (e.g. a serial port) for access. See the Serial I/O section for details.

If the supplied filename is an empty string, or begins with a * (wildcard) character, an Open File dialogue box is displayed.

If the supplied filename contains a wildcard (?) or *) character anywhere, an Open File dialogue box is displayed.

Syntax

```
<n-var>=OPENUP(<string>)
```

Associated Keywords

OPENIN, OPENOUT, CLOSE#, PTR#, PRINT#, INPUT#, READ#, BGET#, BPUT#, EOF#, GET$#
OPT

An assembler pseudo operation controlling the method of assembly (see the Assembler section for more details). OPT is followed by a numeric value with the following meanings:

<table>
<thead>
<tr>
<th>OPT value</th>
<th>Limit check</th>
<th>Code stored at</th>
<th>Errors reported</th>
<th>Listing generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No</td>
<td>P%</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>No</td>
<td>P%</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>No</td>
<td>P%</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>3</td>
<td>No</td>
<td>P%</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>No</td>
<td>O%</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>5</td>
<td>No</td>
<td>O%</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>6</td>
<td>No</td>
<td>O%</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>7</td>
<td>No</td>
<td>O%</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>8</td>
<td>Yes</td>
<td>P%</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>9</td>
<td>Yes</td>
<td>P%</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>10</td>
<td>Yes</td>
<td>P%</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>11</td>
<td>Yes</td>
<td>P%</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>12</td>
<td>Yes</td>
<td>O%</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>13</td>
<td>Yes</td>
<td>O%</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>14</td>
<td>Yes</td>
<td>O%</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>15</td>
<td>Yes</td>
<td>O%</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Syntax

OPT <numeric>

Associated Keywords

None
OR

An operator returning the bitwise or logical OR between two items. The two operands are internally converted to 4 byte integers before the OR operation. For each of the 32 bits OR uses the following truth-table:

<table>
<thead>
<tr>
<th>Input A</th>
<th>Input B</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

You can use OR as a logical operator or as a 'bit-by-bit' (bitwise) operator. The operands can be boolean (logical) or numeric.

```
IF A=2 OR B=3 THEN 110
X=B OR 4
```

BBC BASIC does not have true boolean variables; this can lead to confusion at times (see NOT for more details).

In the example below, the operands are boolean (logical). In other words, the result of the tests (IF) A=2 and (IF) B=3 is either TRUE or FALSE. The result of this example will be TRUE if A=2 or B=3.

```
answer=(A=2 OR B=3)
```

The brackets are not necessary, they have been included to make the example easier to follow.

The last example, uses the OR in a similar fashion to the numeric operators (+, -, etc).

Suppose X was -20 in the following example,
A=X OR 11

the OR operation would be:

```
11111111 11111111 11111111 11101100
00000000 00000000 00000000 00001011
11111111 11111111 11111111 11101111  = -17
```

Syntax

```
<n-var>=<numeric> OR <numeric>
<n-var>OR=<numeric>
```

Associated Keywords

AND , EOR , NOT
ORIGIN

A statement which sets the graphics origin. ORIGIN is followed by the X and Y coordinates of the new origin; subsequent graphics commands operate with respect to these coordinates. The coordinates must be in the range -32768 to +32767.

The graphics origin is reset to the bottom left-hand corner of the 'screen' (BASIC's output window) by a MODE statement or a VDU 26 command.

ORIGIN x,y is equivalent to VDU 29,x;y;

ORIGIN 640,512

Syntax

ORIGIN <numeric>,<numeric>

Associated Keywords

MODE , PLOT , VDU
OSCLI

A statement which allows a string expression to be interpreted as an 'operating system' command. It overcomes problems caused by the fact that 'star' commands cannot include variables. Using this statement, you can, for instance, erase and rename files whose names you only know at run-time.

```plaintext
command$="DEL PHONE.DTA"
OSCLI command$
command$="REN NAME.DTA ADDRESS.DTA"
OSCLI command$
```

OSCLI can issue both resident operating system commands and external commands. See the Operating System interface section for more details.

**Syntax**

```
OSCLI <string>
```

**Associated Keywords**

None.
OTHERWISE

A keyword which is an optional part of the CASE ... ENDCASE clause. OTHERWISE precedes the statement(s) which should be executed if the CASE variable matches none of the values specified in a WHEN statement. OTHERWISE must be the first item on the program line.

```plaintext
CASE die% OF
    WHEN 1,2,3 : bet$ = "lose"
    WHEN 4,5,6 : bet$ = "win"
    OTHERWISE bet$ = "cheat"
ENDCASE
```

Syntax

```
OTHERWISE [<stmt>]::<stmt>
```

Associated Keywords

```
CASE , ENDCASE , OF , WHEN
```
PAGE

A pseudo-variable controlling the starting address of the current BASIC program. When you CHAIN a program it is stored in memory at an address equal to the value of PAGE.

```
PAGE = PAGE + &1000
PRINT ~PAGE
```

PAGE is automatically initialised by BBC BASIC for Windows to the lowest available address. Normally PAGE should be treated as a read-only variable; you can change PAGE, but if you make it less than its initial value or greater than the value of HIMEM (less a safety margin of at least a kilobyte), you are almost certain to crash BASIC. If this happens you will lose all your unsaved work.

With care, several programs can be left around in RAM, at different values of PAGE, without the need for saving them. However you are strongly advised not to try this!

Syntax

```
PAGE=<numeric>
<n-var>=PAGE
```

Associated Keywords

TOP, LOMEM, HIMEM
PI

A function returning 3.14159265.

\[ X=\pi \]

You can use PI to calculate the circumference and area of a circle. The example below calculates the circumference and area of a circle of a given radius.

```basic
CLS
INPUT "What is the radius of the circle ", rad
PRINT "The circumference is: "; 2*PI*rad
PRINT "The area is: "; PI*rad*rad
END
```

PI can also be used to convert degrees to radians and radians to degrees.

\[ \text{radians} = \frac{\pi}{180} \times \text{degrees} \]
\[ \text{degrees} = \frac{180}{\pi} \times \text{radians} \]

However, \textit{BBC BASIC for Windows} has two functions (RAD and DEG) which perform these conversions to a higher accuracy.

Syntax

\[ <n\text{-var}>=\pi \]

Associated Keywords

RAD, DEG
A multi-purpose drawing statement. Three numeric values follow the PLOT keyword: the first specifies the type of point, line, triangle, circle etc. to be drawn; the second and third give the X and Y coordinates to be used (in that order). The coordinates must be in the range -32768 to +32767.

PLOT Mode,X,Y

The graphics origin (X=0, Y=0) is normally the bottom left of the 'screen' (BASIC's output window). The origin can be changed using the ORIGIN statement or the VDU 29 command.

The two most commonly used examples, PLOT 4 and PLOT 5, have the duplicate keywords MOVE and DRAW.

PLOT X,Y is synonymous with PLOT 69,X,Y
PLOT BY X,Y is synonymous with PLOT 65,X,Y
(both BBC BASIC for Windows only).

See the Plotting modes section for full details of the different PLOT commands.

Syntax

PLOT <numeric>,<numeric>,<numeric>
PLOT <numeric>,<numeric>
PLOT BY <numeric>,<numeric>

Associated Keywords

BY, CIRCLE, CLG, DRAW, ELLIPSE, FILL, GCOL, MODE, MOVE, ORIGIN, POINT, RECTANGLE, VDU
POINT

A function which returns a number giving the logical colour of the screen at the coordinates specified. If the point is outside the graphics viewport, then -1 is returned.

There must not be a space between POINT and the opening bracket.

colour=POINT(X,Y)
IF POINT(X,Y)=3 THEN 300

You can use POINT to find out the colour of the screen at the specified point and take action accordingly. In an adventure game, for example, the swamps may be marked in green. If the explorer ventured into a green area he must be in the swamp and 'swamp type demons' would be activated.

Beware that when your display is set to a 'true colour' mode (usually 32768 colours or more) POINT may not always produce the result you expect. Suppose you have set two or more logical colours to correspond to the same physical colour. You may know which logical colour you used to plot at a certain point, but BASIC only knows the physical colour. The value returned by POINT will depend upon which matching logical colour is found in the palette first. This problem doesn't arise with a 'paletted' display (usually 256 colours or fewer).

You can use TINT rather than POINT; this returns the physical (RGB) colour rather than the logical colour.

Syntax

<n-var>=POINT(<numeric>,<numeric>)

Associated Keywords

PLOT , DRAW , MOVE , GCOL , TINT
POS

A function returning the column number containing the text cursor (caret). POS returns the position of the cursor with respect to the text viewport. Column 0 is usually on the left, but this may be changed using VDU 23,16.

\[ X = \text{POS} \]

The difference between POS and COUNT is that POS always returns the true column number of the text cursor whereas COUNT returns the number of 'printing' characters output since the last carriage-return. These will often be the same, but may differ if direct cursor addressing (TAB(X,Y)) has been used, or if the cursor has 'wrapped around' at the end of the line.

See VPOS for an example of the use of POS and VPOS.

Syntax

\[ \langle n\text{-var}\rangle = \text{POS} \]

Associated Keywords

COUNT, TAB, VPOS
A statement which prints characters to the screen or printer (see the hardcopy output to a printer section for details).

The items following PRINT are called the print list. The print list may contain a sequence of string or numeric literals, variables and formatting instructions. The spacing between the items printed will vary depending on the punctuation used. If the print list does not end with a semi-colon, a new-line will be printed after all the items in the print list.

In the examples which follow, commas have been printed instead of spaces to help you count.

The 'screen' (BASIC's output window) is divided into zones (initially) 10 characters wide. By default, numeric quantities are printed right justified in the print zone and strings are printed just as they are (with no leading spaces). Numeric quantities can be printed left justified by preceding them with a semi-colon. In the examples the zone width is indicated as z10, z4 etc.

```
PRINT 23.162        ,,,,23.162
PRINT "HELLO"       HELLO
PRINT ;23.162       23.162
```

Initially numeric items are printed in decimal. If a tilde (~) is encountered in the print list, the numeric items which follow it are printed in hexadecimal. If a comma or a semi-colon is encountered further down the print list, the format reverts to decimal.

```
PRINT ~10 58,58     ,,,,,,,,,A,,,,,,,,3A,,,,,,,,58
```

A comma (,) causes the cursor to TAB to the beginning of the next print zone unless the cursor is already at the start of a print zone. A semi-colon causes the next and following items to be printed immediately after the previous item. This 'no-gap'
printing continues until a comma (or the end of the print list) is encountered. An apostrophe (') will force a new line. TAB (X) and TAB (Y,Z) can also be used at any position in the print line to position the cursor.

```
PRINT "HELLO",24.2  HELLO     ,,,,,,24.2
PRINT "HELLO";24.2  HELLO24.2
PRINT ;2 5 4.3,2    254.3     ,,,,,,,,,2
PRINT "HELLO"'2.45  HELLO
                   ,,,,,,2.45
```

Unlike most other versions of BASIC, a comma at the end of the print list will not suppress the new line and advance the cursor to the next zone. If you wish to split a line over two or more PRINT statements, end the previous print list with a semicolon and start the following list with a comma or end the line with a comma followed by a semicolon.

```
PRINT "HELLO" 12;   HELLO,,,,,,,,12,,,,,,,,,,23.67
PRINT ,23.67
```
or

```
PRINT "HELLO" 12,;
PRINT 23.67
```

Printing a string followed by a numeric effectively moves the start of the print zones towards the right by the length of the string. This displacement continues until a comma is encountered.

```
PRINT "HELLO"12 34  HELLO,,,,,,,,12,,,,,,,,34
PRINT "HELLO"12,34  HELLO,,,,,,,,12    ,,,,,,,34
```

**Print format control**

**The @% system variable**
Although PRINT USING is not implemented in BBC BASIC (but see the FNusing library function for a substitute), similar control over the print format can be obtained. The overall width of the print zones and print field, the number of figures or decimal places and the print format may be controlled by setting the system variable @% to the appropriate value. @% comprises 4 bytes and each byte controls one aspect of the print format. @% can be set equal to a decimal integer, but it is easier to use hexadecimal, since each byte can then be considered separately.

@%=&SSNPPWW

<table>
<thead>
<tr>
<th>Byte</th>
<th>Range</th>
<th>Default</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS</td>
<td>00-01</td>
<td>00</td>
<td>STR$ Format Control</td>
</tr>
<tr>
<td>NN</td>
<td>00-C2</td>
<td>00</td>
<td>Format Selection</td>
</tr>
<tr>
<td>PP</td>
<td>??-??</td>
<td>09</td>
<td>Number of Digits Printed</td>
</tr>
<tr>
<td>WW</td>
<td>00-0F</td>
<td>0A(10)</td>
<td>Zone and Print Field Width</td>
</tr>
</tbody>
</table>

**STR$ format control - SS**

Byte 3 affects the format of the string generated by the STR$ function. If Byte 3 is 1 the string will be generated according to the format set by @%, otherwise the G9 format will be used.

**Format selection - NN**

Byte 2 selects the format as follows:

- **00** General Format (G).
- **01** Exponential Format (E).
- **02** Fixed Format (F).

Adding &80 (i.e. 80, 81 or 82) causes a comma to be output instead of a decimal point.
G Format  Numbers that are integers are printed as such. Numbers in the range 0.0001 to 1 will be printed as such. Numbers less than 0.0001 will be printed in E format. Numbers greater than the range set by Byte 1 will be printed in E format. In which case, the number of digits printed will still be controlled by Byte 1, but according to the E format rules.

The earlier examples were all printed in G9 format.

E Format  Numbers are printed in the scientific (engineering) notation.

F Format  Numbers are printed with a fixed number of decimal places.

Adding &40 (i.e. 42 or C2) overrides the limit on the number of decimal places that may be output in Fixed Format (F) mode. Trailing zeros will be appended as necessary.

**Number of digits - PP**

Byte 1 controls the number of digits printed in the selected format. The number is rounded (NOT truncated) to this size before it is printed. If Byte 1 is set outside the range allowed for by the selected format, it is taken as 9. The effect of Byte 1 differs slightly with the various formats.

<table>
<thead>
<tr>
<th>Format</th>
<th>Range</th>
<th>Control Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>01-14</td>
<td>The maximum number of digits which can be printed, excluding the decimal point, before changing to the E format.</td>
</tr>
<tr>
<td>E</td>
<td>01-FF</td>
<td>The total number of digits to be printed excluding the decimal point and the digits after the E. Three characters or spaces are always printed after the E. If the number of significant figures called for is greater than 10, then trailing zeros will be printed.</td>
</tr>
</tbody>
</table>

```
01234567890123456789
\&030A - G3z10
(00'00'03'0A)
PRINT 1000.31       ,,,,,,,1E3
PRINT 1016.31       ,,,,1.02E3
PRINT 10.57         ,,,,,10.6
```
F 00-14 The number of digits to be printed after the decimal point.

```
PRINT 10.57,,1.06E1
&010F0A - E15z10
(00'01'0F'0A)
01234567890123456789
PRINT 10.57 1.05700000000000E1
```

Zone width - WW

Byte 0 sets the width of the print zones and field.

```
&02020A - F2z10
(00'02'02'0A)
01234567890123456789
PRINT 10.57,,,,10.57
PRINT 100.5864,,100.59
PRINT .64862,,,,0.65
```

Changing the Print Control Variable

It is possible to change the print control variable (@%) within a print list by using the function:

```
DEF FN_pformat(N):@%=N:=""
```

Functions have to return an answer, but the value returned by this function is a null string. Consequently, its only effect is to change the print control variable. Thus the PRINT statement
will print \( x \) in \( G9z10 \) format and \( y \) in \( F2z10 \) format. If you try to alter the zone width using this method the change will only take place at the next comma (if any) in the print list.

**Examples**

\[
\begin{array}{ll}
G9z10 & G2z10 \\
&00090A & &00020A \\
012345678901234 & 012345678901234 \\
1111.1111 & ,,,,,1.1E3 \\
13.7174211 & ,,,,,,14 \\
,1.5241579 & ,,,,,,1.5 \\
1.88167642E-2 & ,,,,,1.9E-2 \\
2.09975158E-3 & ,,,,,,2.1E-3 \\
F2z10 & E2z10 \\
&02020A & &0102A \\
012345678901234 & 012345678901234 \\
,,,,1111.11 & ,,,1.1E3 \\
,,,,,13.72 & ,,,1.4E1 \\
,,,,,1.52 & ,,,1.5E0 \\
,,,,,,0.02 & ,,,1.9E-2 \\
,,,,,,0.00 & ,,,2.1E-3 \\
\end{array}
\]

The results obtained by running the following example program show the effect of changing the zone width. The results for zone widths of 5 and 10 (&0A) illustrate what happens when the zone width is too small for the number to be printed properly. The example also illustrates what happens when the number is too large for the chosen precision.

```
test = 7.8123
FOR i = 5 TO 25 STEP 5
    PRINT @% = &020200 + i
    PRINT "@% = &000"; @%
    PRINT STRING$(3, "0123456789")
    FOR j = 1 TO 10
        PRINT test ^ j
    NEXT
    PRINT'
NEXT
@% = &90A
&00020205
012345678901234567890123456789
7.81
```
61.03
476.80
3724.91
29100.11
227338.75
1776038.54
13874945.89
1.083952398E8
8.46816132E8
&0002020A
012345678901234567890123456789
7.81
61.03
476.80
3724.91
29100.11
227338.75
1776038.54
13874945.89
1.083952398E8
8.46816132E8
&0002020F
012345678901234567890123456789
7.81
61.03
476.80
3724.91
29100.11
227338.75
1776038.54
13874945.89
1.083952398E8
8.46816132E8
&00020214
012345678901234567890123456789
7.81
61.03
476.80
3724.91
29100.11
227338.75
1776038.54
13874945.89
1.083952398E8
8.46816132E8
&00020219
012345678901234567890123456789
7.81
61.03
476.80
3724.91
29100.11
227338.75
1776038.54
13874945.89
1.083952398E8
8.46816132E8
Syntax

PRINT ([TAB(<numeric>,<numeric>)] | [SPC(<numeric>)]
['][,][;][~][<string>|<numeric>])

Associated Keywords

PRINT# , TAB , POS , STR$ , WIDTH , INPUT , VDU
A statement which writes the internal form of a numeric or string value out to a data file.

```
PRINT#E,A,B,C,D$E$,F$
PRINT#4,prn$
```

The format of the variables as written to the file differs from the format used on the BBC Micro. All numeric values are written as five bytes, eight bytes or ten bytes of binary data (see the section entitled Format of data in memory). Strings are written as the bytes in the string (in the correct order) plus a carriage return.

The format of data files differs according to the *FLOAT mode in effect at the time (in *FLOAT 40 mode numeric values are written as 5 bytes, in *FLOAT 64 mode they are written as 8 bytes and - in BBC BASIC for Windows version 6.00a or later only - in *FLOAT 80 mode they are written as 10 bytes). You must ensure that when numeric values are read with INPUT# (or READ#) the same *FLOAT mode is in effect as when the values were written.

Before you use this statement, you must normally have opened a file using OPENOUT or OPENUP.

You can use PRINT# to write data (numbers and strings) to a data file in the 'standard' manner. If you wish to 'pack' your data in a different way, you should use BPUT#. You can use PRINT# and BPUT# together to mix or modify the data format. For example, if you wish to write a 'compatible' ASCII text file, you could PRINT# the string and BPUT# a line-feed. This would write the string followed by a carriage-return and a line-feed to the file.

Remember, with BBC BASIC for Windows the format of the file is completely under your control.

**Syntax**

```
PRINT#<numeric>{,<numeric>|<string>}
```
Associated Keywords

PRINT, OPENUP, OPENOUT, CLOSE#, INPUT#, READ#, BPUT#, BGET#, EXT#, PTR#, EOF#
PRIVATE

A statement to declare variables for private use inside a function (FN ) or procedure (PROC ).

PRIVATE saves the value(s) of the specified variable(s) on the stack, and sets their values to those they had when the function or procedure was last called. The original values are restored from the stack on exit from the function or procedure (i.e. at '=' or ENDPROC ). The variables need not have been previously declared.

An entire array may be made PRIVATE, following which it is in an undimensioned state. Before the private array can be used within the function or procedure it must be re-dimensioned using a DIM statement. The new dimensions can be the same as or different from those of the original array. PRIVATE arrays are allocated on the heap, but can be accessed only in the function or procedure in which they are declared.

Similarly an entire structure may be made PRIVATE, following which it is in an undeclared state. Before the private structure can be used within the function or procedure it must be re-declared using a DIM statement. The new format (number and type of members) can be the same as or different from that of the original structure. PRIVATE structures are allocated on the heap, but can be accessed only in the function or procedure in which they are declared.

See also LOCAL which is similar to PRIVATE except that the variables are initialised to zero/null on each entry to the function/procedure rather than holding their values from one call to the next.

PRIVATE A$,X,Y%,items()

If a function or procedure is called recursively, any PRIVATE variables will acquire the values they had in the calling function/procedure. PRIVATE variables behave like globals, except that they cannot be 'seen' outside the (dynamic) scope of the function/procedure in which they are declared. It is important that your program does not 'jump out' of a function/procedure whilst PRIVATE variables are in use (e.g. as the result of an ON ERROR statement). You are advised to use ON ERROR LOCAL and RESTORE LOCAL within the function/procedure to ensure that PRIVATE variables are properly cleaned up in the event of an error:
PRIVATE b$,x,y%,struct{}
ON ERROR LOCAL RESTORE LOCAL : ERROR ERR, REPORT$

Beware that an asynchronous error (such as pressing Escape) could happen between the PRIVATE and ON ERROR statements. If that is a concern, temporarily disable such errors (e.g. using *ESC OFF).

Syntax

PRIVATE <n-var>|<s-var>|<array()>{,<n-var>|<s-var>|<array()>}
PRIVATE <struct{}>{,<struct{}>}</n-var>|<s-var>|<array()>}

Associated Keywords

DEF , ENDPROC , FN , LOCAL , ON , PROC
PROC

A keyword used to identify a user-defined procedure. PROC is followed immediately by the procedure name; the first character of a procedure name may be an underline (or a digit).

A procedure may take any number of parameters of any type (or none). It does not have to be defined before it is called.

A procedure definition is terminated by ENDPROC.

Procedures are re-entrant and the parameters (arguments) are normally passed by value. The keyword RETURN can be used in the procedure definition to specify that a parameter should instead be passed by reference. Arrays and structures are always passed by reference.

```
INPUT"Number of discs "F
PROC_hanoi(F,1,2,3)
END
:
DEF PROC_hanoi(A,B,C,D)
IF A=0 THEN ENDPROC
PROC_hanoi(A-1,B,D,C)
PRINT"Move disk ";A" from ";B" to ";C
PROC_hanoi(A-1,D,C,B)
ENDPROC
```

Single-line procedures can be placed anywhere within your program. Multi-line procedures must be placed where they will not be executed 'out of sequence', this usually means at the end of the program after the END statement. See the Procedures and functions sub-section for more information.

PROC may alternatively be followed by a numeric value contained in parentheses. This causes the procedure pointed to by the specified value to be called (an indirect call):

```
pptr% = ^PROC_hanoi()
PROC(pptr%)(a, b, c, d)
```

See the Indirect procedure and function calls sub-section for more information.
Syntax

PROC(<numeric>)[(<exp>,<exp>)]
PROC<name>[(<exp>,<exp>)]

Associated Keywords

DEF, ENDPROC, FN, LOCAL, PRIVATE, RETURN
PTR#

A pseudo-variable allowing the random-access pointer of the file whose channel number is its argument to be read and changed.

PTR#F=PTR#F+5 :REM Move pointer to next number
PTR#F=recordnumber*recordlength

Reading or writing (using BGET# , BPUT# , INPUT# , PRINT# or GET$# ) takes place at the current position of the pointer. The pointer is automatically updated following a read or write operation.

You can use PTR# to select which item in a file is to be read or written to next. In a random file (see the section on Disk files ) you use PTR# to select the record you wish to read or write.

If you wish to move about in a file using PTR# you will need to know the precise format of the data in the file.

A file opened with OPENUP may be extended by setting its pointer to its end (PTR#fnum = EXT# fnum) and then writing to it. If you do this, you must remember to CLOSE the file when you have finished with it in order to update the directory entry.

By using PTR# you have complete control over where you read and write data in a file. This is a simple concept, but it may initially be difficult to grasp its many ramifications. The Disk files section has a number of examples of the use of PTR#.

Syntax

PTR#<numeric>=<numeric>
<n-var>=PTR#<numeric>

Associated Keywords

OPENIN , OPENUP , OPENDOUT , CLOSE# , PRINT# , INPUT# , READ# , BPUT# , BGET# , GET$# , EXT# , EOF#
QUIT

A statement which causes the BASIC program to end and its window to close. The difference between QUIT and END is that END leaves the window open and any results displayed, whereas QUIT closes the window (and does not close open files). QUIT has the same effect as *QUIT and *BYE.

If you are writing a program with the intention that it will be compiled you should normally exit with QUIT rather than END. If the CLOSE button (and the close option on the window menu) has been trapped with ON CLOSE then QUIT (or *QUIT or *BYE) is the only way to close the window.

QUIT may optionally be followed by an integer parameter which determines the program's exit code. This is only useful if the program has been compiled to a standalone executable and run from a command prompt (or another program). When run from a command prompt the exit code is copied into the ERRORLEVEL environment variable.

Syntax

QUIT [<numeric>]

Associated Keywords

END, ON CLOSE, STOP
RAD

A function which converts degrees to radians.

X=RAD(Y)
X=SIN RAD(90)

_BBC BASIC for Windows_ wants angles expressed in radians. You can use this function to convert an angle expressed in degrees to radians before using one of the angle functions (SIN, COS, etc).

Using RAD is equivalent to multiplying the degree value by PI/180, but the result is calculated internally to a greater accuracy.

See COS, SIN and TAN for further examples of the use of RAD.

**Syntax**

<n-var>=RAD<numeric>

**Associated Keywords**

DEG
READ

A statement which will assign to variables values read from the DATA statements in the program. Strings must be enclosed in double quotes if they have leading spaces or contain commas.

```
READ C,D,A$
```

In many of your programs, you will want to use data values which do not change frequently. Because these values are subject to some degree of change, you won't want to use constants. On the other hand, you won't want to input them every time you run the program either. You can get the best of both worlds by declaring these values in DATA statements at the beginning or end of your program and READing them into variables in your program.

A typical use for DATA and READ is a name and address list. The addresses won't change very often, but when they do you can easily amend the appropriate DATA statement.

See DATA for more details and an example of the use of DATA and READ.

Syntax

```
READ <n-var>|<s-var>{<n-var>|<s-var>}
```

Associated Keywords

```
DATA , RESTORE
```
**READ#**

A statement which is synonymous with INPUT#. READ# and INPUT# have identical effects; READ# is implemented in the interests of improved compatibility with other dialects of BASIC.

**Syntax**

```
READ #<numeric>,<n-var>|<s-var>{,<n-var>|<s-var>}
```

**Associated Keywords**

READ , INPUT# , OPENIN , OPENUP , OPENOUT , CLOSE# , PRINT# , BPUT# , BGET# , EXT# , PTR# , EOF# , GET$#
RECTANGLE

A statement which draws an outline rectangle or a filled rectangle. RECTANGLE is followed by the X and Y coordinates of the bottom left-hand corner of the rectangle then the width and height of the rectangle. To draw a square you can omit the fourth parameter; the height is then assumed to be the same as the width. To draw a filled (solid) rectangle or square use RECTANGLE FILL.

The graphics origin (X=0, Y=0) is normally the bottom left of the 'screen' (BASIC's output window). The origin can be changed using the VDU29 command or the ORIGIN statement.

The rectangle is drawn in the current graphics foreground colour. This colour can be changed using the GCOL statement.

RECTANGLE can also be used to copy or move a rectangular area of the screen to a different place. In this case RECTANGLE is followed by the X and Y coordinates of the bottom left-hand corner of the source rectangle, the width and height of the rectangle, the keyword TO and finally the X and Y coordinates of the bottom left-hand corner of the destination rectangle. To move the rectangle rather than copy it, use RECTANGLE FILL. To swap two rectangular areas use RECTANGLE SWAP.

\[\text{RECTANGLE} \ x_{\text{pos}}, y_{\text{pos}}, \text{width}, \text{height}\]
\[\text{RECTANGLE FILL} \ 400,500,200\]
\[\text{RECTANGLE} \ 100,100,200,200 \ TO \ 300,300\]
\[\text{RECTANGLE FILL} \ srcx,srcy,dx,dy \ TO \ dstx,dsty\]
\[\text{RECTANGLE SWAP} \ x_1,y_1,dx,dy \ TO \ x_2,y_2\]

RECTANGLE is also used in the MOUSE RECTANGLE statement.

Syntax

\[\text{RECTANGLE [FILL]} \ <\text{numeric}>,<\text{numeric}>,<\text{numeric}>[,<\text{numeric}>] \ [\TO <\text{numeric}>,<\text{numeric}>] \]
\[\text{RECTANGLE SWAP} \ <\text{numeric}>,<\text{numeric}>,<\text{numeric}>[,<\text{numeric}>] \ TO <\text{numeric}>,<\text{numeric}>\]

Associated Keywords

FILL, MOUSE, SWAP, TO
REM

A statement that causes the rest of the line to be ignored thereby allowing comments to be included in a program.

You can use the REM statement to put remarks and comments into your program to help you remember what the various bits of your program do. BBC BASIC for Windows completely ignores anything on the line following a REM statement.

You will be able to get away without including any REMarks in simple programs. However, if you go back to a lengthy program after a couple of months you will find it very difficult to understand if you have not included any REMs.

If you include nothing else, include the name of the program, the date you last revised it and a revision number at the start of your program.

REM WSCONVERT REV 2.30
REM 5 AUG 84
REM Converts 'hard' carriage-returns to 'soft'
REM ones in preparation for use with WS.

Syntax

REM any text

Associated Keywords

None
REPEAT

A statement which is the starting point of a REPEAT...UNTIL loop. The purpose of a REPEAT...UNTIL loop is to make *BBC BASIC for Windows* repeat a set number of instructions until some condition is satisfied:

```
REPEAT UNTIL GET=13 :REM wait for CR
    X=0
    REPEAT
        X=X+10
        PRINT "What do you think of it so far?"
    UNTIL X>45
```

You must not exit a REPEAT...UNTIL loop with a GOTO (see the sub-section on Program Flow Control for more details). You can force a premature end to the loop with the EXIT REPEAT statement.

REPEAT...UNTIL loops may be nested.

**Syntax**

```
REPEAT
```

**Associated Keywords**

```
EXIT , UNTIL , WHILE
```
REPORT/REPORT$

A statement (REPORT) which outputs, and a string function (REPORT$) which returns, the error string associated with the last error which occurred.

You can use these keywords within your own error handling routines to print out or process an error message for those errors you are not able to cope with.

The example below is an error handling routine designed to clear the screen before reporting the error:

```
ON ERROR MODE 3 : REPORT : PRINT " at line ";ERL : END
```

See the sub-section on Error Handling and the keywords ERR, ERL and ON ERROR for more details.

Syntax

```
REPORT
<s-var> = REPORT$
```

Associated Keywords

```
ERR, ERROR, ERL, ON ERROR
```
RESTORE

A statement which moves the data pointer. RESTORE can be used at any time in a program to set the line from where READ reads the next DATA item.

RESTORE on its own resets the data pointer to the first data item in the program.

RESTORE followed by a line number or label sets the data pointer to the first item of data in the specified line (or the next line containing a DATA statement if the specified line does not contain data). This optional parameter for RESTORE can specify a calculated line number.

RESTORE followed by a plus sign (+) and a positive numeric value sets the data pointer to the first item of data in the line offset from the line containing the RESTORE statement by the specified number of lines (or the next DATA statement if the specified line does not contain any data). When using values greater than +1 it is advisable to put the DATA statements immediately after the RESTORE statement to avoid incorrect results if the program is compiled.

RESTORE DATA

RESTORE DATA causes the DATA pointer saved by a previous LOCAL DATA statement to be restored from the stack.

RESTORE ERROR

RESTORE ERROR causes the error-trapping status saved by a previous ON ERROR LOCAL statement to be restored from the stack.

You can use RESTORE to reset the data pointer to the start of your data in order to re-use it. Alternatively, you can have several DATA lists in your program and use RESTORE to set the data pointer to the appropriate list.
RESTORE LOCAL

RESTORE LOCAL (which can be used only inside a user-defined procedure or function) restores the values of formal parameters and LOCAL / PRIVATE variables to those they had before the procedure/function was called. RESTORE LOCAL also performs the action of RESTORE ERROR. This is primarily intended for use within an ON ERROR LOCAL error handler.

Syntax

RESTORE [<l-num>]
RESTORE [[<numeric>]]
RESTORE +<numeric>
RESTORE DATA
RESTORE ERROR
RESTORE LOCAL

Associated Keywords

READ, DATA, ON ERROR LOCAL
A statement causing a RETURN to the statement after the most recent GOSUB statement, or a RETURN to the calling program after a CALL statement, or a RETURN to the original code following an event trapped with ON CLOSE, ON MOUSE, ON MOVE, ON SYS or ON TIME.

You use RETURN at the end of a subroutine to make BBC BASIC for Windows return to the place in your program which originally 'called' the subroutine.

You may have more than one return statement in a subroutine, but it is preferable to have only one entry point and one exit point (RETURN).

Try to structure your program so you don't leave a subroutine with a GOTO. If you do, you should always return to the subroutine and exit via the RETURN statement. If you insist on using GOTO all over the place, you will end up confusing yourself and maybe confusing BBC BASIC for Windows as well. The sub-section on Program Flow Control explains why.

RETURN may optionally be followed by a destination (line-number or label) which causes the subroutine or event-handler to return to the specified line rather than to continue execution from where it was called.

Syntax

```
RETURN
RETURN <l-num>
RETURN (<numeric>)
```

RETURN is also used in a function or procedure DEFinition to specify that a parameter should be passed by reference rather than by value. When used with a whole array or structure parameter, RETURN allows the array or structure to be declared (using DIM) within the function or procedure. When used with a procedure or (particularly) function parameter RETURN specifies that a pointer to the function, not the value returned by the function, will be received by the formal parameter.
Associated Keywords

CALL, DEF, GOSUB, ON GOSUB ON CLOSE, ON MOUSE, ON MOVE, ON SYS, ON TIME
RIGHT$

A string function which returns the right 'num' characters of the string. If there are insufficient characters in the string then all are returned. RIGHT$ may also be used to the left of an equals sign to change the end of a string whilst leaving the rest alone.

There must not be any spaces between the RIGHT$ and the opening bracket.

\[
A$ = \text{RIGHT$(A$,num)} \\
A$ = \text{RIGHT$(A$,2)} \\
A$ = \text{RIGHT$(\text{LEFT$(A$,3)},2)} \\
\text{RIGHT$(A$,3) = B$}
\]

For example,

```basic
name$="BBC BASIC for Windows"
FOR i=3 TO 13
    PRINT \text{RIGHT$}(name$,i)
NEXT
END
```

would print

```plaintext
ows
dows
ndows
indows
Windows
Windows
r Windows
or Windows
for Windows
for Windows
C for Windows
```

When using RIGHT$ on the left of an equals sign, and the expression to the right of the equals sign evaluates to a string with fewer characters than the specified substring length, only that number of characters is changed. For example:

\[
A$ = \text{"BBC BASIC"}
\]
RIGHT$(A$,4) = "ZZ"

will set A$ equal to "BBC BASZZ". Although the sub-string length is set to four, only the last two characters are actually modified since that is the length of the string "ZZ".

RIGHT$(A$) is shorthand for RIGHT$(A$,1), in other words it returns or sets the last character of A$.

Syntax

<s-var>=RIGHT$(<string>[,<numeric>])
RIGHT$(<s-var>[,<numeric>])=<string>

Associated Keywords

LEFT$ , MID$ , INSTR
**RND**

A function which returns a random number. The type and range of the number returned depends upon the optional parameter.

- **RND** returns a random integer 0 - &FFFFFFFF.
- **RND(n)** returns an integer in the range 1 to n (n>1).
- **RND(1)** returns a real number in the range 0.0 to .99999999.

If n is negative the pseudo random sequence generator is set to a number based on n and n is returned.

If n is 0 the last random number is returned in RND(1) format.

```plaintext
X=RND(1)
X%=RND
N=RND(6)
```

**Syntax**

```plaintext
<n-var>=RND[<numeric>]
```

**Associated Keywords**

None
RUN

A statement which starts or restarts execution of the program.

RUN

All variables except the static variables @% to Z% are CLEAR ed by RUN.

If you want to start a program without clearing all the variables, you can use the statement

GOTO nnnn

where nnnn is number of the line at which you wish execution of the program to start.

RUN "filename" can be used as an alternative to CHAIN "filename".

Syntax

RUN
RUN <string>

Associated Keywords

CHAIN
SGN

A function returning -1 for negative argument, 0 for zero argument and +1 for positive argument.

\[ X = \text{SGN}(Y) \]
\[ \text{result} = \text{SGN}(\text{answer}) \]

You can use this function to determine whether a number is positive, negative or zero.

SGN returns:

+1 for positive numbers
0 for zero
-1 for negative numbers

Syntax

\[ <n\text{-var}> = \text{SGN}(<\text{numeric}>) \]

Associated Keywords

ABS
SIN

A function giving the sine of its radian argument.

\[ X = \sin(Y) \]

This function returns the sine of an angle. The angle must be expressed in radians, not degrees.

Whilst the computer is quite happy dealing with angles expressed in radians, you may prefer to express angles in degrees. You can use the RAD function to convert an angle from degrees to radians.

The example below sets Y to the sine of the angle ‘degree_angle’ expressed in degrees.

\[ Y = \sin(\text{RAD}(\text{degree_angle})) \]

Syntax

\[ <n\text{-var}> = \sin(<\text{numeric}> \) \]

Associated Keywords

\[ \text{COS, TAN, ACS, ASN, ATN, DEG, RAD} \]
SOUND

A statement which generates sounds using the sound card (if fitted). If no suitable sound card is fitted, or if the sound system is currently in use by another program, the trappable Device unavailable error will result.

The SOUND statement is followed by four parameters which specify the sound channel to be used (and special effects), the amplitude of the note (or the envelope number), the (initial) pitch of the note and how long the note is to last.

\[
\text{SOUND } \text{Channel,Amplitude,Pitch,Duration}
\]

Channel/Effects

If the channel/effects value is represented by the hexadecimal number &HSFC the individual digits (nibbles) have the following meaning:

C = channel number: There are four sound channels, numbered 0 to 3. Channel 0 is a special channel that can produce 'noise'; channels 1-3 are used to produce 'tones'. Channel 0 can be configured to be a 'tone' channel by adding 128 to the *TEMPO value.

F = flush control: If F is 1, the sound statement in which it occurs 'flushes' any notes waiting in the queue for that channel. This causes the sound to be played immediately.

S = synchronisation: If S is 0 the note is played as soon as the last note on that channel has completed (i.e. reached the start of the 'release' phase). If S is non-zero the note does not play until a corresponding note on another channel is ready to be played. A value of S=1 implies that there is one other note in the group, S=2 implies two other notes (i.e. a total of three) etc.

H = hold note: If H=1 any note currently playing on the selected channel is allowed to continue without being interrupted. The Amplitude and Pitch parameters are ignored. If an envelope is in use, the release phase is entered.

Amplitude/Envelope
If the Amplitude parameter is negative or zero it determines the loudness of the note, from -15 (loudest) to 0 (silent). If the Amplitude parameter is positive it selects one of 16 envelopes (1-16). See the keyword ENVELOPE for more details.

**Pitch**

This sets the pitch of the note (initial pitch in the case of a pitch envelope), in units of quarter-semitones. Middle C is generated with the value 100 and International Standard Pitch (440 Hz) with the value 136.

To go up or down an octave, the Pitch must be changed by 48.

To go up a perfect 5th, the Pitch must be increased by 28.

Increasing the Pitch by 1 will increase the frequency of the note produced by \( \frac{1}{4} \) of a semitone.

<table>
<thead>
<tr>
<th>Note</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>48</td>
<td>96</td>
<td>144</td>
<td>192</td>
<td>240</td>
<td></td>
</tr>
<tr>
<td>A#</td>
<td>44</td>
<td>92</td>
<td>140</td>
<td>188</td>
<td>236</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>40</td>
<td>88</td>
<td>136</td>
<td>184</td>
<td>232</td>
<td></td>
</tr>
<tr>
<td>G#</td>
<td>36</td>
<td>84</td>
<td>132</td>
<td>180</td>
<td>228</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>32</td>
<td>80</td>
<td>128</td>
<td>176</td>
<td>224</td>
<td></td>
</tr>
<tr>
<td>F#</td>
<td>28</td>
<td>76</td>
<td>124</td>
<td>172</td>
<td>220</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>24</td>
<td>72</td>
<td>120</td>
<td>168</td>
<td>216</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>20</td>
<td>68</td>
<td>116</td>
<td>164</td>
<td>212</td>
<td></td>
</tr>
<tr>
<td>D#</td>
<td>16</td>
<td>64</td>
<td>112</td>
<td>160</td>
<td>208</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>12</td>
<td>60</td>
<td>108</td>
<td>156</td>
<td>204</td>
<td>252</td>
</tr>
<tr>
<td>C#</td>
<td>8</td>
<td>56</td>
<td>104</td>
<td>152</td>
<td>200</td>
<td>248</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td>52</td>
<td>100</td>
<td>148</td>
<td>196</td>
<td>244</td>
</tr>
</tbody>
</table>

If the channel number is zero, the pitch parameter behaves differently. Values 4 to 7 result in 'white' noise and values 0 to 3 result in periodic 'buzzing' noises. Values 3 and 7, which on the BBC Micro allow you to control the 'pitch' of the noise, are not implemented. They have the same effect as values 1 and 5 respectively.
Duration

This sets the duration of the note, in units of twentyths of a second (by default). A value of -1 signifies 'play indefinitely' (i.e. the note will sound until you explicitly stop it by pressing <Esc>, sending another note with the 'flush control' set to 1, or executing the SOUND OFF statement). In the case of an amplitude envelope, the duration is the period before the note enters the 'release' phase.

The duration can have any value between -1 and 254.

You can change the units of the duration parameter to a value between 1 centisecond (1/100 second) and 5 centiseconds (1/20 second), using the *TEMPO command.

Examples

The example below simply generates two notes one after the other. The first note is middle C and it sounds for one (20/20) second. The second note is C\textsuperscript{1}; it sounds for two (40/20) seconds.

\begin{verbatim}
SOUND 1,-15,100,20
SOUND 1,-15,148,40
\end{verbatim}

In the next example, the first SOUND statement has a duration of -1. This will cause the note to continue until a SOUND statement with bit 4 of the channel number set cancels it (or <Esc> is pressed). The second line sets up another note and adds it to the sound queue. However, this note does not sound immediately because the first note is still sounding. After a short pause the final SOUND statement is executed. The channel number has bit 4 set (add 16 to the channel number), so the sound queue is flushed by this command and its note (middle C) is sounded immediately. The sound added to the queue by the second SOUND statement is lost when the sound queue is flushed.

\begin{verbatim}
SOUND 1,-15,52,-1
SOUND 1,-15,56,50
WAIT 200
SOUND &11,-15,100,10
\end{verbatim}

The following example illustrates the use of the synchronisation parameter. The first two SOUND statements specify that notes should be played on channels 1 and 3, but
because their S values are set to 2 they wait until a third channel (also with S = 2) is ready to play. The third SOUND statement plays a high-pitched sound on channel 2 for one second, but since S = 0 channels 1 and 3 continue to wait. The fourth SOUND statement then plays a note on channel 2 with S = 2; as a result all three channels start to play simultaneously:

```
SOUND &201,-15,100,20
SOUND &203,-15,116,20
SOUND 2,-15,200,20
SOUND &202,-15,128,20
```

The next example uses a pitch ENVELOPE. The sound produced is a slow 'warble' about the chosen note. Each step of the envelope is 0.1 (10/100) seconds. The pitch of the note will change by ¼ semitone every step of section 1 of the envelope, by -¼ semitone for every step of section 2, and by ¼ semitone again for every step of section 3. Section 1 will last for 10 steps (1 second), section 2 for 20 (2 seconds) steps and section 3 for 10 steps. The SOUND statement initially sounds middle C. The duration is -1, so you will need to press <Esc> to stop it.

```
ENVELOPE 1,10,1,-1,1,10,20,10,20,10,126,0,0,-126,126,0
SOUND 1,1,100,-1
```

The final example uses an amplitude ENVELOPE, and also illustrates the use of the 'hold' (H) bit. The envelope specifies an initial attack rate of 5 until the level reaches 126, followed by a decay rate of -5 until the level reaches 63. The level is then maintained (sustain = 0) for the rest of the note's duration, after which the level falls to zero at a rate of -5 during the 'release' phase.

The second SOUND statement has the 'hold' bit set, causing the note being played on channel 1 to continue for another ¾ second, which is enough for the release phase to complete (the amplitude and pitch parameters in this SOUND statement are ignored). If it were not present, the release phase would never occur since another note is waiting to sound on the same channel:

```
ENVELOPE 1,0,0,0,0,0,0,5,-5,0,-5,126,63
SOUND 1,1,148,40
SOUND &1001,-15,100,15
SOUND 1,-15,200,10
```
Syntax

SOUND <numeric>,<numeric>,<numeric>,<numeric>
SOUND OFF

Associated Keywords

ENVELOPE
SPC

A statement which prints a number of spaces to the screen. The argument specifies the number of spaces to be printed.

SPC can only be used within a PRINT or INPUT statement.

```
PRINT DATE; SPC(6); SALARY
INPUT SPC(10) "What is your name ", name$
```

Syntax

```
PRINT SPC(<numeric>)
INPUT SPC(<numeric>)
```

Associated Keywords

```
TAB, PRINT, INPUT
```
**SQR**

A function returning the square root of its argument.

\[ X = \text{SQR}(Y) \]

If you attempt to calculate the square root of a negative number, a 'Negative root' error will occur. You could use error trapping to recover from this error, but it is better to check that the argument is not negative before using the SQR function.

**Syntax**

\[ <\text{var}> = \text{SQR}(<\text{numeric}>) \]

**Associated Keywords**

None
STEP

Part of the FOR statement, this optional section specifies step sizes other than 1.

```
FOR i=1 TO 20 STEP 5
```

The step may be positive or negative. STEP is optional; if it is omitted, a step size of +1 is assumed.

You can use this optional part of the FOR...TO...STEP...NEXT structure to specify the amount by which the FOR ...NEXT loop control variable is changed each time round the loop. In the example below, the loop control variable, 'cost' starts as 20, ends at 5 and is changed by -5 each time round the loop.

```
FOR cost=20 TO 5 STEP -5
  PRINT cost,cost*1.15
NEXT
```

STEP is also used in the TRACE STEP statement.

Syntax

```
FOR <n-var>=<numeric> TO <numeric> [STEP <numeric>]
```

Associated Keywords

```
FOR , TO , NEXT
```
STOP

A statement which is syntactically equivalent to END. STOP also prints a message to the effect that the program has stopped.

You can use STOP at various places in your program to aid debugging. Once your program has STOPped you can investigate the values of the variables to find out why things happened the way they did.

STOP does not close data files. If you use STOP to exit a program for debugging, CLOSE all the data files before RUN ning the program again. If you don't you may get unexpected results.

Syntax

```
STOP
```

Associated Keywords

```
END, QUIT
```
**STR$**

A string function which returns the string form of the numeric argument as it would have been printed.

If the most significant byte of @% is not zero, STR$ uses the current @% description when generating the string. If it is zero (the initial value) then the G9 format (see PRINT) is used.

If STR$ is followed by ~ (tilde) then a hexadecimal conversion is carried out.

```
A$=STR$(PI)
B$=STR$~(100) :REM B$ will be "64"
```

The opposite function to STR$ is performed by the VAL function.

**Syntax**

```
<s-var>=STR$[~](<numeric>)
```

**Associated Keywords**

VAL, PRINT
**STRING$**

A function returning N concatenations of a string.

A$=STRING$(N,"hello")
B$=STRING$(10,"-*-")
C$=STRING$(Z%,S$)

You can use this function to print repeated copies of a string. It is useful for printing headings or underlinings. The last example for PRINT uses the STRING$ function to print the column numbers across the page. For example,

```
PRINT STRING$(4,"-=-*-=-")
```

would print

```
-=-=-=-=-=-=-=-=-=-
```

and

```
PRINT STRING$(3,"0123456789")
```

would print

```
012345678901234567890123456789
```

**Syntax**

```
<s-var>=STRING$(<numeric>,<string>)
```

**Associated Keywords**

None
SUM

A function which returns the sum of all the elements of a numeric array (floating-point, integer or byte) or the concatenation of all the strings in a string array.

```
DIM marks%(10)
marks%() = 0,1,2,3,4,5,6,7,8,9,10
PRINT SUM(marks%())
```

The above program segment will print the value 55, being the sum of the eleven values in the array marks%(10).

If the array has more than one dimension, all the elements in all the dimensions are summed:

```
DIM grid(7,7)
grid() = PI
PRINT SUM(grid())
```

This will print the value 201.06193, being PI * 64 (64 being the number of elements in the 8 x 8 array).

SUMLEN

A function which returns the total length of all the strings in a string array. The difference between SUMLEN(array$()) and LENSUM(array$()) is that the former will work correctly whatever the total length of the strings, whereas the latter may result in the String too long error.

**Syntax**

```
<n-var>=SUM(<n-array()>)
<s-var>=SUM(<s-array()>)
<n-var>=SUMLEN(<s-array()>)
```

**Associated Keywords**
SWAP

A statement which swaps (exchanges) the contents of two variables, two arrays or two structures. The variables or arrays must be of the same type, for example two integer numeric variables, or two string arrays. If the types differ, a Type mismatch error will result. If two arrays are swapped, their dimensions are also swapped. If two structures are swapped, their formats (number and type of members) are also swapped.

```plaintext
SWAP a,b
SWAP Coat$,Hat$
SWAP A$,B$
SWAP my_struct{},your_struct{}
```

SWAP is also used in the RECTANGLE SWAP statement.

Syntax

```plaintext
SWAP <n-var>,<n-var>
SWAP <s-var>,<s-var>
SWAP <array()>,<array()>
SWAP <struct{}>,<struct{}>
```

Associated Keywords

```
DIM , RECTANGLE
```
SYS

A statement which calls a Windows™ Application Program Interface function. SYS is followed by the function's address or the function's name as a (case sensitive) string plus, optionally, a list of comma-separated numeric or string parameters to be passed to the function. An integer value may be returned from the function by adding TO followed by a numeric variable name.

SYS "MessageBeep", 16
SYS "GetTickCount" TO tick%
SYS "PlaySound", "SystemStart", 0, &10001

SYS is a very powerful statement. It allows you to access the full range of facilities provided by the Windows™ Application Program Interface. Further details can be found in the Accessing the Windows API section.

SYS is also used in the ON SYS statement.

Syntax

SYS <numeric>|<string> {,<numeric>|<string>} [TO <n-var>]

Associated Keywords

CALL , ON SYS
TAB

A keyword available in PRINT or INPUT.

There are two versions of TAB: TAB(X) and TAB(X,Y) and they are effectively two different keywords.

TAB(X) is a printer orientated statement. The number of printable characters since the last new-line (COUNT) is compared with X. If X is equal or greater than COUNT, sufficient spaces to make them equal are printed. These spaces will overwrite any characters which may already be on the screen. If X is less than COUNT, a new-line will be printed first.

PRINT TAB(10);A$

TAB(X) may not have the desired effect if a proportional-spaced font is selected, or when UTF-8 mode is enabled.

TAB(X,Y) is a screen orientated statement. It will move the cursor on the screen to character cell X,Y (column X, row Y) if possible. No characters are overwritten and COUNT is NOT updated. Consequently, a TAB(X,Y) followed by a TAB(X) will give unpredictable (at first glance) results.

The coordinates are with respect to the current text viewport; column 0 is normally on the left and row 0 is normally at the top, but this can be changed using VDU 23,16. If the specified coordinates correspond to a location outside the current text viewport the behaviour is undefined.

PRINT TAB(X,Y);B$

Note that text written in the VDU 5 (print text at graphics cursor) mode is not affected by TAB(X,Y). In that mode you must position the text using graphics statements (for example MOVE). You can achieve a similar functionality to TAB(X,Y) by using the following user-defined function:

DEF FNTAB(X%,Y%) : X% = X%*16 : Y% = 1023−Y%*32
= CHR$25+CHR$4+CHR$(X%) +CHR$(X%DIV256)+CHR$(Y%) +CHR$(Y%DIV256)
which you would call in a context such as

    PRINT FNTAB(5,5) "Text in VDU 5 mode"

The constants in the above function are correct for screen MODE s 0 and 8. For other modes replace 16 with the appropriate character width, 32 with the appropriate character height and 1023 with one less than the height of the current graphics viewport (all in BBC BASIC graphics units).

Syntax

    PRINT TAB(<numeric>[,<numeric>])
    INPUT TAB(<numeric>[,<numeric>])

Associated Keywords

    POS , VPOS , PRINT , INPUT
TAN

A function giving the tangent of its radian argument.

\[ X = \tan(Y) \]

This function returns the tangent of an angle. The angle must be expressed in radians, not degrees.

Whilst the computer is quite happy dealing with angles expressed in radians, you may prefer to express angles in degrees. You can use the RAD function to convert an angle from degrees to radians.

The example below sets \( Y \) to the tangent of the angle 'degree_angle' expressed in degrees.

\[ Y = \tan(\text{RAD}(\text{degree_angle})) \]

**Syntax**

\[ <n-var> = \text{TAN} <\text{numeric}> \]

**Associated Keywords**

COS, SIN, ACS, ATN, ASN, DEG, RAD
An optional part of the IF ... THEN ... ELSE statement. It introduces the action to be taken if the testable condition evaluates to a non-zero value.

```
IF A=B THEN 3000
IF A=B THEN PRINT "Equal" ELSE PRINT "Help"
IF A=B THEN
    PRINT "Equal"
ELSE
    PRINT "Help"
ENDIF
```

You need to use THEN if it is:

- Followed directly by a line number:

  ```
  IF a=b THEN 320
  ```

- Followed by a 'star' (*) command:

  ```
  IF a=b THEN *DIR
  ```

- Followed by an assignment of a pseudo-variable:

  ```
  IF a=b THEN TIME=0
  ```

- Introducing a multi-line IF ... ENDIF clause:

  ```
  IF a=b THEN
    PRINT "Equals"
  ENDIF
  ```

or you wish to exit from a function as a result of the test:

```python
DEF FN_test(a,b,num)
```
IF (a>b) THEN =num
   =num/256

When THEN introduces a multi-line IF ... ENDIF statement it must be the very last thing on the line, not even followed by a REMark.

THEN may be followed immediately by a semicolon (;) to aid translation of the ELSEIF keyword available in some other BASIC dialects, for example:

```basic
IF condition1 THEN
   PROCone
ELSEIF condition2 THEN;
   PROCtwo
ELSEIF condition3 THEN;
   PROCthree
ENDIF
```

This facility is intended for use only by automatic translators, rather than in new programs.

**Syntax**

```basic
IF <t-cond> THEN <stmt>{:<stmt>}[ELSE <stmt>{:<stmt>}]
```

**Associated Keywords**

IF, ELSE, ENDF
A pseudo-variable which reads and sets the elapsed time clock.

```
X=TIME
TIME=100
```

You can use TIME to set and read BBC BASIC for Windows 's internal clock. The value of the clock is returned in centi-seconds (one-hundredths of a second) and it is quite accurate. However, you should not assume that TIME is updated 100 times per second; depending on the version of Windows™, it may change only about 70 times per second (each update adding a value greater than 1 to the elapsed time count when necessary). For this reason, a delay loop such as REPEAT UNTIL TIME=T can fail. The compound conditional test REPEAT UNTIL TIME >=T should always be used.

TIME is also used in the ON TIME statement.

**Syntax**

```
TIME=<numeric>
<n-var>=TIME
```

**Associated Keywords**

```
TIME$ , ON TIME , WAIT
```
TIME$

A 24 character long string pseudo-variable which reads and sets the system clock. The format of the character string is:

Day.dd Mon yyyy, hh:mm:ss

Where:

- **Day** is the day of the week (Mon, Tue, etc).
- **dd** is the day of the month (01, 02, etc).
- **Mon** is the abbreviated month name (Jan, Feb, etc).
- **yyyy** is the year (2003, 2004, etc).
- **hh** is hours (00 to 23).
- **mm** is minutes (00 to 59).
- **ss** is seconds (00 to 59).

The format is similar to that used on the Master Series BBC Micro except that the full-stop and comma are exchanged.

clock$=TIME$
TIME$="Sun.02 Feb 2003,18:33:30"

The time, date or both time and date may be set as shown below.

TIME$="Day.dd Mon yyyy" sets the date.
TIME$="hh:mm:ss" sets the time.
TIME$="Day.dd Mon yyyy, hh:mm:ss" sets date & time.

When setting the clock, the day of the week is ignored and may be omitted.

The first example below sets only the date and the second sets the date and the time.

TIME$="02 Feb 2003"
clock$="Mon.03 Feb 2003, 22:31:15"
TIME$=clock$
No error is reported if the format is not accurately adhered to, however the clock may be set incorrectly.

Because the system clock is a shared resource, and changing it could adversely affect other programs, you can only assign a new value to TIME$ if you have the appropriate administrative privileges.

**Syntax**

```
TIME$=<string>
<s-var>=TIME$
```

**Associated Keywords**

TIME
TINT

A function which returns an integer containing the RGB (Red, Green, Blue) value of the physical colour at the specified screen coordinates. If the point is outside the graphics viewport, then the value -1 is returned.

The colour is packed into a 32-bit integer such that the red component is in the least-significant byte, the green component is in the second byte and the blue component is in the third byte. Each component can have a value between 0 (black) and 255 (maximum brightness). The most-significant byte is always zero, except when the value -1 is returned. For example, if the returned value is &FF0000 the colour is a bright blue, and if the value is &400040 the colour is a dark purple.

The difference between TINT and POINT is that TINT returns the physical colour whereas POINT returns the logical colour (palette index).

rgb% = TINT(X,Y)

You can use TINT to find out the colour of the screen at the specified point and take action accordingly. In an adventure game, for example, the swamps may be marked in green. If the explorer ventured into a green area he must be in the swamp and 'swamp type demons' would be activated.

Syntax

<n-var>=TINT(<numeric>,<numeric>)

Associated Keywords

PLOT, DRAW, MOVE, GCOL, POINT
TO

The part of the FOR ... TO ... STEP statement which introduces the terminating value for the loop. When the loop control variable exceeds the value following 'TO' the loop is terminated.

For example,

```
FOR i=1 TO 5 STEP 1.5
    PRINT i
NEXT
PRINT "**********"
PRINT i
```

will print

```
1
2.5
4
**********
5.5
```

Irrespective of the initial value of the loop control variable and the specified terminating value, the loop will execute at least once. For example,

```
FOR i= 20 TO 10
    PRINT i
NEXT
```

will print

```
20
```

TO is also used in the SYS, RECTANGLE and MOUSE statements, and in the GET$#channel function.

**Syntax**

```
FOR <n-var>=<numeric> TO <numeric> [STEP <numeric>]
```
Associated Keywords

FOR, GET$, MOUSE, NEXT, RECTANGLE, STEP, SYS
TOP

A function which returns the value of the first free location after the end of the current program.

The length of your program is given by TOP-PAGE.

PRINT TOP-PAGE

Syntax

<n-var>=TOP

Associated Keywords

PAGE, HIMEM, LOMEM
TRACE

A statement which causes the interpreter to print executed line numbers when it encounters them. TRACE ON prints all line numbers that are executed.

TRACE X sets a limit on the range of line numbers which will be printed out. Only those line numbers less than X will appear. If you are careful to place all your subroutines, procedures and functions at the end of the program, you can display the main structure of the program without cluttering up the trace.

TRACE OFF turns trace off. TRACE is also turned off if an error is reported or you press <Esc>.

An alternative (and often better) way of tracing execution of your program is with the Trace command in the Utilities menu.

TRACE STEP ON (or TRACE STEP) enables single-step (pause) mode so you can step through a section of your program one statement (or line) at a time. TRACE STEP OFF restores normal execution.

Syntax

\[
\begin{align*}
\text{TRACE } & \text{ ON|OFF|<l-num>} \\
\text{TRACE } & \text{ ON|OFF|(<numeric>)} \\
\text{TRACE } & \text{ STEP [ON|OFF]}
\end{align*}
\]

Associated Keywords

None
TRUE

A function returning the value -1.

```plaintext
flag=FALSE
....
IF answer$=correct$ flag=TRUE
....
IF flag PROC_got_it_right ELSE PROC_wrong
```

*BBC BASIC for Windows* does not have true Boolean variables. This can lead to confusion; see the keyword NOT for details.

**Syntax**

```plaintext
<n-var>=TRUE
```

**Associated Keywords**

AND EOR FALSE NOT OR
UNTIL

A keyword which marks the end of a REPEAT ... UNTIL loop.

You can use a REPEAT...UNTIL loop to repeat a set of program instructions until some condition is met.

If the condition associated with the UNTIL statement is never met, the loop will execute for ever (at least, until <Esc> is pressed or some other error occurs).

The following example will continually ask for a number and print its square. The only way to stop it is by pressing <Esc> or forcing a 'Too big ' error.

```cpp
z=1
REPEAT
  INPUT "Enter a number " num
  PRINT "The square of ";num;" is ";num*num
UNTIL z=0
```

Since the result of the test z=0 is ALWAYS FALSE, we can replace z=0 with FALSE . The program now becomes:

```cpp
REPEAT
  INPUT "Enter a number " num
  PRINT "The square of ";num;" is ";num*num
UNTIL FALSE
```

This is a much neater way of unconditionally looping than using a GOTO statement. The program executes at least as fast and the section of program within the loop is highlighted by the indentation.

See the keyword REPEAT for more details on REPEAT...UNTIL loops.

Syntax

```cpp
UNTIL <t-cond>
```
Associated Keywords

REPEAT
**USR**

A function which allows an assembly language routine to return a value directly.

USR calls the machine code subroutine whose start address is its argument. Prior to calling the subroutine, the processor's EAX, EBX, ECX and EDX registers are initialised to the contents of A%, B%, C% and D% respectively (see also CALL). FLAGS is initialised to the least significant word of F%. However, you cannot disable interrupts nor enter single-step mode by setting F% to an appropriate value because this could affect the operation of Windows™.

Your machine-code routine should return to *BBC BASIC for Windows* with a RET instruction, but a RETF instruction can also be used for improved compatibility with BBC BASIC (86).

USR provides you with a way of calling a machine code routine which is designed to return one integer value. Parameters are passed via the processor's registers and the machine code routine returns a 32-bit (signed) integer result in the processor's EAX register.

\[ x = \text{USR}(\text{lift\_down}) \]

Unlike CALL, USR returns a result. Consequently, you must assign the result to a variable. It may help your understanding if you look upon CALL as the machine code equivalent to a PROCedure and USR as equivalent to a Function.

**Operating system interface**

USR and CALL operate differently when addresses in the range &FF00 to &FFFF are used. See the Operating System interface section for more details.

**Syntax**

\[ <\text{n-var}> = \text{USR}(<\text{numeric}>) \]

**Associated Keywords**
**VAL**

A function which converts a character string representing a number into numeric form.

```plaintext
X = VAL(A$)
```

VAL makes the best sense it can of its argument. If the argument starts with numeric characters (with or without a preceding sign), VAL will work from left to right until it meets a non numeric character. It will then 'give up' and return what it has got so far. If it can't make any sense of its argument, it returns zero.

For example,

```plaintext
PRINT VAL("-123.45.67ABC")
```

would print

```
-123.45
```

and

```plaintext
PRINT VAL("A+123.45")
```

would print

```
0
```

VAL will NOT work with hexadecimal numbers. You must use EVAL to convert hexadecimal number strings.

If you want to force conversion to a 64-bit double, even when in *FLOAT40 mode, you can do so by adding a # (hash) suffix to the supplied string:

```plaintext
A# = VAL("1.234#")
nr# = VAL("7899876553.#")
```
Note that you should use this facility only when you are certain a numeric co-
processor is available, since it bypasses the normal check carried out when
*FLOAT64 is executed.

Syntax

\[ \text{\texttt{\textless n-var\textgreater }= \text{VAL(\textless string\textgreater )}} \]

Associated Keywords

\{ STR$ , EVAL \}
A statement which takes a list of numeric arguments and sends their least-significant
bytes as characters to the VDU emulator.

A 16-bit value can be sent if the value is followed by a ';'. It is sent as a pair of
characters, least significant byte first.

It is very important that the correct number of bytes be sent, as appropriate for the
particular VDU command. If too few are sent, the 'remaining' bytes will be taken
from subsequent VDU or PRINT statements, sometimes with (at first sight)
extremely strange results. This problem can be circumvented by terminating the
VDU statement with the '|' character which ensures that sufficient bytes are sent.

VDU 8,8 :REM cursor left two places.
VDU &0A0D;&0A0D; :REM CRLF twice
VDU 23,1,0 | : REM disable cursor

The bytes sent using the VDU statement do not contribute to the value of COUNT, 
but may well change POS and VPOS.

You can use VDU to send individual characters directly to the screen without having
to use a PRINT statement. It offers a convenient way of sending 'control characters'.

In UTF-8 mode each character may require 1, 2 or 3 bytes depending on its position
within the Basic Multilingual Plane.

Syntax

VDU <numeric>{,|;<numeric>}[;][|]

Associated Keywords

CHR$, PRINT
VPOS

A function returning the row number containing the text cursor (caret). VPOS returns the position of the cursor with respect to the text viewport; row 0 is normally at the top, but this may be changed using VDU 23,16.

\[ Y = \text{VPOS} \]

You can use VPOS in conjunction with POS to return to the present position on the screen after printing a message somewhere else. The example below is a procedure for printing a 'status' message at line 23. The cursor is returned to its previous position after the message has been printed.

```plaintext
DEF PROC_message(message$)
LOCAL x,y
x=POS
y=VPOS
PRINT TAB(0,23) CHR$(7);message$;
PRINT TAB(x,y);
ENDPROC
```

Syntax

\[ <\text{n-var}> = \text{VPOS} \]

Associated Keywords

POS
**WAIT**

A statement causing the program to pause for the specified number of centiseconds (hundredths of a second).

```
WAIT 100
```

This causes the program to pause for approximately one second.

WAIT 0 results in only a short pause (typically about 1 millisecond), but does give other programs an opportunity to run; it can be useful in preventing your program consuming 100% of the processor's time.

WAIT without a parameter does nothing; it is provided for compatibility with Acorn's BASIC 5 (in which it waits for the next vertical-retrace event).

**Syntax**

```
WAIT [<numeric>]
```

**Associated Keywords**

```
INKEY, TIME
```
WHEN

A keyword which is part of the CASE ... ENDCASE clause. WHEN precedes the value (or values) against which the CASE variable is tested. WHEN must be the first item on the program line.

```plaintext
CASE die% OF
    WHEN 1,2,3 : bet$ = "lose"
    WHEN 4,5,6 : bet$ = "win"
    OTHERWISE bet$ = "cheat"
ENDCASE
```

Syntax

```
WHEN <numeric>{,<numeric>} : {<stmt>}
WHEN <string>{,<string>} : {<stmt>}
```

Associated Keywords

```
CASE , OF , OTHERWISE
```
WHILE

A statement which is the starting point of a WHILE...ENDWHILE loop.

The purpose of a WHILE...ENDWHILE loop is to make BBC BASIC for Windows repeat a set number of instructions while some condition is satisfied. The difference between a WHILE...ENDWHILE loop and a REPEAT ... UNTIL loop is that the instructions within a REPEAT...UNTIL loop are always executed at least once (the test for completion is performed at the end of the loop) whereas the instructions within a WHILE...ENDWHILE loop are not executed at all if the condition is initially FALSE (the test is performed at the start of the loop).

```
WHILE LEFT$(A$,1)=" 
    A$=MID$(A$,2) : REM Remove leading spaces
ENDWHILE
```

You must not exit a WHILE...ENDWHILE loop with a GOTO (see the sub-section on Program Flow Control for more details). You can force a premature end to the loop with the EXIT WHILE statement.

WHILE...ENDWHILE loops may be nested.

Syntax

```
WHILE <t-cond>
```

Associated Keywords

ENDWHILE, EXIT, REPEAT
**WIDTH**

A statement controlling the overall field width for text output.

```
WIDTH 80
```

If the specified width is zero (the initial value) the interpreter will not attempt to control the overall field width.

WIDTH n will cause the interpreter to force a new line after n MOD 256 characters have been written; it also affects output to the printer.

WIDTH may not have the desired effect if a proportional-spaced font is selected, or when UTF-8 mode is enabled.

WIDTH can also be used as a function. It returns the current field width.

**BBC BASIC for Windows version 6.10a or later only**

WIDTH may also be used to measure the width, in BBC BASIC graphics units, of a string as it would be displayed in the main output window using the current font and character set (ANSI or UTF-8):

```
w% = WIDTH("Hello world!")
```

Note that because of kerning, and especially in the case of italic text, the returned value may not be precise.

**Syntax**

```
WIDTH <numeric>
<n-var> = WIDTH
<n-var> = WIDTH(<string>)
```

**Associated Keywords**

COUNT
Introduction to VDU commands

On the BBC Microcomputer and Acorn Archimedes the VDU commands perform a number of important actions, primarily involving the screen display and graphics. Although not strictly part of BASIC, most of these commands have been emulated by BBC BASIC for Windows.

The VDU statement takes a list of numeric arguments (constants or variables) and sends their least significant bytes as characters to the currently selected output stream.

A 16 bit value (word) can be sent if the value is followed by a semi-colon. It is sent as a pair of characters, the least significant byte first.
# VDU code summary

<table>
<thead>
<tr>
<th>Number</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Null - it does nothing.</td>
</tr>
<tr>
<td>1</td>
<td>Send the next character to the printer ONLY.</td>
</tr>
<tr>
<td>2</td>
<td>Enable the printer.</td>
</tr>
<tr>
<td>3</td>
<td>Disable the printer.</td>
</tr>
<tr>
<td>4</td>
<td>Write text at the text cursor position.</td>
</tr>
<tr>
<td>5</td>
<td>Write text at the graphics cursor position.</td>
</tr>
<tr>
<td>6</td>
<td>Enable output to the screen.</td>
</tr>
<tr>
<td>7</td>
<td>Bell - make a short 'beep'.</td>
</tr>
<tr>
<td>8</td>
<td>Move the text cursor backwards one character.</td>
</tr>
<tr>
<td>9</td>
<td>Move the text cursor forwards one character.</td>
</tr>
<tr>
<td>10</td>
<td>Move the text cursor down one line.</td>
</tr>
<tr>
<td>11</td>
<td>Move the text cursor up one line.</td>
</tr>
<tr>
<td>12</td>
<td>Clear the text area - identical to CLS.</td>
</tr>
<tr>
<td>13</td>
<td>Move the text cursor to the start of the current line.</td>
</tr>
<tr>
<td>14</td>
<td>Enable the auto-paging mode.</td>
</tr>
<tr>
<td>15</td>
<td>Disable the auto-paging mode.</td>
</tr>
<tr>
<td>16</td>
<td>Clear the graphics area - identical to CLG.</td>
</tr>
<tr>
<td>17</td>
<td>Define a text colour - identical to COLOUR.</td>
</tr>
<tr>
<td>18</td>
<td>Define a graphics colour - identical to GCOL.</td>
</tr>
<tr>
<td>19</td>
<td>Select a colour palette.</td>
</tr>
<tr>
<td>20</td>
<td>Restore the default logical colours.</td>
</tr>
<tr>
<td>21</td>
<td>Disable output to the screen.</td>
</tr>
<tr>
<td>22</td>
<td>Select the screen mode - identical to MODE.</td>
</tr>
<tr>
<td>23</td>
<td>Create user-defined characters and screen modes, etc.</td>
</tr>
<tr>
<td>24</td>
<td>Define a graphics viewport.</td>
</tr>
<tr>
<td>25</td>
<td>Identical to PLOT.</td>
</tr>
<tr>
<td>26</td>
<td>Restore the default text and graphics viewports.</td>
</tr>
<tr>
<td>27</td>
<td>Send the next character to the screen.</td>
</tr>
</tbody>
</table>
28 Define a text viewport.
29 Set the graphics origin - identical to ORIGIN.
30 Home the text cursor to the top left of the screen.
31 Move the text cursor - identical to TAB(x,y).
127 Backspace and delete.
VDU 0

Does nothing. In other words, it is ignored.
VDU 1

Provided the printer has been enabled (with VDU 2), the next character (byte) is sent to the printer driver and not to the screen. If the printer is not enabled, the character is discarded. Any 8-bit value (0 to 255) can be sent.

VDU 1 provides a simple way of sending single characters to the printer driver. It works whether the VDU has been disabled with the VDU 21 command or not. For example VDU 1,12 will send a 'form feed' and force the current page to be printed.

Note that you should not attempt to send printer-specific control characters (e.g. for setting the font, style etc.) as this won't work. BBC BASIC for Windows uses the Windows™ printer drivers and changes to the font or style must be made with the *PRINTERFONT command.

VDU 1,12
VDU 2

VDU 2 enables the printer. It causes all subsequent text output to be sent to both the screen and the printer.

The following control characters are acted upon by the printer: cursor left (VDU 8), cursor right (VDU 9), cursor down (VDU 10), cursor up (VDU 11), form feed (VDU 12), carriage return (VDU 13) and home (VDU 30).

Bytes which are parameters for VDU commands are not sent to the printer. For example,

VDU 17,65

does not send 'A' (CHR$65) to the printer.

If you need to line up columns on the printer using TAB(X) ensure you select a monospaced (fixed-pitch) font, for example:

*PRINTERFONT Courier New,12

Note that BBC BASIC for Windows assumes that the printer is a page printer. That is, output will not be printed until the current page is complete (i.e. a CLS or VDU 12 was sent, or the BASIC program terminates). This ensures that printer output is not mixed up with that from other programs running at the same time. You can force the current page to print using:

VDU 2,1,12,3
VDU 3

VDU 3 disables the printer. It cancels the effect of VDU 2 so that subsequent output to the screen is not also sent to the printer. If you want to force the page to be printed immediately, send a form-feed (VDU 12 ) before the VDU 3.
VDU 4

VDU 4 causes text to be written at the text cursor position in the normal way. This is the default mode.

Characters can be positioned only at text coordinates within the text viewport; the text viewport scrolls as necessary. Characters are written using the current text foreground colour on an opaque background of the current text background colour (see COLOUR ).
VDU 5

VDU 5 causes text to be written at the graphics cursor position. It works in all modes except MODE 7.

Characters may be positioned at any graphics coordinate within the graphics viewport. The top left corner of the character cell is the reference point. Characters are clipped to the limits of the graphics viewport if necessary. No scrolling takes place.

Characters are plotted using the current graphics foreground colour and in the current graphics foreground plotting mode (see GCOL).

The character background is 'transparent', i.e. it does not change what was previously displayed underneath. VDU 127 (DEL) is an exception; it backspaces and deletes just as it does in the normal text mode, using the current background GCOL rules and colour.
VDU 6

VDU 6 enables output to the VDU screen. It cancels the effect of VDU 21.

VDU 6
VDU 7

VDU 7 causes a short audible 'beep'.

VDU 7
VDU 8

VDU 8 moves the text cursor one character in the negative 'X' direction (normally left, but this can be changed using VDU 23,16 ). By default, if the cursor was at the start of a line, it moves to the end of the previous line (right edge of the text viewport). If it was also at the top line of the text viewport, the viewport scrolls down (except in the VDU 5 mode). The cursor is constrained to remain within the text viewport.
VDU 9

VDU 9 moves the text cursor one character in the positive 'X' direction (normally right, but this can be changed using VDU 23,16 ). By default, if the cursor was at the end of a line, it moves to the start of the next line (left edge of the text viewport). If the cursor was also at the bottom of the text viewport, the viewport scrolls up (except in the VDU 5 mode). The cursor is constrained to remain within the text viewport.
VDU 10

VDU 10 moves the text cursor one line in the positive 'Y' direction (normally down, but this can be changed using VDU 23,16 ). By default, if the cursor was on the bottom line of the text viewport, the viewport scrolls up (except in VDU 5 mode). Scrolling is paused if <Ctrl> and <Shift> are held down together. The cursor is constrained to remain within the text viewport.
VDU 11

VDU 11 moves the text cursor one line in the negative 'Y' direction (normally up, but this can be changed using VDU 23,16 ). By default, if the cursor was on the top line of the text viewport, the viewport scrolls down (except in VDU 5 mode). The cursor is constrained to remain within the text viewport.
VDU 12

VDU 12 clears the text viewport to the current text background colour and moves the text cursor to column 0, row 0 (normally the top-left corner of the text viewport, but this can be changed using VDU 23,16 ); it is identical to CLS . If sent to the printer (e.g. using VDU 2,1,12,3) it causes the current page to be printed.
VDU 13

VDU 13 moves the text cursor to column 0 within the current row (normally the left edge of the text viewport, but this can be changed using VDU 23,16 ). The cursor is constrained to remain within the text viewport.
VDU 14

VDU 14 enables auto-paging mode. Scrolling will stop after each page. When the text viewport has been filled, output is paused until <Shift> is pressed.
VDU 15

VDU 15 disables auto-paging mode. This is the default condition.
VDU 16

VDU 16 clears the graphics viewport using the current background GCOL action and colour. It does not move the graphics cursor. VDU 16 is identical to CLG. VDU 16 does nothing in MODE 7.
VDU 17

VDU 17 is identical to COLOUR. The next byte determines the text foreground or background colour. See COLOUR for more details. The example below sets the text foreground colour to red (assuming that VDU 19 has not been used to change the default logical to physical colour assignments).

VDU 17,1
**VDU 18**

VDU 18 is identical to GCOL. It takes the next two characters as the plotting mode and graphics colour respectively. Thus,

\[
\text{VDU 18,mode,colour}
\]

and

\[
\text{GCOL mode,colour}
\]

have the same effect. See the Graphics and colours section for more details.
VDU 19

VDU 19 sets the colour palette, i.e. the mapping between logical colours and physical colours.

VDU 19 takes the next five characters as parameters. The first parameter is the logical colour to be set and the remaining parameters determine the physical colour that it is to be set to.

If the second parameter is in the range 0 to 15 this determines the physical colour as an offset into the physical palette, in this case the remaining three parameters are ignored.

VDU 19,logical,physical,0,0,0

If the second parameter is -1 (or 255) the physical colour is determined by interpreting the remaining three parameters as red, green and blue values in the range 0 to 63, where 0 is none and 63 is maximum.

VDU 19,logical,-1,r,g,b

If the second parameter is 16 the physical colour is determined by interpreting the remaining three parameters as red, green and blue values in the range 0 to 255, where 0 is none and 255 is maximum.

VDU 19,logical,16,R,G,B
VDU 20

The VDU 20 command performs two distinct actions:

- It sets the text and graphics foreground and background to their default logical colours and plotting modes. This action is the same as issuing the commands:

  ```
  COLOUR 7
  COLOUR 128
  GCOL 0,7
  GCOL 0,128
  ```

- It sets up the default palette. This is equivalent to the program segment:

  ```
  FOR N = 0 TO 15
    VDU 19,N,N,0,0,0
  NEXT N
  ```
VDU 21

VDU 21 disables the VDU drivers until a VDU 6 is received. All VDU commands except 1 and 6 are ignored. If the printer is enabled, VDU commands 8 to 13 will still be sent to the printer.
VDU 22

VDU 22 is identical to MODE, except that MODE zeros the value of COUNT whereas VDU 22 does not. The mode is set according to the value of the byte following the VDU 22 command. The example below sets mode 3.

```
VDU 22, 3
```

VDU 22 also resets all the screen driver variables (colour, palette, viewports, cursor position, graphics origin, etc). In particular, VDU 22 performs the actions of VDU 4, VDU 12, VDU 15, VDU 20 and VDU 26.

See the Graphics and colours section and the keyword MODE for further details.

```
VDU 22,mode
```
**VDU 23**

VDU 23 is used for several different purposes: to create user-defined characters, to control the appearance of the text cursor (caret), to scroll the display, to select a user-defined display mode, to determine line thickness and to control the way the text cursor moves.

**User-defined characters**

Characters from &20 to &FF (32 to 255) may be programmed using the VDU 23 command. The programmable characters are not available in MODE 7.

The format of the VDU 23 command is:

```
VDU 23, char_no, r1, r2, r3, r4, r5, r6, r7, r8
```

'Char_no' is the character number to be programmed and 'r1' to 'r8' are the values of the bytes which define the character, top to bottom. For example, the character illustrated below is defined as character 130 by the following VDU 23 command.

```
VDU 23, 130, 32, 98, 226, 62, 62, 35, 33, 33
```

User defined characters occupy a 'cell' which is 8 dots wide by 8 dots high. The number of rows of user defined text which may be displayed on the screen depends on the vertical resolution of the display mode in use.

Note that although the current character font can be changed to any size using the *FONT command, user-defined characters are always the same size in a particular display MODE.

**Text cursor control**
The text cursor (caret) may be disabled and enabled using the VDU 23 command as shown below. The first is equivalent to the BASIC statement OFF and the second to the statement ON.

```
VDU 23,1,0;0;0;0; : REM Disable cursor
VDU 23,1,1;0;0;0; : REM Enable cursor
```

The shape of the text cursor can be controlled (to some degree) by using VDU 23 to set the start line and end line of the cursor 'block'; this affects its height and vertical position. The 'start line' and 'end line' are controlled by the following commands:

```
VDU 23,0,10,s,0;0;0; : REM Set start line
VDU 23,0,11,e,0;0;0; : REM Set end line
```

For example, the text cursor is normally a flashing underline in 'insert' mode and a flashing block in 'overtype' mode. To turn it from an underline into a block without changing to 'overtype' mode you can use the following command:

```
VDU 23,0,10;0;0;0;
```

This facility should be used with care.

The width of the text cursor can also be set:

```
VDU 23,0,18,w,0;0;0; : REM Set cursor width
```

This sets the width of the cursor to \( w \) pixels. If \( w \) is zero (the default) the cursor is set to the average character width of the current font.

For example, to set the cursor to a vertical line two pixels wide use the following code:

```
VDU 23,0,18,2,0;0;0;23,0,10;0;0;0;
```

**Scrolling the window**
The contents of BASIC's output window (or just the current text viewport) can be scrolled, by one character width or height, using VDU 23,7:

$$\text{VDU 23,7}, m, d, 0; 0; 0;$$

where m and d are as follows:

- m = 0 scroll just the text viewport
- m = 1 scroll BASIC's entire output window
- d = 0 scroll right one character
- d = 1 scroll left one character
- d = 2 scroll down one row
- d = 3 scroll up one row

**User-defined modes**

Although the MODE statement gives a wide choice of different display modes, there may be circumstances when a mode with different characteristics is required. The VDU 23 command can be used to select a user-defined mode as follows:

$$\text{VDU 23,22, width; height; charx, chary, ncols, charset}$$

where:

- width = window width in pixels (e.g. 640)
- height = window height in pixels (e.g. 512)
- charx = character width in pixels (e.g. 8)
- chary = character height in pixels (e.g. 16)
- ncols = number of colours (e.g. 16)
- charset = 0 for ANSI, 1 for OEM (IBM PC), 8 for UTF-8

Note that the width and height values must be followed by semicolons. You should choose the width and height values to be integer multiples of the charx and chary values respectively. For example, you can select a mode which has 10 (large!) characters per line and 8 lines of text as follows:

$$\text{VDU 23,22, 640; 512; 64, 64, 16, 0}$$
The maximum values for **width** and **height** are 1920 and 1440 respectively.

By default the background colour is black and the text colour is white. To select a mode with a white background and black text add 128 to the **charset** value.

If the **charset** value is set to 8, text output to the screen or printer is assumed to be in UTF-8 format. This gives access to the complete Basic Multilingual Plane (including alphabets like Greek, Cyrillic, Arabic, Hebrew etc.) whilst remaining compatible with ASCII. To utilise this mode you should select a Unicode font after the VDU 23 command:

VDU 23,22,640;512;8,16,16,8
*FONT Arial,16

**Line thickness**

Straight lines and outline shapes, plotted using VDU 25 (or, equivalently, using **DRAW**, **LINE**, **CIRCLE**, **ELLIPSE**, **RECTANGLE** etc.), are normally one pixel thick. The thickness can be changed to a different value as follows:

VDU 23,23,t;0;0;0;
or
VDU 23,23,t|

where *t* is the required line thickness in pixels. Note that only **solid** lines can have their thickness changed, not dotted or dashed lines.

**Cursor movement control**

The way the text cursor (caret) moves after a character has been output is determined by the state of eight **flag bits**; both screen and printer output are affected. One or more of the flag bits can be modified as follows:

VDU 23,16,x,y,0;0;0;

where the existing 8-bit flags byte is first **ANDed** with *y* and then exclusive-**ORed** with *x*. To set the entire flags byte to the value *f* you can do:
The flag bits control the cursor movement as follows (bit 7 must be zero):

<table>
<thead>
<tr>
<th>Flag bit</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 6 = 0</td>
<td>The text cursor is constrained to remain within the viewport/page (e.g. by wrapping or scrolling).</td>
</tr>
<tr>
<td>Bit 6 = 1</td>
<td>The text cursor can move beyond the edge of the viewport/page (except in VDU 4 mode).</td>
</tr>
<tr>
<td>Bit 5 = 0</td>
<td>The text cursor moves after a character is output.</td>
</tr>
<tr>
<td>Bit 5 = 1</td>
<td>The text cursor does not move after a character is output.</td>
</tr>
<tr>
<td>Bit 4 = 0</td>
<td>If appropriate the text viewport scrolls, or the printer page is ejected.</td>
</tr>
<tr>
<td>Bit 4 = 1</td>
<td>The text cursor wraps to the opposite edge of the viewport/page.</td>
</tr>
<tr>
<td>Bit 3 = 0</td>
<td>The 'X' direction is horizontal and the 'Y' direction is vertical.</td>
</tr>
<tr>
<td>Bit 3 = 1</td>
<td>The 'X' direction is vertical and the 'Y' direction is horizontal.</td>
</tr>
<tr>
<td>Bit 2 = 0</td>
<td>The 'positive' vertical direction is downwards.</td>
</tr>
<tr>
<td>Bit 2 = 1</td>
<td>The 'positive' vertical direction is upwards.</td>
</tr>
<tr>
<td>Bit 1 = 0</td>
<td>The 'positive' horizontal direction is rightwards.</td>
</tr>
<tr>
<td>Bit 1 = 1</td>
<td>The 'positive' horizontal direction is leftwards.</td>
</tr>
<tr>
<td>Bit 0 = 0</td>
<td>In VDU 4 mode, 'pending scroll' is enabled.</td>
</tr>
<tr>
<td>Bit 0 = 1</td>
<td>In VDU 4 mode, 'pending CRLF' is enabled.</td>
</tr>
</tbody>
</table>

Bits 1, 2 and 3 affect the column and row addressing used by TAB(X,Y), POS and VPOS. The default value of all the bits is zero.

In 'pending scroll' mode, if a character would normally cause the text viewport to scroll (thus causing the top line to be lost) it will not do so until just before the next character is output. If the cursor is repositioned (e.g. using VDU 13, VDU 30 or VDU 31) before the next character is output, no scroll will take place.

In 'pending CRLF' mode, if a character would normally cause a CRLF action (the text cursor wrapping to the beginning of the next line) it will not do so until just before the next character is output. If the cursor is repositioned before the next
character is output, no CRLF will take place.
**VDU 24**

In all modes except MODE 7, VDU 24 defines a graphics viewport. The following four words (pairs of bytes) are the X & Y coordinates of the bottom left corner of the viewport and the X & Y coordinates of the top right corner of the viewport, in that order. The coordinates are with respect to the current graphics origin.

It is particularly easy to select invalid viewport limits if the graphics origin has been moved. It is advisable, therefore, to precede a VDU 24 command with a VDU 29,0;0; command or ORIGIN 0,0 statement to reset the graphics origin.

The following example defines a graphics viewport with the bottom left corner at 200,100 and the top right corner at 500,300.

```plaintext
VDU 24,200;100;500;300;
```

```plaintext
VDU 24,left;bottom;right;top;
```

*Be very careful not to omit the final semicolon.*
VDU 25

VDU 25 is identical to the PLOT statement. See the Graphics and colours section or the keyword PLOT for more details.

\[ VDU \ 25, \text{mode, x\_coord; y\_coord;} \]

The following example draws a line in the current foreground colour to the point 350,525:

\[ VDU \ 25, 5, 350; 525; \]
VDU 26

VDU 26 resets the text and graphics viewports to their default positions (filling the whole screen), homes the text cursor to the top left of the screen (0,0), resets the graphics origin to the bottom left of the screen (0,0) and moves the 'current point' to the graphics origin.

*Important note:* VDU 26 resets the text and graphics viewports to the current size of BASIC's *screen* (output window), *not* to the default size appropriate to the current MODE. In the event that BASIC's output window has been resized by the user (by dragging a corner or side) or has a reduced size because the display resolution is inadequate for the selected MODE, VDU 26 may alter the relative alignment of text and graphics. If this could be a problem, avoid the use of VDU 26 (or make sure the display resolution is sufficient and prevent the user resizing the window).
VDU 27

VDU 27 sends the next byte to the screen without interpreting it as a control character. It allows graphics characters corresponding to VDU 0 to VDU 31 and VDU 127 to be displayed. It acts for characters sent to the screen in a similar manner to the way VDU 1 does for characters sent to the printer.

VDU 27,127
VDU 28

VDU 28 defines a text viewport. The following four bytes are the X & Y coordinates of the bottom left corner of the viewport and the X & Y coordinates of the top right corner of the viewport, in that order. The coordinates are with respect to the text origin (top left) and are measured in 'character positions'.

If the text cursor is outside the new viewport, it is moved to the new home position (top left of the viewport). If it is inside the new viewport, it is not moved.

The following example defines a text viewport with the bottom left corner at X=0, Y=15 and the top right corner at X=30, Y=3.

```
VDU 28,0,15,30,3

VDU 28,left,bottom,right,top
```
VDU 29

VDU 29 moves the graphics origin to the coordinates specified by the following two words (pairs of bytes). The first word specifies the X coordinate of the new origin and the second specifies the Y coordinate. Subsequent graphics commands operate with respect to this origin. VDU 29 has the same effect as the ORIGIN statement.

The following example sets the graphics origin to 640,400:

```
VDU 29,640;400;
ORIGIN 640,400
```
**VDU 30**

In VDU 4 mode, VDU 30 homes the text cursor to column 0, row 0 (normally the top-left corner of the text viewport, but this can be changed using VDU 23,16 ). In VDU 5 mode, VDU 30 homes the graphics cursor to the 'top left' corner of the graphics viewport.
VDU 31

VDU 31 is identical to PRINT TAB(x,y) . It positions the text cursor according to the following two bytes, the first determining the column number and the second the row number. Normally column 0 is the left of the text viewport and row 0 is the top of the text viewport, but this can be changed using VDU 23,16 . The example below positions the text cursor in column 15 and row 10.

VDU 31,15,10

See the keyword TAB for further details.

VDU 31,col,row
VDU 127

Deletes the character to the left of the cursor and backspaces the cursor to this position. In VDU 4 mode this has the same effect as the sequence backspace-space-backspace (VDU 8,32,8).
Introduction to star commands

On the BBC Microcomputer and Acorn Archimedes, star (*) commands provide access to the 'operating system'. These commands can either be 'resident' (i.e. permanently part of the Operating System) or 'transient' (i.e. loaded when required). Windows™ does not have 'resident' commands in quite the same way, so BBC BASIC for Windows implements its own resident commands as part of the interpreter. However, 'transient' or 'external' commands can still be activated via the star commands (or OSCLI ). When a star command is issued, BBC BASIC for Windows first checks to see if it is one of its built-in resident commands. If not it is passed to Windows™ for action.

File specifiers

Many of the 'star' commands are concerned with files; whenever a filename or file specifier is used it must comply with the standard Windows™ naming conventions:

```
[drive:][\][path]\filename[.ext]
```

- **drive:** The single letter name of the drive where the file will be found. The colon is mandatory. If the drive name is omitted, the current drive is assumed.

- **path** The list of directories and sub-directories (folders) which must be followed in order to find the specified file. The names of each directory in the path must be separated by the backslash character. If the path is omitted, the current directory on the specified drive is assumed.

- **filename** The name of the file. If this contains spaces the entire file specifier should be enclosed in quotation marks (see below).

- **ext** The optional extension of the file. If an extension is used it must be separated from the filename by a full-stop. If the extension is omitted, .BBC is (usually) assumed.

File and directory names are not case-sensitive and generally do not need to be
enclosed in quotes. However if a file or directory name includes any spaces (or punctuation characters) you must enclose it in quotes; unmatched quotes will cause a 'Bad string ' error. If you need to use the OSCLI statement there are two main ways of incorporating quotation marks, with CHR$(34) or using the 'escape' sequence "". For example both the following will work:

```
OSCLI "CD "+CHR$(34)+directory$+CHR$(34)
OSCLI "CD ""+directory$+""
```

When an 'ambiguous' file specifier is needed the standard wildcard characters (? and *) may be used.

Control characters, DEL, quotation marks and other special characters may be incorporated in filenames etc. by using the 'escape' character '|', however the host operating system (e.g. Windows™) may not accept them:

```
|A gives ^A (control A).
|? gives Del.
|" gives the quotation mark ".
|| gives the escape character |.
|! sets bit 7 of the following character.
```

**Accessing star commands**

The star commands may be accessed directly or via the OSCLI statement. OSCLI must be used when all or part of the star command needs to be determined at run-time (i.e. is a variable rather than a constant). The two examples below both access the *HELP command.

```
*HELP
OSCLI ("HELP")
```

A star command must be the last (or only) command on a program line and its argument may not be a variable. Unlike keywords, star commands are not case-sensitive (and are not affected by *LOWERCASE ).

**Similarly named star commands**
If you wish to execute a Windows™ command with the same name as a resident command, precede it with a second star. For example,

*COPY

will execute *BBC BASIC for Windows* 'resident file copy command but

**COPY

will pass the command to Windows™.

**Errors in star commands**

If a star command is mistyped, or does not exist, it will be interpreted as an 'external' command and passed to Windows™ for execution. If it is not the name of a valid Windows™ GUI or Console command, the message 'Bad command or file name' will be reported in a console window. However this may appear on the screen very transitorily and it may not be apparent what has happened. In addition this error cannot be detected or trapped by *BBC BASIC for Windows*, and execution of the BASIC program will continue.

For this reason, the *FX (with the exception of *FX 15 and *FX 21) and *TV commands are trapped and ignored. These two star commands are commonly found in programs for the BBC Micro, but they have no counterpart in Windows™.
## Star command summary

**Command** | **Function**
--- | ---
*BYE | exit the BASIC program and close its window
*CHDIR (*CD) | display or change the current directory
*COPY | copy a file
*DELETE (*DEL) | delete a file
*DIR (*.) | list the directory contents
*DISPLAY | display a Windows Bitmap (BMP) file
*EGA | emulate an EGA or CGA display
*ERASE (*ERA) | delete a file
*ESC | enable or disable the Escape key
*EXEC | accept console input from a file
*FLOAT | select floating-point numeric precision
*FONT | select a screen font and style
*FX | flush keyboard and/or sound buffers
*GSAVE | synonymous with *SCREENSAVE
*HARDCOPY | copy graphics to the printer
*HELP | display version information
*HEX | control hexadecimal conversion and shift operators
*INPUT | redirect console input
*KEY | program a function or special key
*LIST | display the contents of a program file
*LOAD | load a file into memory
*LOCK | set a file to read-only status
*LOWERCASE | enable or disable lower-case keywords
*MARGINS | set the printer page margins
*MKDIR (*MD) | create a sub-directory (folder)
*MDISPLAY | display a BMP image stored in memory
*NOEGA | cancel the effect of *EGA
*OUTPUT (*OPT) | redirect console output
*PLAY     play a MIDI file
*PRINTER   select a printer
*PRINTERFONT   select a printer font and style
*QUIT      exit the BASIC program and close its window
*REFRESH   control when the screen is refreshed
*RENAME (*REN)   rename a file
*RMDIR (*RD)  remove (delete) a directory
*RUN       execute a Windows GUI or Console command
*SAVE      save memory to a file
*SCREENSAVE save the output window as a BMP file
*SPOOL     copy console output to a file
*SPOOLON   append console output to an existing file
*SYS       enable trapping of additional system messages
*TEMPO     control the speed of the SOUND statement
*TIMER     control the periodicity of the ON TIME interrupt
*TV        accepted but ignored
*TYPE      display the contents of a text file
*UNLOCK    set a file to read-write status
*|         a comment
*BYE

Exit the interpreter and return to the calling program. If the BASIC program is running in the interactive environment, the program's output window closes and control is returned to the command and editing window.

If the BASIC program is running stand-alone (either using the BBCWRUN utility or having been converted to an executable with the Compile command) the program quits and returns control to Windows™

*BYE is synonymous with the *QUIT command and the QUIT statement.

Syntax:

*BYE
**CHDIR (*CD)**

Change the current working directory (folder) to a different drive and/or path. The syntax is similar to the CHDIR command available at a Command Prompt.

*CHDIR BBCBASIC\DOC
*CD A:\LETTERS\RATES
*CD "Program Files"
OSCLI "CD """"+directory$++""

*CD followed only by a backslash will change the current working directory to the root (top level) directory of the current drive. *CD followed only by a drive name will change the current working directory to the root directory of that drive.

*CHDIR \\
*CD A:

*CD on its own will display the current working directory (folder). If your program needs to know the current directory you can either use *CD and SPOOL the output to a file, or you can use the GetCurrentDirectory API function:

```
DEF FNcurrentdirectory
DIM cd% LOCAL 255
SYS "GetCurrentDirectory", 256, cd%
= $$cd%
```

**Syntax:**

```
*CD [d:]\[\]\[pth]
*CHDIR [d:]\[\]\[pth]
```
*COPY

Copy a file. The syntax is similar to the COPY command available at a Command Prompt, except that the filename extension defaults to .BBC if omitted. To specify a filename with no extension, add a final full-stop.

*COPY srcfile.dat destfile.dat
OSCLI "COPY "+sourcefile$+" "+destfile$+"

If the *summed* lengths of the source and destination filenames exceeds 254 characters a String too long error will result. In that case you can use the Windows™ API instead (the extensions must be included explicitly):

SYS "CopyFile", sourcefile$, destfile$, 0

The above commands will copy only one file at a time; wildcards are not permitted in the file specifiers. If you wish to include wildcards in order to copy more than one file, you can use the **COPY command which will be executed by Windows™. Alternatively you can use the following routine based on the FindFirstFile and FindNextFile Windows™ API functions:

DEF PROC copyfiles(src$, dst$, afsp$)
LOCAL dir%, buf%, sh%, res%, fpt%, num%
DIM dir% LOCAL 317, buf% LOCAL 260
IF RIGHT$(src$,1) <> "\" src$ += "\"
IF RIGHT$(dst$,1) <> "\" dst$ += "\"
SYS "FindFirstFile", src$+afsp$, dir% TO sh%
IF sh% <> -1 THEN
REPEAT
SYS "GetFullPathName", dir%+44, 260, buf%, ^fpt% TO num%
buf%?num% = 13
IF (!dir% AND 16) = 0 THEN
SYS "CopyFile", src$+$fpt%, dst$+$fpt%, 0 TO res%
IF res% = 0 PRINT "File copy failed"
ENDIF
SYS "FindNextFile", sh%, dir% TO res%
UNTIL res% = 0
SYS "FindClose", sh%
ENDIF
ENDPROC
The parameters are the source and destination directories and a file specification containing wildcards (* or ?).

**Syntax:**

```
*COPY ufsp ufsp
```
**DELETE (**DEL**)

Delete the specified disk file. The syntax is similar to the DEL command available at a Command Prompt except that the extension defaults to .BBC if it is omitted. To specify a filename with no extension, add a final full-stop.

*DEL GAME1.DTA
OSCLI "DEL "**file$***"

This command will delete only one file at a time; a wildcard is not permitted in the file specifier. If you wish to include wildcards in order to delete more than one file, you can use the **DEL command which will be executed by Windows™. Alternatively you can write your own routine based on the **FindFirstFile and **FindNextFile Windows™ API functions .

**Syntax:**

*DEL ufsp
*DELETE ufsp
List the disk directory. The syntax is similar to the DIR command available at a Command Prompt except that the extension defaults to .BBC if it is omitted. To specify a filename with no extension, add a final full-stop.

*DIR List all .BBC files in the current directory (folder).
*DIR B:*.* List all files on disk B.
*.*.* List all the files in the current directory.

Syntax:

*DIR [afsp]
*DISPLAY

Display a Windows Bitmap image file, optionally specifying its width and height:

```
*DISPLAY bmpfile
*DISPLAY bmpfile x,y
*DISPLAY bmpfile x,y,w,h
OSCLI "DISPLAY """"+bmpfile$+""""+STR$x+"","+STR$y+"","+STR$w+"","+STR$h"
```

The specified position (x,y) corresponds to the bottom-left corner of the image, and is given in BBC BASIC for Windows graphics units. The size and shape of the image may optionally be specified, in which case the image is scaled as necessary; this may result in a loss of quality. If the width and height are omitted the image is displayed without scaling.

**BBC BASIC for Windows version 6.10a or later only**

An additional parameter may be specified, corresponding to a hexadecimal value that will be interpreted as a 'transparent' or 'key' colour; any pixels with this colour in the image will not be plotted and whatever was 'behind' will show through (requires Windows 2000 or later):

```
OSCLI "DISPLAY """"+bmpfile$+"""" x,y,w,h,k"
```

The key colour must be specified in whatever format the BMP file uses internally, so if it's a 24 bits-per-pixel RGB file specify a six-digit hexadecimal value in the format **rrggbb** (if the key colour is black specify it as **1000000**). If the file has a different pixel format specify it in whatever representation the file uses. In the special case of a 16 bits-per-pixel file ensure that you choose a key colour which has low RGB values, for example R=1, G=1, B=1 which corresponds to the hexadecimal value **0421** in a RGB555 file.

The quality of the displayed image will be affected by your current display settings (right-click on the desktop, select **Properties** then **Settings**). For best results ensure you are using High Colour (16-bit) or True Colour (24-bit), but see the Compatibility limitations section for notes on the effects of the different settings.

The position and size values must be **integers**; if necessary use the INT function to ensure there is no decimal point.
If the filename extension is omitted, .BMP is assumed. See also *MDISPLAY and *SCREENSAVE.

When the image is scaled, you may find that the quality is improved by preceding the *DISPLAY command with the following statement:

```plaintext
SYS "SetStretchBltMode", @memhdc%, 3
```

**Syntax:**

*DISPLAY ufsp [num,num[,num,num[,hex]]]
*EGA

This command is provided for BBC BASIC (86) compatibility. *EGA ON causes screen modes 0 to 15 to be mapped to the dimensions and number of colours they would have in BBC BASIC (86) when the *EGA command is used, i.e. modes of which an EGA adaptor and monitor are capable.

*EGA OFF causes screen modes 0 to 15 to be mapped to the dimensions and number of colours they would have in BBC BASIC (86) when the *EGA command is not used (or the *EGA OFF command is used), i.e. modes of which a CGA adaptor and monitor are capable.

BBC BASIC (86) programs which include a *EGA (or *EGA ON) command should therefore select an appropriate mode without needing to be modified. BBC BASIC (86) programs which don't include a *EGA command should be modified by adding a *EGA OFF command. This will cause BBC BASIC for Windows to select an appropriate mode, but will not affect the operation of the program in BBC BASIC (86).

If you need to cancel the effect of *EGA, use *NOEGA.

*EGA ON
*EGA OFF
*NOEGA

Syntax:

*EGA [ON|OFF]
*ERASE (*ERA)

This command is synonymous with *DELETE.

Syntax:

*ERASE ufsp
*ESC

Disable or enable the abort action of the ESCape key.

*ESC OFF
*ESC ON

After *ESC OFF the <Esc> key simply returns the ASCII code for ESCape (27). *ESC ON (or *ESC) restores the normal abort action of the <Esc> key.

Syntax:

*ESC [ON|OFF]
*EXEC

Accept console input from the specified file instead of from the keyboard. If the extension is omitted, .BBC is assumed. To specify a filename with no extension, add a final full-stop.

    *EXEC STARTUP.CMD
    OSCLI "EXEC """"+cmdfile$++++"

Once the entire file has been read, keyboard input is restored. *EXEC without a filename closes the file and restores keyboard input immediately.

Syntax:

    *EXEC [ufsp]
*FLOAT

Select the required floating-point numeric precision (40, 64 or 80); the default is *FLOAT 40:

*FLOAT 40
*FLOAT 64
*FLOAT 80

The *FLOAT command affects the precision with which floating-point values are stored and retrieved using floating-point indirection or in data files (PRINT# and INPUT# statements). When reading or modifying a data file you should always use the same *FLOAT mode as when it was originally written.

*FLOAT command additionally controls the precision of variant (suffixless) numeric variables; only *FLOAT40 (the default) and *FLOAT 64 are available. In *FLOAT40 mode variant (real) numbers have approximately a 9 significant-figure accuracy; in *FLOAT64 mode they have approximately a 15 significant-figure accuracy. *FLOAT64 mode (in fact, it may run more quickly because it uses the CPU's numeric data processor) but variant numeric variables and arrays occupy more memory.

*FLOAT command additionally controls the precision of variant (suffixless) numeric variables; only *FLOAT40 (the default) and *FLOAT 64 are available. In *FLOAT40 mode variant (real) numbers have approximately a 9 significant-figure accuracy; in *FLOAT64 mode they have approximately a 15 significant-figure accuracy. *FLOAT64 mode (in fact, it may run more quickly because it uses the CPU's numeric data processor) but variant numeric variables and arrays occupy more memory.

BBC BASIC for Windows version 5.95a or earlier only

Variant variables created in *FLOAT40 mode are independent of variables created in *FLOAT64 mode, and can co-exist. In the rare circumstance that you need to copy a 40-bit variable into a 64-bit variable, or vice versa, you can do that only in *FLOAT40 mode by adding a # suffix to the name of the 64-bit variable. One use for that facility is to read a data file written in a different *FLOAT mode:

*FLOAT 40
INPUT #file, A
A# = A
*FLOAT 64
PRINT A#

The above program segment reads a 40-bit value from a data file, then copies it into a 64-bit variable.
BBC BASIC for Windows version 5.95a or earlier only
Numeric constants are converted to 40-bit or 64-bit binary floating-point format according to which *FLOAT mode is currently selected. If you want to force conversion to a 64-bit double, even when in *FLOAT40 mode, you can do so by adding a # (hash) suffix:

```
A# = 1.234#
B# = PI#
PRINT PI#-PI
```
*FONT

Select the specified font (typeface) and size, and optionally select an attribute (bold, italic, underscore and, in BBC BASIC for Windows version 6.00a or later only, strikeout). The size should be specified in 'points'.

*FONT Arial,16
*FONT Courier New,20,U
*FONT Letter Gothic,12,BI

The required attribute(s) are specified using the letters B, I, U and (for strikeout, when available) Q. The underscore (U) and strikeout (Q) attributes are mutually-exclusive.

If the *FONT command is mistyped, or if the requested font is not installed, Windows™ will do its best to select a font similar to the one you requested. Unfortunately it doesn't always do this very well, and the result can be the selection of a 'symbols' font. If this happens, what is displayed won't make much sense! You can find out which font has been selected using the GetTextFace Windows API function.

If no typeface, size or style is specified, *FONT selects the System Fixed Font:

*FONT

Note that the size at which the font is displayed on the screen depends on the current dots per inch (dpi) setting. This is most commonly 96dpi, but may be 120dpi (or another value) if Large Fonts have been selected or when using some high-resolution displays. Ideally you should ensure that your program works satisfactorily with either setting.

Use the *FONT command with care, as selecting a font other than the default can cause unwanted side-effects. For example, if you select a proportional-spaced font and then use the INPUT statement, the input editing features provided will not work correctly. The <backspace> key will not correctly 'blank' (on the screen) the previous character entered, nor will the <cursor left> and <cursor right> keys have the desired effect. This problem can be circumvented by using the FNpsinput
function listed below; this implements most of the features of the INPUT statement but will work correctly with a proportional-spaced font:

```
DEF FNpsinput(X%,Y%,prompt$) : LOCAL A$,C%,K%
SYS "SetBkMode", @memhdc%, 1
VDU 23,0,18,2,0;0;0;23,0,10;0;0;0;
REPEAT : OFF
   PRINT TAB(X%,Y%) prompt$;A$;"    ";
   PRINT TAB(X%,Y%) prompt$;LEFT$(A$,C%);
   ON : REPEAT K% = INKEY(1) : UNTIL K%<>-1
      CASE K% OF
         WHEN 8,127: IF C% A$ = LEFT$(A$,C%-1) + MID$(A$,C%+1) : C% -= 1
         WHEN 13: 
         WHEN 136: IF C% C% -= 1
         WHEN 137: IF C% < LEN(A$) C% += 1
      OTHERWISE: A$ = LEFT$(A$,C%) + CHR$K% + MID$(A$,C%+1) : C% += 1
   ENDCASE
UNTIL K% = 13
PRINT
= A$
```

The function takes as parameters the column and row numbers (as would be needed by TAB(X,Y)) and a prompt string. It returns the string which was entered by the user.

**Syntax:**

```
*FONT [name,size[,style]]
```
*FX

With the exception of *FX 15 and *FX 21, all the *FX commands are ignored.

*FX 15,n and *FX21,n flush selected internal buffers as follows:

*FX 15,0 flushes all the internal buffers
*FX 15,1 flushes the currently selected input buffer
*FX 21,0 flushes the keyboard buffer
*FX 21,4 flushes the channel 0 SOUND buffer
*FX 21,5 flushes the channel 1 SOUND buffer
*FX 21,6 flushes the channel 2 SOUND buffer
*FX 21,7 flushes the channel 3 SOUND buffer

Syntax:

*FX num,num
*GSAVE

This command is synonymous with *SCREENSAVE.

Syntax:

*GSAVE ufsp [num, num, num, num]
*HARDCOPY

Copy graphics from the screen to the printer.

*HARDCOPY x, y, w, h, X, Y, W, H

A rectangular region of BASIC's output window is copied to the printer (assuming your printer supports bit-mapped graphics). The parameters are as follows:

x, y  The coordinates, in BBC BASIC graphics units, of the bottom-left corner of the region to be printed.

w, h  The width and height, in BBC BASIC graphics units, of the region to be printed.

X, Y  The offset, in millimetres, from the top-left corner of the printable area of the paper to the top-left corner of the printed graphics.

W, H  The width and height, in millimetres, of the printed graphics.

You can print more than one graphic on the same page by using *HARDCOPY multiple times with different offsets specified. Similarly you can combine graphics and text by using *HARDCOPY in conjunction with the VDU 2 and VDU 3 commands. When the entire contents of the page have been output, send a form-feed character (e.g. VDU 2,1,12,3 ) to cause it to be printed.

If you have increased the size of the graphics 'canvas' to be greater than the size of your output window (for example using the technique used in SCROLL.BBC ) then you can use *HARDCOPY to print the entire canvas, not just what is visible on the screen.

If you omit the destination parameters (X, Y, W, H) the selected region is automatically scaled and centered to fit the page. If you omit all the parameters, the entire output window is scaled and centered to fit the page.

Syntax:

*HARDCOPY [num, num, num, num[, num, num, num, num]]
Display the name and version number of *BBC BASIC for Windows*. This results in the following message (or something like it) being displayed:

BBC BASIC for Windows 5.91a

Syntax:
**HEX**

*BBC BASIC for Windows version 6.00a or later only*

Control whether the hexadecimal conversion (& and ~) and bit-shift (<< and >>) operators treat their operands as 32-bit or 64-bit integers. In the (default) **HEX 32** mode the operators work in a fashion which is compatible with earlier versions of BBC BASIC; the operands are treated as 32-bit signed integers. In **HEX 64** mode the operands are treated as 64-bit signed integers.

*HEX 32
HEX 64

For example consider the following program:

```plaintext
PRINT ~TRUE
PRINT &FFFFFFFF
PRINT 2000000000 << 1
```

In **HEX 32** mode the output is:

```
FFFFFFFF
-1
-294967296
```

In **HEX 64** mode the output is:

```
FFFFFFFFFFFFFFFF
4.2949673E9
4E9
```
*INPUT

Redirect console input to come from a communications port rather than from the local keyboard:

*INPUT 1
OSCLI "INPUT "+STR$channel%"

The number following *INPUT corresponds to the channel number returned by the OPENUP function used to open the required comms port; this will be in the range 1 to 4 (see the Serial I/O section for details). Since this is determined at run-time, you will normally need to use OSCLI to access this command.

*INPUT 0 restores normal (keyboard) console input.

The following special values provide improved support for console mode programs:

*INPUT 13  as *INPUT 1, except that 'non overlapped' I/O is used.
*INPUT 14  as *INPUT 2, except that 'non overlapped' I/O is used.
*INPUT 15  as *INPUT 3, except that 'non overlapped' I/O is used.

Syntax:

*INPUT num
Redefine a function or cursor key to return the specified string.

```
*KEY 2 "This is key 2"
OSCLI "KEY "+STR$keynum%+" ""+keystr$+""
```

The 'key number' is from 1 to 24 as follows:

<table>
<thead>
<tr>
<th>No</th>
<th>Key</th>
<th>No</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>f1</td>
<td>12</td>
<td>←</td>
</tr>
<tr>
<td>2</td>
<td>f2</td>
<td>13</td>
<td>→</td>
</tr>
<tr>
<td>3</td>
<td>f3</td>
<td>14</td>
<td>↓</td>
</tr>
<tr>
<td>4</td>
<td>f4</td>
<td>15</td>
<td>↑</td>
</tr>
<tr>
<td>5</td>
<td>f5</td>
<td>16</td>
<td>Ctrl/* ← (or mouse wheel down)</td>
</tr>
<tr>
<td>6</td>
<td>f6</td>
<td>17</td>
<td>Ctrl/* → (or mouse wheel up)</td>
</tr>
<tr>
<td>7</td>
<td>f7</td>
<td>18</td>
<td>Home</td>
</tr>
<tr>
<td>8</td>
<td>f8</td>
<td>19</td>
<td>End</td>
</tr>
<tr>
<td>9</td>
<td>f9</td>
<td>20</td>
<td>PgUp</td>
</tr>
<tr>
<td>10</td>
<td>f10</td>
<td>21</td>
<td>PgDn</td>
</tr>
<tr>
<td>11</td>
<td>Shift/Tab (or f11)</td>
<td>22</td>
<td>Ins</td>
</tr>
<tr>
<td>24</td>
<td>Backspace</td>
<td>23</td>
<td>Del</td>
</tr>
</tbody>
</table>

The string may contain the 'escape' symbol '|' in order to include non-printing characters. For example, |M indicates CR (carriage-return), |? indicates DEL, || indicates the escape character itself and |! causes bit 7 of the following character to be set. If the string is enclosed in quotes (which is optional), |" allows the character " to be included in the string.

If there is insufficient room for the string, a 'Bad string ' error will occur and the key will be loaded with as much of the string as would fit.

*KEY num without a string will empty the specified key and return it to its normal code (see GET ).
Compatibility with some BBC Micro programs can be improved by redefining the *Backspace* key to return the code 127 (DEL) rather than 8 (BS). That can be achieved as follows:

```
*KEY 24 | ?
```

**Syntax:**

```
*KEY num [str]
```
*LIST

List the specified internal format (tokenised) program file to the screen. If the filename extension is omitted, .BBC is assumed. To specify a filename with no extension, add a final full-stop.

*LIST progfile
OSCLI "LIST "+progfile$

Note that no indentation or syntax colouring takes place.

You can use this command to display the contents of a program file saved in BBC BASIC (.BBC) format. You can pause the display either by holding down <Shift> and <Ctrl> simultaneously, or by first entering 'paged' mode by typing Ctrl/N (VDU 14). In 'paged' mode, the display pauses until <Shift> is pressed. To cancel 'paged' mode press Ctrl/O (VDU 15).

To display a program saved in ASCII text (.BAS) format use *TYPE.

Syntax:

*LIST ufsp
*LOAD

Load a specified file into memory at a specified address. The load address must always be specified. If the filename extension is omitted, .BBC is assumed. To specify a filename with no extension, add a final full-stop.

```
OSCLI "LOAD A:WOMBAT "+STR$-A%
OSCLI "LOAD """"+file$"""" "+STR$-address%
```

The load address must be given as a hexadecimal number. You must ensure that the specified address is valid and that there is sufficient room for the file to be loaded there. Generally, the address should have been allocated using the DIM statement. If you fail to take these precautions it is very likely that *LOAD will crash BBC BASIC for Windows !.

You can optionally specify the maximum number of bytes to be loaded (prefixed by a + sign) or, equivalently, the first address not to be loaded. The value must be specified as a hexadecimal number:

```
OSCLI "LOAD """"+file$"""" "+STR$-addr%" "+STR$-max%
OSCLI "LOAD """"+file$"""" "+STR$-addr%" "+STR$-(addr%+max%)
```

You must specify the maximum size when loading into an area of memory allocated by DIM LOCAL or a Windows™ API function .

**Syntax:**

*LOAD ufsp hexaddr
*LOAD ufsp hexaddr +hexsize
*LOAD ufsp hexaddr hextop*
*LOCK

Set the specified file to 'read-only' status. If the extension is omitted, .BBC is assumed. To specify a filename with no extension, add a final full-stop.

*LOCK WOMBAT.DAT
OSCLI "LOCK """"+file$+"""

Once a file has been made read-only, any attempt to write to it will result in the Access denied error.

Syntax:

*LOCK ufsp
*LOWERCASE

Allow BASIC keywords, hexadecimal numbers and numbers in 'exponent' (E) format to be accepted in lower-case:

*LOWERCASE ON

This affects the recognition of keywords by EVAL. For example if *LOWERCASE ON has been executed, EVAL("pi") will return the value of PI. Otherwise it would result in a No such variable error.

A word will only be recognised as a keyword if it is entirely capitals or entirely lower-case. A word consisting of a mixture of capitals and lower-case characters will never be mistaken for a keyword.

*LOWERCASE ON also allows lower-case hexadecimal numbers to be accepted (for example &abcd) and the use of a lower-case "e" in numbers entered in 'exponent' format (e.g. 1.2e-3).

To disable the acceptance of lower-case keywords etc. use *LOWERCASE OFF.

When entering statements or commands in Immediate mode the acceptance of lower-case keywords etc. is controlled by the Lowercase keywords item in the Options menu.

Syntax:

*LOWERCASE [ON|OFF]
*MARGINS

Set the printer page margins.

*MARGINS left, bottom, right, top

This sets the printer page margins to the specified values in millimetres. BASIC will not print closer to the edge of the paper than these values. Initially all four margins are set to 10 mm, but note that both the *PRINTER and *PRINTERFONT commands reset the margins to zero, not to 10 mm.

Syntax:

*MARGINS num, num, num, num
*MKDIR (*MD)

Create a new directory (folder) with the given name. The syntax is similar to the MKDIR command available at a Command Prompt:

```
*MD D:DATA
*MKDIR \BBCBASIC\TRIALS
OSCLI "MKDIR """"+directory$+"""""
```

If a directory with that name already exists, the File exists error will result.

**Syntax:**

```
*MD [d:][\]path
*MKDIR [d:][\]path
```
*MDISPLAY

Display a Windows Bitmap image stored in memory, optionally specifying its width and height:

```
OSCLI "MDISPLAY "+STR$~addr
OSCLI "MDISPLAY "+STR$~addr+" x,y"
OSCLI "MDISPLAY "+STR$~addr+" x,y,w,h"
```

The specified position (x,y) corresponds to the bottom-left corner of the image, and is given in BBC BASIC for Windows graphics units. The size and shape of the image may optionally be specified, in which case the image is scaled as necessary; this may result in a loss of quality. If the width and height are omitted the image is displayed without scaling.

**BBC BASIC for Windows version 6.10a or later only**

An additional parameter may be specified, corresponding to a hexadecimal value that will be interpreted as a 'transparent' or 'key' colour; any pixels with this colour in the image will not be plotted and whatever was 'behind' will show through (requires Windows 2000 or later):

```
OSCLI "MDISPLAY "+STR$~addr+" x,y,w,h,k"
```

The key colour must be specified in whatever format the bitmap data uses internally, so if it's a 24 bits-per-pixel bitmap specify a six-digit hexadecimal value in the format `rrggbb` (if the key colour is black specify it as `1000000`). If the bitmap has a different pixel format specify it in whatever representation the bitmap uses. In the special case of a 16 bits-per-pixel bitmap ensure that you choose a key colour which has low RGB values, for example R=1, G=1, B=1 which corresponds to the hexadecimal value `0421` in a RGB555 bitmap.

The quality of the displayed image will be affected by your current display settings (right-click on the desktop, select **Properties** then **Settings**). For best results ensure you are using High Colour (16-bit) or True Colour (24-bit), but see the Compatibility limitations section for notes on the effects of the different settings.

The position and size values must be **integers**; if necessary use the INT function to ensure there is no decimal point.
When the image is scaled, you may find that the quality is improved by preceding
the *MDISPLAY command with the following statement:

   SYS "SetStretchBltMode", @memhdc%, 3

*MDISPLAY is useful if you want to display the same image many times, since using *DISPLAY involves an overhead of reading the file each time. With
*MDISPLAY you can load the file once (using *LOAD ) and then display it many
times. See also *SCREENSAVE .

Syntax:

   *MDISPLAY hexaddr [num,num[,num,num[,hex]]]
*NOEGA

Cancels the effect of *EGA ON or *EGA OFF. After *NOEGA no mapping of modes takes place.

Syntax:

*NOEGA
*OUTPUT (*OPT)

Redirect console output to go to a communications port rather than the screen:

```
*OUTPUT 2
OSCLI "OUTPUT " + STR$(channel%)
```

The number following *OUTPUT corresponds to the channel number returned by the OPENUP function used to open the required comms port; this will be in the range 1 to 4 (see the Serial I/O section for details). Since this is determined at runtime, you will normally need to use OSCLI to access this command.

*OUTPUT 0 restores normal (screen) console output.

The following special values provide improved support for console mode programs and for emulating the LPRINT statement available in some BASIC dialects:

*OUTPUT 13  as *OUTPUT 1, except that 'non overlapped' I/O is used.
*OUTPUT 14  as *OUTPUT 2, except that 'non overlapped' I/O is used.
*OUTPUT 15  output is redirected to the printer.

Outputting text in *OUTPUT 15 mode has the side-effect of doing a VDU 3, 6 .

Syntax:

```
*OUTPUT num
```
*PLAY

Plays the specified MIDI file, assuming a suitable sound card is fitted.

*PLAY midifile
OSCLI "PLAY """"+midifile$+"""

Once a MIDI file is playing you cannot issue another *PLAY command until it has stopped (doing so will result in the Device unavailable error). However, you can cancel the current MIDI file using SOUND OFF.

You can discover whether a MIDI file is playing by using the @midi% System variable; this will be non-zero if a MIDI file is playing.

If the filename extension is omitted, .MID is assumed.

If you have a SoundBlaster™-compatible sound card, you can control the playback volume as follows:

SYS "auxSetVolume", 1, volume% + (volume% ◦ 16)

where volume% is in the range 0 (minimum) to 65535 (maximum). Note that the volume change will affect all subsequent MIDI audio output.

Syntax:

*PLAY ufsp
*PRINTER

Select the specified printer. Normally *BBC BASIC for Windows uses the current default printer, but you can select a different printer using this command.

```
*PRINTER HP Laserjet Plus
OSCLI "PRINTER "+printer$
```

The printer name is case insensitive, but otherwise must be precise. If the specified printer does not exist the No such printer error will result.

Changing the printer initialises the page margins to zero, *not* to the default value of 10 millimetres. You should normally follow the *PRINTER command with a *MARGINS command to set the page margins appropriately.

An alternative way of changing the printer is to present the user with the Print Setup dialogue box which will allow him not only to select the printer but also the printer options (e.g. print quality), page orientation and paper size.

**Syntax:**

```
*PRINTER name
```
*PRINTERFONT

Select the specified printer font (typeface) and size, and optionally select an emphasis style (bold, italic and/or underlined).

*PRINTERFONT Arial,16
*PRINTERFONT Courier New,20,U
*PRINTERFONT Letter Gothic,12,BI

If the *PRINTERFONT command is mistyped, or if the requested font is not installed, Windows™ will do its best to select a font similar to the one you requested. Unfortunately it doesn't always do this very well, and the result can be the selection of a 'symbols' font. If this happens, what is printed won't make much sense!

Changing the printer font initialises the page margins to zero. You may need to follow the *PRINTERFONT command with a *MARGINS command to reset the page margins appropriately.

Syntax:

*PRINTERFONT name,size[,style]
**QUIT**

This command is synonymous with *BYE and the QUIT statement.

**Syntax:**

*QUIT
**REFRESH**

By default the screen (BASIC's output window) is refreshed, i.e. updated, at times determined by Windows™. Typically if several things are being plotted in quick succession Windows will wait until all the changes have been made before refreshing the screen. This is normally perfectly satisfactory, but there may be special circumstances when you need to take control of when the refresh occurs.

For example if displaying an animated graphic you may need to force the display to update immediately, otherwise one or more frames of the animation may never be seen. Alternatively you may want to delay the refresh until an entire frame has been drawn, to reduce flicker. The options are as follows:

- **REFRESH** forces an immediate screen refresh.
- **REFRESH OFF** disables the automatic refresh; the screen will only be updated as the result of a *REFRESH command.
- **REFRESH ON** restores the normal behaviour.

Syntax:

```
*REFRESH [ON|OFF]
```
*RENAME (*REN)

Rename a disk file. The syntax is similar to the REN command available at a Command Prompt, except that the extension defaults to .BBC if it is omitted. To specify a filename with no extension, add a final full-stop.

```plaintext
*REN OLDFILE NEWFILE
OSCLI "REN """"+oldname$+"""" """"+newname$+""""
```

If a file already exists with the new name, a File exists error will occur.

If the *summed* lengths of the old and new filenames exceeds 254 characters a String too long error will result. In that case you can use the Windows™ API instead (the extensions must be included explicitly):

```plaintext
SYS "MoveFile", oldname$, newname$
```

Syntax:

```plaintext
*REN ufsp ufsp
*RENAME ufsp ufsp
```
**RMDIR (**RD\*)**

Remove (delete) the directory with the given name. The syntax is similar to the RMDIR command available at a Command Prompt:

*RD D:DATA
*RMDIR \BBCBASIC\TRIALS
OSCLI "RMDIR """+directory$+"""

You can only remove an empty directory. If the directory contains any files you will get an 'Access denied ' error. If the directory doesn't exist, you will get a File or path not found error.

**Syntax:**

*RD [d:]\[\]path
*RMDIR [d:]\[]path
Execute the specified external command and return to *BBC BASIC for Windows*. BASIC first tries to execute a Windows™ GUI application of the given name, and if that fails it tries to execute a console command at a Command Prompt (either a 'resident' command or a 'transient' program). If neither succeeds the message "Bad command or filename" will appear momentarily in the console window, but this may not be obvious. Care should therefore be taken to ensure that the specified command name is valid.

*BBC BASIC for Windows* will wait until the command completes, but if you press <Escape> the waiting will be aborted. If you terminate the command string with a semicolon (;) then BASIC will not wait for the command to complete.

```
*RUN C:\Program Files\Microsoft Office\Office\Winword
OSCLI "RUN " + command$
```

See also the ShellExecute Windows API function.

**Syntax:**

```
*RUN command [;]
```
*SAVE

Save an area of memory to a disk file. If the filename extension is omitted, .BBC is assumed. To specify a filename with no extension, add a final full-stop.

```
OSCLI "SAVE PROGFILE "+STR$~PAGE+" "+STR$~TOP
OSCLI "SAVE "+file$++++" "+STR$~start%++" "+STR$~length%
```

You **must** specify the start address and either the length of the area of memory (preceded by a + ) or the first address **not** to save. There is no 'load address' or 'execute address'.

**Syntax:**

```
*SAVE ufsp aaaaaaaaa +llllllll
*SAVE ufsp aaaaaaaaa bbbbbbbb
```
*SCREENSAVE

Save BASIC's output window (or part of it) as a Windows Bitmap file.

*SCREENSAVE screen.bmp xpos,ypos,width,height
OSCLI "SCREENSAVE ""+file$+"" ""+STR$xpos%+",""+STR$ypos%+",""+STR$width%+",""+STR$

If no position or size is specified, the entire window is saved (so long as the graphics ORIGIN hasn't been changed). Otherwise the rectangle whose bottom-left corner is at xpos,ypos and whose size is width (horizontal) by height (vertical) is saved. The position and size are given in BASIC's graphics units.

You can, of course, copy BASIC's output window to the clipboard using Alt-PrintScreen.

To display an image saved with *SCREENSAVE see *DISPLAY and *MDISPLAY.

Syntax:

*SCREENSAVE ufsp [num,num,num,num]
*SPOOL

Copy all subsequent console output to the specified file. If the filename is omitted, any current spool file is closed and spooling is terminated. If the extension is omitted,.BBC is assumed. To specify a filename with no extension, add a final full-stop.

*SPOOL LISTING.TXT
OSCLI "SPOOL """"+file$+""
*SPOOL

See also *EXEC and *SPOOLON .

Syntax:

*SPOOL [ufsp]
**SPOOLON**

Append all subsequent console output to the specified file. If the file does not exist, the File or path not found error will occur. If the extension is omitted, .BBC is assumed. To specify a filename with no extension, add a final full-stop.

*SPOOLON LISTING.TXT  
OSCLI "SPOOLON "***+file$+++"

You can use this command to add text to the end of a previously used spool file.

**Syntax:**

*SPOOLON ufsp
By default ON SYS intercepts only Windows™ WM_COMMAND messages. This is sufficient for its principal use of allowing a program to react to menu and dialogue box selections. However there are other messages to which you might wish to react. By issuing the following command:

```
*SYS 1
```

you can additionally intercept the WM_NOTIFY, WM_DROPFILES, WM_MENUSELECT, WM_INITMENU, WM_INITMENUPopup, WM_HOTKEY, WM_HELP and WM_CONTEXTMENU messages; refer to Microsoft documentation for details of these messages.

The ON SYS handler must test the value of the `@msg%` system variable to determine which message was received, as follows:

<table>
<thead>
<tr>
<th>Message</th>
<th><code>@msg%</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>WM_NOTIFY</td>
<td>78</td>
</tr>
<tr>
<td>WM_HELP</td>
<td>83</td>
</tr>
<tr>
<td>WM_CONTEXTMENU</td>
<td>123</td>
</tr>
<tr>
<td>WM_COMMAND</td>
<td>273</td>
</tr>
<tr>
<td>WM_INITMENU</td>
<td>278</td>
</tr>
<tr>
<td>WM_INITMENUPopup</td>
<td>279</td>
</tr>
<tr>
<td>WM_MENUSELECT</td>
<td>287</td>
</tr>
<tr>
<td>WM_DROPFILES</td>
<td>563</td>
</tr>
<tr>
<td>WM_HOTKEY</td>
<td>786</td>
</tr>
</tbody>
</table>

*SYS is equivalent to *SYS 0; it restores the default condition of intercepting only WM_COMMAND.

The `@wparam%` value of the WM_NOTIFY message contains the control's ID in the LS 16-bits (LOWORD) and the notification code in the MS 16-bits (HIWORD). This makes it possible to determine the notification code even when the parameter block pointed to by `@lparam%` is no longer valid.
The `wparam%` value of the WM_HELP message contains the context type in the LS 16-bits (LOWORD) and the control's ID in the MS 16-bits (HIWORD). This makes it possible to determine the control's ID even when the parameter block pointed to by `lparam%` is no longer valid.

**Syntax:**

```
*SYS [num]
```
**TEMPO**

Set the units of the duration parameter of the SOUND statement to the specified number of centiseconds (hundredths of a second). The default units are twentieths of a second (equivalent to *TEMPO 5*) which is suitable for most purposes. If you are using SOUND to play music, selecting *TEMPO 4* will cause it to play more quickly and selecting *TEMPO 6* will cause it to play more slowly.

You can select a short unit, for example one centisecond, if you want finer control over the duration of sounds. This may be important if you are using SOUND for purposes other than music, for example to generate morse code. Selecting a shorter unit also reduces the latency (the time between executing the SOUND statement and the sound actually being produced). However it places greater demands on your PC and on some systems may cause the sound output to 'stutter'.

Note that the maximum duration of a single SOUND statement is 254 times the selected unit.

*TEMPO 4

By adding 128 to the *TEMPO* value you can configure SOUND channel 0 to be a 'tone' channel rather than a 'noise' channel; this makes it possible to generate 'four voice' music:

*TEMPO 132

**Syntax:**

*TEMPO num
*TIMER

Set the periodicity of the ON TIME interrupt to the specified number of milliseconds; the smallest value you can specify is 10 ms (resulting in approximately 100 ON TIME interrupts per second).

*TIMER 200

Note that the timer period determines the flash rate of flashing characters in MODE 7. You are recommended not to change it when using that mode. The default periodicity (if no *TIMER command is issued) is 250 milliseconds (4 Hz).

Syntax:

*TIMER num
*TV

The *TV command is trapped and ignored.

Syntax:

    *TV
Type the specified file to the screen. If the filename extension is omitted, .BBC is assumed. To specify a filename with no extension, add a final full-stop.

```
*TYPE modes.txt
OSCLI "TYPE """"+file$+"""
```

You can use this command to display the contents of an ASCII file. You can pause the display either by holding down <Shift> and <Ctrl> simultaneously, or by first entering 'paged' mode by typing Ctrl/N (VDU 14). In 'paged' mode, the display pauses until <Shift> is pressed. To cancel 'paged' mode press Ctrl/O (VDU 15).

To display a program saved in internal (tokenised) format use *LIST.

**Syntax:**

```
*TYPE ufsp
```
*UNLOCK

Set the specified file to 'read/write' status. If the extension is omitted, .BBC is assumed. To specify a filename with no extension, add a final full-stop.

*UNLOCK ADDRESS.DTA
OSCLI "UNLOCK """"+file$"""""

This command reverses the effect of *LOCK.

Syntax:

*UNLOCK ufsp
This is a comment line. Anything following the '|' is ignored. The effect is similar to the REM statement, except that comments delimited by *| are retained in a compiled executable even if the Remove REMs crunch option is selected. Thus you should normally avoid using *| if you intend to 'compile' your program.

Syntax:

*| any text
Additional OS Interfaces

In addition to the facilities already described *BBC BASIC for Windows* provides a very limited emulation of the BBC Micro's OSBYTE and OSWORD functions. The emulation of OSBYTE is limited to reading the character at the current text cursor position (OSBYTE function &87) and the emulation of OSWORD is limited to reading the dot pattern of a given font character (OSWORD call &0A). Because these are the only OSBYTE and OSWORD functions emulated, it is not strictly necessary to pass the function number in A%, but it is best to do so to ensure compatibility with any future enhancements to *BBC BASIC for Windows*.

**Read current character - OSBYTE &87**

The following program segment returns the ASCII code of the character at the current text cursor position and assigns it to a variable called 'char':

\[ A\%=&87 \]
\[ \text{char}=(\text{USR}(&\text{FFF4}) \text{ AND } &\text{FF00}) \text{ DIV } 256 \]

**Read character dot pattern - OSWORD &0A**

The following program segment reads the dot pattern of a character:

\[ \text{DIM pattern 8} \]
\[ ?\text{pattern}=\text{character} \]
\[ A\%=&0A \]
\[ X\%=\text{pattern MOD 256} : Y\%=\text{pattern DIV 256} \]
\[ \text{CALL } &\text{FFF1} \]

The ASCII value is loaded into byte zero of the 9 byte table called 'pattern' and the pattern is returned in the 8 bytes 'pattern?1' to 'pattern?8. As with the BBC Micro, the address of the table 'pattern' is passed in X% and Y%. The least significant byte is passed in X% and the most significant in Y%.

Note that the pattern is read from a built-in font which resembles that of the BBC Micro. It does **not** correspond to the actual font which is used when text is written to the screen by BASIC. Generally speaking the fonts used by Windows™ do not correspond to an 8 x 8 pixel pattern.
You can, if you wish, use this built-in font in a BASIC program (instead of a standard Windows™ font) by replacing all the standard characters with user-defined characters using VDU 23 as follows:

```
DIM p 8
A% = &0A
X% = p MOD 256
Y% = p DIV 256
FOR N% = 32 TO 127
  ?p = N%
  CALL &FFF1
NEXT
```
Assembler access to OS routines

Your assembler programs may access the OSBGET, OSBPUT, OSRDCH, OSASCI, OSNEWL, OSWRCH, OSWORD, OSBYTE and OSCLI routines by name as shown below:

```
CALL "osbget" ; Read byte from file to AL, EBX contains channel number
CALL "osbput" ; Write byte from AL to file, EBX contains channel number
CALL "osrdch" ; Read keyboard character to AL
CALL "osasci" ; Write AL to the VDU drivers (plus LF if CR)
CALL "osnewl" ; Write LF,CR
CALL "oswrch" ; Write AL to the VDU drivers
CALL "osword" ; Read character dot pattern, EDX addresses buffer
CALL "osbyte" ; Read character at cursor position to AL
CALL "oscli" ; Various OS commands, EDX addresses string
CALL "osline" ; Read a line from the console, EDX addresses buffer (DL=0)
CALL "osshut" ; Close a file, EBX = channel number
CALL "getptr" ; Read file pointer, EBX = channel number, result in EDX:EAX
CALL "setptr" ; Set file pointer, EBX = channel number, EDX:EAX = value
CALL "getext" ; Read file length, EBX = channel number, result in EDX:EAX
CALL "setext" ; Set file length, EBX = channel number, EDX:EAX = value
CALL "osopen" ; Open a file, EDX addresses filename, AL = 0 (read)
              ; 1 (create) or 2 (update), channel number returned in EAX
```

All strings are CR-terminated. In the case of 'oskey' the carry flag is cleared if no key was pressed within the timeout period. In the case of 'osbget' the carry flag is cleared if at end-of-file. In *BBC BASIC for Windows version 6 or later only* the EDX register contains the most significant 32-bits of the file pointer/length.
Introduction to disk files

These notes start with some basic information on files, and then go on to discuss file access in BBC BASIC, serial (sequential) files, random access files and, finally, indexed files. The commands and functions used are explained, and illustrated by examples.

If you are new to BBC BASIC for Windows, or you are experiencing difficulty with disk files, you might find these notes useful. Some of the concepts and procedures described are quite complicated and require an understanding of file structures. If you have trouble understanding these parts, don't worry. Try the examples and write some programs for yourself and then go back and read the notes again. As you become more comfortable with the fundamentals, the complicated bits become easier.

The programs listed in these notes are included with the supplied example programs. They are provided for demonstration and learning purposes and are not intended to be taken and used as working programs without modification to suit your needs. If you want to, you are free to incorporate any of the code in the programs you write. Use them, change them, or ignore them as you wish. There is only one proviso; the programs have been tested and used a number of times, but we cannot say with certainty that they are bug free. Remember, debugging is the art of taking bugs out - programming is the art of putting them in.
The structure of files

If you understand the way files work, skip the next two paragraphs. If you understand random and indexed files, skip the following two paragraphs as well.

File basics

Many people are confused by the jargon that is often used to describe the process of storing and retrieving information. This is unfortunate, because the individual elements are very simple and the most complicated procedures are only a collection of simple bits and pieces.

All computers are able to store and retrieve information from a non-volatile medium (in other words, you don't lose the information when the power gets turned off). Audio cassettes were once used for small micro computers, diskettes for medium sized systems and magnetic tape and large disks for big machines. Nowadays, all but the smallest machines incorporate a hard disk with several gigabytes of storage. In order to be able to find the information you want, the information has to be organised in some way. All the information on one general subject is gathered together into a FILE. Within the file, information on individual items is grouped together into RECORDS.

Serial (sequential) files

Look upon the disk as a drawer in a filing cabinet. The drawer is full of folders called FILES and each file holds a number of enclosures called RECORDS. Sometimes the files are in order in the drawer, sometimes not. If you want a particular file, you start at the beginning of the drawer and search your way through until you find the file you want. Then you search your way through the records in the file until you find the record you want.

Life is easier with a computer that uses disks. There is an index which tells the computer where to look for each of the files and the serial search for the file is not necessary. However, once you have found the file, you still need to read through it to find the record you want.

There are a number of ways to overcome this problem. We will consider the two simplest; random access (or relative) files and indexed files.
Random access files

The easiest way to find the record you want is to identify each record with a number, like an account number. You can then ask for, say, the 23rd record. This is similar to turning to page 23 in the account book. This works very well at first. Every time you get a new customer you start a new page. Most of the pages have a lot of empty space, but you must have the same amount of space available for each account, otherwise your numbering system won't work. So, even at the start, there are a lot of gaps.

What happens when you close an account? You can't tear out the page because that would upset the numbering system. All you can do is draw a line through it - in effect, turn it into a blank page. Before long, quite a number of pages will be 'blank' and a growing proportion of your book is wasted.

With other forms of 'numbering', say by the letters of the alphabet, you could still not guarantee to fill all the pages. You would have to provide room for the Zs, but you may never get one. When you started entering data, most of the pages would be blank and the book would only gradually fill up.

The same happens with this sort of file on a disk. A random file which has a lot of empty space in it is described as sparse. Most random files start this way and most never get more than about ¾ full. Count the number of empty 'slots' in your address book and see what proportion this is of the total available.

Indexed files

Suppose we want to hold our address book on the computer. We need a number of records each holding the name, address, telephone number, etc of one person. In our address book, we have one or two pages per letter of the alphabet and a number of 'slots' on each page. With this arrangement, the names are in alphabetical order of their first letter. This is very similar to the way the accounts book was organised except that we don't know the page number for each name.

If we had an index at the front of the book we could scan the index for the name and then turn to the appropriate page. We would still be wasting a lot of space because some names, addresses etc are longer than others and our 'slots' must be large enough to hold the longest.
Suppose we numbered all the character positions in the book and we could easily move to any character position. We could write all the names, addresses, etc., one after the other and our index would tell us the character position for the start of each name and address. There would be no wasted space and we would still be able to turn directly to the required name.

What would happen when we wanted to cancel an entry? We would just delete the name from the index. The entry would stay in its original place in the book, but we would never refer to it. Similarly, if someone changed their address, we would just write the name and the new address immediately after the last entry in the book and change the start position in the index. Every couple of years we would rewrite the address book, leaving out those items not referenced in the index and update the index (or write another one).

This is not a practical way to run a paper and pencil address book because it's not possible to turn directly to the 3423rd character in a book, and the saving in space would not be worth the tedium involved. However, with BBC BASIC you can turn to a particular character in a file and the tedium only takes a couple of seconds, so it's well worth doing.
Files in BBC BASIC

Introduction

With serial files the records need only be as large as the data to be stored and there are no empty records (the data item FRED only occupies 4 bytes whereas ERMINTRUDE occupies 10 bytes). Consequently serial files are the most space efficient way to hold data on a disk (or any other storage medium).

Serial files cannot be used to access particular records from within the file quickly and easily. In order to do this with the minimum access time, random access files are necessary. However, a random file generally occupies more space on the disk than a serial file holding the same amount of data because the records must be a fixed length and some of the records will be empty.

Most versions of BASIC only offer serial and random files, but because of the way that disk files are handled by BBC BASIC it is possible to construct indexed, and even linked, files as well. Indexed files take a little longer to access than random files and it is necessary to have a separate index file, but they are generally the best space/speed compromise for files holding a large amount of data.

How data is read/written

As far as the programmer is concerned, data can be written to and read from a file a data item or a character (byte) at a time. In fact, there is a buffer between the program and the operating system (e.g. Windows™), but this need only concern you when you are organising your program for maximum disk efficiency.

Because of the character by character action of the write/read process, it is possible (in fact, necessary) to keep track of your position within the file. BBC BASIC does this for you automatically and provides a pointer PTR (a pseudo-variable) which holds the position of the NEXT character (byte) to be written/read. Every time a character is written/read PTR is incremented by one, but it is possible to set PTR to any number you like. This ability to 'jump around' the file enables you to construct both random (relative) and indexed files.

BBC BASIC provides the facility for completely free-format binary data files. Any file which can be read by the computer, from any source and in any data format, can
be processed using the BGET, BPUT and PTR functions.

**How data is stored**

**Numeric data**

In order to make the most efficient use of disk space and to preserve accuracy, numerics are stored in a data file in binary format, not as strings of characters. To prevent confusion when numerics are being read from a file, both integers and reals occupy 5 bytes (40 bits), or 8 bytes in *FLOAT 64 mode. If they were stored as character strings they could occupy up to 10 bytes. For compatibility with other BASICs, you can store numerics as strings by using the STR$ function.

**How strings are stored**

Strings are stored in a data file (with PRINT# ) as the ASCII bytes of the string followed by a carriage-return. If you need a line feed as well, it's no problem to add it using BPUT#. Similarly, extraneous characters included in files produced by other programs can be read and, if necessary, discarded using BGET#.

**How files are referred to**

We refer to a file by its name. Unfortunately, this is too complicated (and inefficient) for the computer. Consequently, the only time BBC BASIC for Windows refers to a file by its name is when it opens the file. From then on, it refers to the file by the number it allocated to it when it was opened. This number is called the 'channel number'.

Page 550
BBC BASIC file access

File buffering

Logically, BBC BASIC for Windows transfers data to and from files one byte at a time. However, Windows™ does not handle single byte data transfer efficiently and, in order to speed disk access, BASIC buffers data into blocks. There are eight buffers provided for this purpose, so you can open a maximum of eight files at any one time. If you attempt to open a ninth file, you will get a Too many open files error.

Apart from the speed increase, file buffering should be transparent to the user and, unless you are doing something unconventional, you can forget about it. It is worth noting that using the PTR#file_num= statement has the side-effect of flushing the appropriate BBC BASIC for Windows file buffer.

Networking - shared files

BBC BASIC for Windows provides limited support for file sharing on a networked system. The effect is that any given file can be opened (at the same time) once with OPENUP or OPENOUT and any number of times with OPENIN (in other words only one process can be writing to the file, but any number can be reading from it). The first program (process) to successfully open the file with OPENUP or OPENOUT prevents any other program doing the same; other programs can only open the file with OPENIN.

If an attempt to open a file fails because of a sharing violation, the OPENOUT or OPENUP call will return the value zero. In the case of OPENUP this is the same value which is returned if the file is not found; you can distinguish between the two cases by attempting to open the file with OPENIN. If that succeeds, the problem must have been a sharing violation.
Disk file commands

Introduction

The statements and functions used in disk file manipulation are described below. They are not in alphabetical order, but in the order you are likely to want to use them. Whilst these notes repeat much of the material covered in the Keywords section, additional information has been added and they are presented in a more readable order.

Filenames

Please refer to your Windows™ documentation for a full explanation of disk, directory (folder) and file names. The explanation below is only intended as a brief reference guide.

Windows™ accepts a composite file name (file specification) in the following format:

```
DRIVENAME\PATHNAME\FILENAME.EXTension
```

The drivename is a single letter followed by a colon and denotes the disk drive on which the file will be found or created, e.g. C:

The pathname is the name of the directory or the path to the directory (folder) in which the file will be found or created. Directories are separated by the backslash character, e.g. WINDOWS\SYSTEM

The filename and optional extension identify a single file within the directory (folder). Whenever a filename without an extension is given, BBC BASIC for Windows will append .BBC as the default extension. If you want to maintain compatibility with MS-DOS™ you should restrict filenames to eight characters or fewer and extensions to three characters or fewer; you should also use only capital letters and numbers. Windows™ does not have these restrictions, although filenames and path names are not case-sensitive.

Disk data files
Introduction

The statements and functions used for data files are:

```
OPENIN
OPENUP
OPENOUT
EXT#
PTR#
INPUT#  BGET#  GET$
PRINT#  BPUT#
CLOSE#  END
EOF#
```

Opening files

You cannot use a file until you have OPENed it and it has been allocated a 'channel number'. Most other versions of BASIC allow you to choose the channel number. In order to improve efficiency, BBC BASIC for Windows chooses it for you.

When you open the file, a channel number (an integer number) is returned by the interpreter and you will need to store it for future use. The open commands are, in fact, functions which open the appropriate file and return its channel number.

You use the channel number for all subsequent access to the file (with the exception of the Operating System commands).

If the system has been unable to open the file, the channel number returned will be 0. This will occur if you try to open a non-existent file for input (OPENIN or OPENUP) or a read-only file for output (OPENOUT or OPENUP) or whenever the file cannot be opened for another reason.

File opening functions

The three functions which open files are OPENIN, OPENUP and OPENOUT. OPENOUT should be used to create new files, or overwrite old ones. OPENIN should be used for input only and OPENUP should be used for input/output (update).

OPENOUT
A function which opens a file for output and returns the channel number allocated. The use of OPENOUT destroys the contents of the file if it previously existed and sets its length to zero. If the file did not exist, a new file is created.

```plaintext
file_num = OPENOUT "PHONENUMS.DAT"
```

You always need to store the channel number because it must be used for all the other file statements and functions. If you choose a variable with the same name as the file, you will make programs which use a number of files easier to understand:

```plaintext
phonenums=OPENOUT "PHONENUMS.DAT"
opfile=OPENOUT opfile$
```

**OPENIN**

A function which opens a file for input only. You cannot write to a file opened with OPENIN.

```plaintext
address=OPENIN "ADDRESS.DAT"
check_file=OPENIN check_file$
```

OPENIN will fail (channel number returned = 0) if the file does not already exist.

If you try to write to a file opened with OPENIN you will get an 'Access denied' error (error number 189).

A file may be opened any number of times with OPENIN, but only once with OPENUP or OPENOUT. See the sub-section Networking - shared files for more details.

**OPENUP**

A function which opens a file for update (input or output) without destroying the existing contents of the file. None of the previously written data is lost unless it has been over-written. Consequently, you would use OPENUP for reading serial and random files, adding to the end of serial files or writing to random files.

```plaintext
address=OPENUP "ADDRESS.DAT"
```
OPENUP will fail (channel number returned = 0) if the file does not already exist.

A file may be opened once with OPENUP and any number of times with OPENIN. See the earlier sub-section Networking - Shared Files for more details.

CLOSE#

A statement which closes a file, signifying that your program has finished with it. Once a file has been closed the channel number associated with that file is no longer valid. CLOSE#0, END or 'dropping off the end' of a program will close all open files.

CLOSE#fnum
CLOSE#0

INPUT#

A statement which reads data (in BBC BASIC for Windows internal format) from a file into one or more variables. Several values can be read using the same INPUT# statement.

INPUT#data,name$,age,height,sex$

For this to work correctly the data must have been written with a PRINT# statement. READ# can be used as an alternative to INPUT#

PRINT#

A statement which writes data to a file (in BBC BASIC for Windows internal format). Several values can be written using the same PRINT# statement. String variables are written as the character bytes in the string plus a carriage-return. Numeric variables are written as 5 bytes of binary data (8 bytes in *FLOAT64 mode).

PRINT#data,name$,age,height,sex$
**EXT#**

A function which returns the total length of the file in bytes (characters).

\[ \text{length} = \text{EXT#fnum} \]

EXT# can also be used to change the length of a file:

\[ \text{EXT#fnum} = \text{newlength} \]

**PTR#**

A pseudo-variable which points to the position within the file from where the next byte to be read will be taken or where the next byte to be written will be put.

\[ \text{oldptr} = \text{PTR#fnum} \]
\[ \text{PTR#fnum} = \text{newptr} \]

When the file is OPENED, PTR# is set to zero. However, you can set PTR# to any value you like (even beyond the end of the file - so take care).

Reading or writing, using INPUT# and PRINT# , (and BGET# and BPUT# - explained later), takes place at the current position of the pointer. The pointer is automatically updated following a read or write operation.

A file opened with OPENUP may be extended by setting PTR# to its end (PTR# = EXT#), and then writing the new data to it. You must remember to CLOSE such a file in order to update its directory entry with its new length. A couple of examples of this are included in the sections on serial and indexed files.

Using a 'PTR#fnum=' statement will flush the appropriate BBC BASIC for Windows file buffer. This can be useful when you need to ensure that the data has actually been written to disk (or sent to a remote device).

**EOF#**

A function which returns -1 (TRUE ) if the data file whose channel number is the argument is at (or beyond) its end. In other words, when PTR# points beyond the
current end of the file.

```
EOF# is only really of use when dealing with serial (sequential) files. It indicates
that PTR# is greater than the recorded length of the file (found by using EXT# ).
When reading a serial file, EOF# would go true when the last byte of the file had
been read.

EOF# will NOT be true if an attempt has been made to read from an empty area of a
sparse random access file. Reading from an empty area of a sparse file will return
garbage. Because of this, it is difficult to tell which records of an uninitialised
random access file have had data written to them and which are empty. These files
need to be initialised and the unused records marked as empty.

BGET#

A function which reads a single byte (character) from a file, at the position pointed
to by PTR#. PTR# is incremented by one following the read. Each byte in the file is
returned as a positive integer between 0 and 255 (&00 to &FF). This can be
converted into a single- character string using the CHR$ function.

```
byte=BGET#fnum
char$=CHR$(byte)
```
or, more expediently

```
char$=CHR$(BGET#fnum)
```

BPUT#

A statement which writes a single byte (character) to a file, at the position pointed to
by PTR#. PTR# is incremented by one following the write.

```
BPUT#fnum, &1B
```
BPUT#fnum,house_num
BPUT#fnum,ASC "E"
Serial files

The section on serial files is split into three parts. The first deals with character data files. These are the simplest type of files to use and the examples are correspondingly short. The second part looks at mixed numeric/character data files. The final part describes conversion between *BBC BASIC for Windows* format files and the file formats required/produced by other systems.

Character data files

The first three examples are programs to write data in character format to a serial file and to read the data back. All the data is in character format and, since the files will not be read by other versions of BASIC, no extra control characters have been added.

You may notice that we have cheated a little in that a procedure is called to close the files and end the program without returning. This saves using a GOTO, but leaves the return address on the stack. However, ending a program clears the stack and no harm is done. You should not use this sort of trick anywhere else in a program. If you do you will quickly use up memory.

Writing serial character data

```plaintext
REM F-WSER1
:
REM EXAMPLE OF WRITING TO A SERIAL CHARACTER DATA FILE
:
REM This program opens a data file and writes serial
REM character data to it. The use of OPENOUT ensures that,
REM even if the file existed before, it is cleared before
REM being written to.
:
OSCLI "CD """@usr$"""
:
phonenos=OPENOUT "PHONENOS.DAT"
PRINT "File Name PHONENOS.DAT Opened as Channel "; phonenos
PRINT
REPEAT
  INPUT "Name ? " name$
  IF name$="" THEN PROC_end
  INPUT "Phone Number ? " phone$
  PRINT
  PRINT#phonenos, name$, phone$
```
UNTIL FALSE

DEF PROC_end
CLOSE#phonenos
END

Reading serial character data

REM F-RSER1 :
REM EXAMPLE OF READING A SERIAL CHARACTER FILE :
REM This program opens a previously written serial file
REM and reads it. :
OSCLI "CD "+@usr$+"
phonenos=OPENIN "PHONENOS.DAT"
PRINT "File Name PHONENOS.DAT Opened as Channel ";phonenos
PRINT
REPEAT
  INPUT#phonenos,name$,phone$
  PRINT name$,phone$
UNTIL EOF#phonenos :
CLOSE#phonenos
END

Appending to character data files

The next example extends the write program from Example 1. This new program
opens the file, sets PTR# to the end and then adds data to it. A procedure is used to
open the file. This has the advantage of making the program more understandable by
putting the detailed 'open at end' coding out of the main flow of the program.

REM F-WESER1 :
REM EXAMPLE OF WRITING TO THE END OF A SERIAL DATA FILE :
REM This program opens a file and sets PTR to the end
REM before writing more data to it. :
REM A function is used to open the file. :
OSCLI "CD "+@usr$+"
phonenos=FN_openend("PHONENOS.DAT")
PRINT "File Name PHONENOS.DAT Opened as Channel ";phonenos
PRINT
REPEAT
    INPUT "Name ? " name$
    IF name$="" THEN PROC_end
    INPUT "Phone Number ? " phone$
    PRINT
    PRINT#phonenos,name$,phone$
UNTIL FALSE
:
DEF PROC_end
CLOSE#phonenos
END
:
REM Open the file 'AT END'.
:
REM If the file does not already exist, it is created
REM with OPENOUT. PTR# is left at zero and the file
REM number is returned. If the file exists, PTR# is
REM set to the end and the file number returned.
:
DEF FN_openend(name$)
LOCAL fnum
fnum=OPENUP(name$)
IF fnum=0 THEN fnum=OPENOUT(name$): =fnum
PTR#fnum=EXT#fnum
=fnum

Mixed numeric/character data files

The second three examples are also programs which write data to a file and read it back, but this time the data is mixed. They are simply extensions of the previous examples which illustrate the handling of mixed data.

Writing a mixed data file

REM F-WSER2
 :
REM EXAMPLE OF WRITING TO A MIXED NUMERIC/CHAR DATA FILE
 :
REM This program opens a data file and writes numeric
REM and character data to it. The use of OPENOUT
REM ensures that, even if the file existed before,
REM it is cleared before being written to.
 :
REM Functions are used to accept and validate the
REM data before writing it to the file.
:
OSCLI "CD """@usr$""
:
stats=OPENOUT("STATS.DAT")
PRINT "File Name STATS.DAT Opened as Channel ";stats
PRINT
REPEAT
  name$=FN_name
  IF name$="" THEN PROC_end
  age=FN_age
  height=FN_height
  sex$=FN_sex
  PRINT
  PRINT#stats,name$,age,height,sex$
UNTIL FALSE:
DEF PROC_end
PRINT "The file is ";EXT#stats;" bytes long"
CLOSE#stats
END:
:
REM Accept a name from the keyboard and make sure it
REM consists only of spaces and upper or lower case
REM characters. Leading spaces are automatically
REM ignored on input.
:
DEF FN_name
LOCAL name$,FLAG,n
REPEAT
  FLAG=TRUE
  INPUT "Name ? " name$
  IF name$<>"" THEN
    FOR I=1 TO LEN(name$)
      n=ASC(MID$(name$,I,1))
      IF NOT(n=32 OR n>64 AND n<91 OR n>96 AND n<123) THEN FLAG=FALSE
    NEXT
    IF NOT FLAG THEN PRINT "No funny characters please !!!"
  ENDIF
UNTIL FLAG

=NAME$
:
REM Accept the age from the keyboard and round to one
REM place of decimals. Ages of 0 and less or 150 or
REM more are considered to be in error.
:
DEF FN_age
LOCAL age
REPEAT
  INPUT "What age ? " age
  IF age<=0 OR age >=150 THEN PRINT "No impossible ages please !!!"
UNTIL age>0 AND age<150
=INT(age*10+.5)/10
:
:

REM Accept the height in centimeters from the keyboard
REM and round to an integer. Heights of 50 or less and
REM 230 or more are considered to be in error.
 :
DEF FN_height
LOCAL height
REPEAT
  INPUT "Height in centimeters ? " height
  IF height<=50 OR height>=230 THEN PRINT "Verry funny !!!"
UNTIL height>50 AND height<230
=INT(height+.5)
 :
REM Accept the sex from the keyboard. Only words
REM beginning with upper or lower case M or F are OK.
REM The returned string is truncated to 1 character.
 :
DEF FN_sex
LOCAL sex$,FLAG
REPEAT
  FLAG=TRUE
  INPUT "Male or Female - M or F ? " sex$
  IF sex$<>"" THEN sex$=CHR$(ASC(MID$(sex$,1,1)) AND 95)
  IF sex$<="M" AND sex$<>"F" THEN FLAG=FALSE
  IF NOT FLAG THEN PRINT "No more sex(es) please !!!"
UNTIL FLAG
=sex$

Reading a mixed data file

REM F-RSER2
 :
REM EXAMPLE OF READING FROM A MIXED NUMERIC/CHAR DATA FILE
 :
REM This program opens a data file and reads numeric and
REM character data from it.
 :
OSCLI "CD "+@usr$+
 :
stats=OPENIN("STATS.DAT")
PRINT "File Name STATS.DAT Opened as Channel ";stats
PRINT
REPEAT
  INPUT#stats,name$,age,weight,sex$
  PRINT "Name ";name$
  PRINT "Age ";age
  PRINT "Height in centimeters ";weight
  IF sex$="M" THEN PRINT "Male" ELSE PRINT "Female"
  PRINT
UNTIL EOF#stats
 :
CLOSE#stats
Appending to mixed data files

This example is similar to Example 3, but for a mixed data file.

```fortran
REM F-WESER2
: REM EXAMPLE OF WRITING AT THE END OF A MIXED NUMERIC/CHAR
REM DATA FILE :
: REM This program opens a data file, sets PTR to its end
REM and then writes numeric and character data to it. :
: REM Functions are used to accept and validate the data
REM before writing it to the file. :
: OSCLI "CD "@usr$""
:
stats=FN_open("STATS.DAT")
PRINT "File Name STATS.DAT Opened as Channel ";stats
PRINT
REPEAT
  name$=FN_name
  IF name$="" THEN PROC_end
  age=FN_age
  height=FN_height
  sex$=FN_sex
  PRINT
  PRINT#stats,name$,age,height,sex$
UNTIL FALSE :
DEF PROC_end
PRINT "The file is ";EXT#stats;" bytes long"
CLOSE#stats
END :
REM Open the file. If it exists, set PTR# to the end and
REM return the file number. If it does not exist, open it,
REM leave PTR# as it is and return the file number. :
DEF FN_open(name$)
LOCAL fnum
fnum=OPENUP(name$)
IF fnum=0 THEN fnum=OPENOUT(name$): =fnum
PTR#fnum=EXT#fnum =fnum :
REM Accept a name from the keyboard and make sure it
REM consists of spaces and upper or lower case characters.
```
REM Leading spaces are automatically ignored on input.

:  DEF FN_name
LOCAL name$,FLAG,n
REPEAT
  FLAG=TRUE
  INPUT "Name ? " name$
  IF name$<>"" THEN
     FOR I=1 TO LEN(name$)
        n=ASC(MID$(name$,I,1))
        IF NOT(n=32 OR n>64 AND n<91 OR n>96 AND n<123) THEN FLAG=FALSE
     NEXT
     IF NOT FLAG THEN PRINT "No funny characters please !!!"
  ENDIF
UNTIL FLAG
=name$
:
REM Accept the age from the keyboard and round to one place
REM of decimals. Ages of 0 and less or 150 or
REM more are in error.
:  DEF FN_age
LOCAL age
REPEAT
  INPUT "What age ? " age
  IF age<=0 OR age >=150 THEN PRINT "No impossible ages please !!!"
UNTIL age>0 AND age<150
=INT(age*10+.5)/10
:
REM Accept the height in centimeters from the keyboard and
REM round to an integer. Heights of 50 or less and 230 or
REM more are in error.
:  DEF FN_height
LOCAL height
REPEAT
  INPUT "Height in centimeters ? " height
  IF height<=50 OR height>=230 THEN PRINT "Very funny !!!"
UNTIL height>50 AND height<230
=INT(height+.5)
:
REM Accept the sex from the keyboard. Only words beginning
REM with upper or lower case M or F are acceptable. The
REM returned string is truncated to 1 character.
:  DEF FN_sex
LOCAL sex$,FLAG
REPEAT
  FLAG=TRUE
  INPUT "Male or Female - M or F ? " sex$
  IF sex$<="" THEN sex$=CHR$(ASC(MID$(sex$,1,1)) AND 95)
  IF sex$<="M" AND sex$<="F" THEN FLAG=FALSE
Compatible data files

The next example tackles the problem of writing files which will be compatible with other versions of BASIC. The most common format for serial files is as follows:

- Data is written to the file as ASCII characters.
- Data items are separated by commas.
- Records are terminated by the two characters CR and LF.
- The file is terminated by a Control Z (&1A).

The example program accepts data from the keyboard and writes it to a file in the above format.

Writing a compatible data file

```basic
REM F-WSTD
: REM EXAMPLE OF WRITING A COMPATIBLE DATA FILE
: REM This program opens a data file and writes numeric and
REM character data to it in a compatible format. Numerics
REM are changed to strings before they are written and the
REM data items are separated by commas. Each record is
REM terminated by CR LF and the file is terminated by a
REM Control Z.
: REM Functions are used to accept and validate the data
REM before writing it to the file.
: OSCLI "CD ""+@usr$+"""
: compat=OPENOUT("COMPAT.DAT")
PRINT "File Name COMPAT.DAT Opened as Channel ";compat
PRINT
REPEAT
  name$=FN_name
  IF name$="" THEN PROC_end
  age=FN_age
  height=FN_height
  sex$=FN_sex
```

Page 566
PRINT
record$=name$+","+STR$(age)$+","+STR$(height)$+","+sex$
PRINT#compat,record$
BPUT#compat,&0A
UNTIL FALSE

DEF PROC_end
BPUT#compat,&1A
CLOSE#compat
END:

REM Accept a name from the keyboard and make sure it
REM consists only of spaces and upper or lower case
REM characters. Leading spaces are automatically ignored
REM on input.

DEF FN_name
LOCAL name$,FLAG,n
REPEAT
  FLAG=TRUE
  INPUT "Name ? " name$
  IF name$<>"" THEN
      FOR I=1 TO LEN(name$)
          n=ASC(MID$(name$,I,1))
          IF NOT(n=32 OR n>64 AND n<91 OR n>96 AND n<123) THEN FLAG=TRUE
      NEXT
      IF NOT FLAG THEN PRINT "No funny characters please !!!"
  ENDIF
  UNTIL FLAG
=\name$
:

REM Accept the age from the keyboard and round to one place
REM of decimals. Ages of 0 and less or 150 or more are
REM considered to be in error.

DEF FN_age
LOCAL age
REPEAT
  INPUT "What age ? " age
  IF age<=0 OR age >=150 THEN PRINT "No impossible ages please !!!"
UNTIL age>0 AND age<150
=INT(age*10+.5)/10
::

REM Accept the height in centimeters from the keyboard and REM round to an integer. Heights of 50 or less and 230 or REM more are considered to be in error.

DEF FN_height
LOCAL height
REPEAT
  INPUT "Height in centimeters ? " height
  IF height<=50 OR height>=230 THEN PRINT "Verry funny !!!"
UNTIL height>50 AND height<230
=INT(height+.5)
:
:
REM Accept the sex from the keyboard. Only words beginning
REM with upper or lower case M or F are acceptable. The
REM returned string is truncated to 1 character.
:
DEF FN_sex
LOCAL sex$,FLAG
REPEAT
  FLAG=TRUE
  INPUT "Male or Female - M or F ? " sex$
  IF sex$<>"" THEN sex$=CHR$(ASC(MID$(sex$,1,1)) AND 95)
  IF sex$<"M" AND sex$<"F" THEN FLAG=FALSE
  IF NOT FLAG THEN PRINT "No more sex(es) please !!!"
UNTIL FLAG
=sex$

**Reading a compatible data file**

The last example in this section reads a file written in the above format and strips off
the extraneous characters. The file is read character by character and the appropriate
action taken. This is a simple example of how *BBC BASIC for Windows* can be used
to manipulate any Windows™ file by processing it on a character by character basis.

REM F-RSTD
:
REM EXAMPLE OF READING A COMPATIBLE DATA FILE
:
REM This program opens a data file and reads numeric and
REM character data from it. The data is read a byte at a
REM time and the appropriate action taken depending on
REM whether it is a character, a comma, or a control char.
:
OSCLI "CD "@usr$+""
:
compat=OPENUP("COMPAT.DAT")
PRINT "File Name COMPAT.DAT Opened as Channel ";compat
PRINT
REPEAT
  name$=FN_read
  PRINT "Name ";name$
  age=VAL(FN_read)
  PRINT "Age ";age
  height=VAL(FN_read)
  PRINT "Height in centimeters ";height
  sex$=FN_read
  IF sex$="M" THEN PRINT "Male" ELSE PRINT "Female"
  PRINT
UNTIL FALSE
:
:
REM Read a data item from the file. Treat commas and CRs
REM as data item terminators and Control Z as the file
REM terminator. Since we are not interested in reading a
REM record at a time, the record terminator CR LF is of no
REM special interest to us. We use the CR, along with
REM commas, as a data item separator and discard the LF.
:
DEF FN_read
LOCAL data$,byte$,byte
dat$=""
REPEAT
  byte=BGET#compat
    IF byte=&1A OR EOF#compat THEN CLOSE#compat: END
    IF NOT(byte=&0A OR byte=&0D OR byte=&2C) THEN data$=data$+CHR$(byte)
UNTIL byte=&0D OR byte=&2C
=data$
Random (relative) files

There are three example random file programs. The first is very simple, but it demonstrates the principle of random access files. The second expands the first into quite a useful database program. The final example is an inventory program. Although it does not provide application dependent features, it would serve as it stands and it is sufficiently well structured to be expanded without too many problems.

Designing the file

Unlike other versions of BASIC, there is no formalised record structure in BBC BASIC. A file is considered to be a continuous stream of bytes (characters) and you can directly access any byte of the file. This approach has many advantages, but most files are logically considered as a sequence of records (some of which may be empty). How then do we create this structure and access our logical records?

Record structure

Creating the structure is quite simple. You need to decide what information you want to hold and the order in which you want store it. In the first example, for instance, we have two items of information (fields) per logical record; the name and the remarks. The name can be a maximum of 30 characters long and the remarks a maximum of 50 characters. So our logical record has two fields, one 30 characters long and the other 50 characters long. When the name string is written to disk it will be terminated by a CR - and so will the remarks string. So each record will be a maximum of 82 characters long.

We haven't finished yet, however. We need to be able to tell whether any one record is 'live' or empty (or deleted). To do this we need an extra byte at the start of each record which we set to one value for 'empty' and another for 'live'. In all the examples we use 0 to indicate 'empty' and NOT 0 to indicate 'live'. We are writing character data to the file so we could use the first byte of the name string as the indicator because the lowest ASCII code we will be storing is 32 (space). You can't do this for mixed data files because this byte could hold a data value of zero. Because of this, we have chosen to use an additional byte for the indicator in all the examples.
Our logical record thus consists of:

- 1 indicator byte
- 31 bytes for the name
- 51 bytes for the remarks

Thus the maximum amount of data in each record is 83 bytes. Because we cannot tell in advance how big each record needs to be (and we may want to change it later), we must assume that ALL the records will be this length. Since most of the records will be smaller than this, we are going to waste quite a lot of space in our random access file, but this is the penalty we pay for convenience and comparative simplicity.

When we write the data to the file, we could insist that each field was treated as a fixed length field by packing each string out with spaces to make it the 'correct' length. This would force each field to start at its 'proper' byte within the record. We don't need to do this, however, because we aren't going to randomly access the fields within the record; we know the order of the fields within the record and we are going to read them sequentially into appropriately named variables. We can write the fields to the file with each field following on immediately behind the previous one. All the 'spare' room is now left at the end of the record and not split up at the end of each field.

**Accessing the records**

In order to access any particular record, you need to set PTR# to the first byte of that record. Remember, you can't tell *BBC BASIC for Windows* that you want 'record 5', because it knows nothing of your file and record structure. You need to calculate the position of the first byte of 'record 5' and set PTR# to this value.

To start with, let's call the first record on the file 'record zero', the second record 'record 1', the third record 'record 2', etc. The first byte of 'record zero' is at byte zero on the file. The first byte of 'record 1' is at byte 83 on the file. The first byte of 'record 2' is at byte 166 (2*83) on the file. And so on. So, the start point of any record can be calculated by:

\[
\text{first\_byte} = 83 \times \text{record\_number}
\]

Now, we need to set PTR# to the position of this byte in order to access the record.
If the record number was held in 'recno' and the channel number in 'fnum', we could do this directly by:

\[
\text{PTR#fnum}=83*\text{recno}
\]

However, we may want to do this in several places in the program so it would be better to define and use a function to set PTR# as illustrated below.

\[
\text{PTR#fnum}=\text{FN_ptr(-recno)}
\]

\[
\text{DEF FN_ptr(record)=83*record}
\]

Whilst the computer is quite happy with the first record being 'record zero', us mere humans find it a little confusing. What we need is to be able to call the first record 'record 1', etc. We could do this without altering the function which calculates the start position of each record, but we would waste the space allocated to 'record 0' since we would never use it. We want to call it 'record 1' and the program wants to call it 'record 0'. We can change the function to cater for this. If we subtract 1 from the record number before we multiply it by the record length, we will get the result we want. Record 1 will start at byte zero, record 2 will start at byte 83, etc. Our function now looks like this:

\[
\text{DEF FN_ptr(record)=83*(record-1)}
\]

In our example so far we have used a record length of 83. If we replace this with a variable 'rec_len' we have a general function which we can use to calculate the start position of any record in the file in any program (you will need to set rec_len to the appropriate value at the start of the program). The function now becomes:

\[
\text{DEF FN_ptr(record)=rec_len*(record-1)}
\]

We use this function (or something very similar to it) in the following three example programs using random access files.

**Simple random access file**

```plaintext
REM  F-RAND1
;```

Page 572
REM VERY SIMPLE RANDOM ACCESS PROGRAM
:
REM This program maintains a random access file of names
REM and remarks. There is room for a maximum of 20
REM entries. Each name can be up to a maximum of 30
REM characters long and each remark up to 50 characters.
REM The first byte of the record is set non zero (in fact
REM &FF) if there is a record present. This gives a
REM maximum record length of 1+31+51=83. (Don't forget
REM the CRs.)
:
OSCLI "CD "+@usr$+""
:
bell$=CHR$(7)
temp$=STRING$(50," ")
maxrec=20
rec_len=83
ans$=""
CLS
WIDTH 0
fnum=OPENUP "RANDONE.DAT"
IF fnum=0 fnum=FN_setup("RANDONE.DAT")
REPEAT
  REPEAT
    INPUT "Enter record number: "ans$
    IF ans$="0" CLOSE#fnum:CLS:END
    IF ans$="" record=record+1 ELSE record=VAL(ans$)
    IF record<1 OR record>maxrec PRINT bell$;
    UNTIL record>0 AND record<=maxrec
    PTR#fnum=FN_ptr(record)
    PROC_display
    INPUT "Do you wish to change this record",ans$
    PTR#fnum=FN_ptr(record)
    IF FN_test(ans$) PROC_modify
  UNTIL FALSE
END
:
DEF FN_test(A$)=LEFT$(A$,1)="Y" OR LEFT$(A$,1)="y"
:
DEF FN_ptr(record)=rec_len*(record-1)
REM This makes record 1 start at PTR# = 0
:
DEF PROC_display
PRINT "Record number ";record'
flag=BGET#fnum
IF flag=0 PROC_clear:ENDPROC
INPUT#fnum,name$,remark$
PRINT name$;" ";remark
ENDPROC
:
DEF PROC_clear
PRINT "Record empty"
name$=""
remark$=""
ENDPROC :
:
DEF PROC_modify
PRINT "(Enter <RETURN> for no change or DELETE to delete)"
INPUT "Name ",temp$
temp$=LEFT$(temp$,30)
IF temp$<>"" name$=temp$
INPUT "Remark ",temp$
temp$=LEFT$(temp$,50)
IF temp$<>"" remark$=temp$
INPUT "Confirm update record",ans$
IF NOT FN_test(ans$) ENDPROC
IF name$="DELETE" BPUT#fnum,0:ENDPROC
BPUT#fnum,255
PRINT#fnum,name$,remark$
ENDPROC :
:
DEF FN_setup(fname$)
PRINT "Setting up the database file"
fnum=OPENOUT(fname$)
FOR record=1 TO maxrec
   BPUT#fnum,0
   PTR#fnum=FN_ptr(record)
NEXT
=fnum

**Simple random access database**

The second program in this sub-section expands the previous program into a simple, but quite versatile, database program. A setup procedure has been added which allows you to specify the file name. If it is a new file, you are then allowed to specify the number of records and the number, name and size of the fields you wish to use. This information is stored at the start of the file. If the file already exists this data is read from the records at the beginning of the file. The function for calculating the start position of each record is modified to take into account the room used at the front of the file to store information about the database.

REM F-RAND2
REM SIMPLE DATABASE PROGRAM
REM Written by R T Russell  Jan 1983
REM Modified for BBCBASIC(86) by Doug Mounter Dec 1985 :
REM This is a simple database program. You are asked for
REM the name of the file you wish to use. If the file
REM does not already exist, you are asked to specify the
REM number and format of the records. If the file does
REM already exist, the file specification is read from
REM the file.
;
@%=10
bell$=CHR$(7)
CLS
WIDTH 0
INPUT "Enter the filename of the data file: "filename$ fnum=OPNUP(filename$)
IF fnum=0 fnum=FN_setup(filename$) ELSE PROC_readgen
PRINT
:
REPEAT
  REPEAT
    INPUT "Enter record number: "ans$
    IF ans$="0" CLOSE#fnum:CLS:END
    IF ans$="" record=record+1 ELSE record=VAL(ans$)
    IF record<1 OR record>maxrec PRINT bell$;
    UNTIL record>0 AND record<=maxrec
    PTR#fnum=FN_ptr(record)
    PROC_display
    INPUT "Do you wish to change this record",ans$
    PTR#fnum=FN_ptr(record)
    IF FN_test(ans$) PROC_modify
  UNTIL FALSE
END
:
DEF FN_test(A$) =LEFT$(A$,1)="Y" OR LEFT$(A$,1)="y"
:
DEF FN_ptr(record)=base+rec_len*(record-1)
:
DEF FN_setup(filename$)
PRINT "New file."
  fnum=OPNOUT(filename$)
  REPEAT
    INPUT "Enter the number of records (max 1000): "maxrec
    UNTIL maxrec>0 AND maxrec<1001
  REPEAT
    INPUT "Enter number of fields per record (max 20): "fields
    UNTIL fields>0 AND fields<21
    DIM title$(fields),size(fields),A$(fields)
    FOR field=1 TO fields
      PRINT "Enter title of field number ";field;": "
      INPUT ""title$(field)
      PRINT
      REPEAT
        INPUT "Maximum size of field (characters)",size(field)
UNTIL size(field)>0 AND size(field)<256
NEXT field
rec_len=1
PRINT#fnum,maxrec,fields
FOR field=1 TO fields
  PRINT#fnum,title$(field),size(field)
  rec_len=rec_len+size(field)+1
NEXT field
base=PTR#fnum :
FOR record=1 TO maxrec
  PTR#fnum=FN_ptr(record)
  BPUT#fnum,0
NEXT
=fnum :
:
DEF PROC_readgen
rec_len=1
INPUT#fnum,maxrec,fields
DIM title$(fields),size(fields),A$(fields)
FOR field=1 TO fields
  INPUT#fnum,title$(field),size(field)
  rec_len=rec_len+size(field)+1
NEXT field
base=PTR#fnum ENDPROC :
:
DEF PROC_display
PRINT "Record number ";record
flag=BGET#fnum IF flag=0 PROC_clear:ENDPROC
FOR field=1 TO fields
  INPUT#fnum,A$(field)
  PRINT title$(field); " ";A$(field)
NEXT field
ENDPROC :
:
DEF PROC_clear
FOR field=1 TO fields
  A$(field)="
NEXT
ENDPROC :
:
DEF PROC_modify
PRINT "(Enter <RETURN> for no change)"
FOR field=1 TO fields
  REPEAT
    PRINT title$(field); ";
    INPUT LINE ""A$"
    IF A$="" PRINT TAB(POS,VPOS-1)title$(field); " ";A$(field)
  REM TAB(POS,VPOS-1) moves the cursor up 1 line
UNTIL LEN(A$)<=size(field)
  IF A$="" A$(field)=A$
NEXT field
INPUT "Confirm update record",ans$
IF NOT FN_test(ans$) ENDPROC
IF A$(1)="DELETE" BPUT#fnum,0:ENDPROC
BPUT#fnum,255
FOR field=1 TO fields
  PRINT#fnum,A$(field)
NEXT field
ENDPROC

Random access inventory program

The final example in this sub-section is a full-blown inventory program. Rather than go through all its aspects at the start, they are discussed at the appropriate point in the listing (these comments are not, of course, part of the program).

REM F-RAND3
:REM Written by Doug Mounter - Jan 1982
REM Modified for BBCBASIC(86) Dec 1985
:REM EXAMPLE OF A RANDOM ACCESS FILE
:REM This is a simple inventory program. It uses the
REM item's part number as the key and stores:
REM The item description - character max length 30
REM The quantity in stock - numeric
REM The re-order level - numeric
REM The unit price - numeric
REM In addition, the first byte of the record is used
REM as a valid data flag. Set to 0 if empty, D if the
REM record has been deleted or V if the record is
REM valid.
REM This gives a MAX record length of 47 bytes
REM (Don't forget the CR after the string)
:PROC_initialise
  inventory=FN_open("INVENTRY.DAT")

The following section of code is the command loop. You are offered a choice of functions until you eventually select function 0. The CASE statement is used for menu selection processing:

REPEAT
  CLS
This is the data entry function. You can delete or amend an entry or enter a new one. Have a look at the definition of FN_getrec for an explanation of the ASC "V" in its parameters:

REM AMEND/CREATE AN ENTRY
DEF PROCcreateentry
REPEAT
  CLS
  PRINT "AMEND/CREATE"
  partno=FN_getpartno
  flag=FN_getrec(partno,ASC"V")
  PROC_display(flag)
  PRINT"Do you wish to ";
  IF flag PRINT "change this entry ? "; ELSE PRINT "enter data ? ";
  IF (GET AND &5F)<ASC"N" flag=FN_amend(partno):PROC_cteos
  PROC_write(partno,flag,type)
  PRINT bell$;"Do you wish to amend/create another record ? ";
  UNTIL (GET AND &5F)=ASC"N"
ENDPROC
This subroutine allows you to look at a record without the ability to change or delete it:

REM DISPLAY AN ENTRY
DEF PROCdisplaypart
REPEAT
   CLS
   PRINT "DISPLAY"
   partno=FN_getpartno
   flag=FN_getrec(partno,ASC"V")
   PROC_display(flag)
   PRINT '
   PRINT "Do you wish to view another record ? ";
UNTIL (GET AND &5F)=ASC"N"
ENDPROC

The purpose of this subroutine is to allow you to update the stock level without having to amend the rest of the record:

REM CHANGE THE STOCK LEVEL FOR ONE PART
DEF PROCchangepart
REPEAT
   CLS
   PRINT "CHANGE STOCK"
   partno=FN_getpartno
   flag=FN_getrec(partno,ASC"V")
   REPEAT
      PROC_display(flag)
      PROC_cteos
      REPEAT
         PRINT TAB(0,12);:PROC_cteol
         INPUT "What is the stock change ? " temp$
         change=VAL(temp$)
         UNTIL INT(change)=change AND stock+change>=0
         IF temp$="" THEN
            flag=FALSE
         ELSE
            stock=stock+change
            PROC_display(flag)
         PRINT"Is this correct ? ";
         temp$=GET$
         ENDIF
      UNTIL NOT flag OR temp$="Y" OR temp$="y"
   PROC_write(partno,flag,ASC"V")
   PRINT return$;bell$;
   PRINT "Do you want to update any more stock levels ? ";
UNTIL (GET AND &5F)=ASC"N"
ENDPROC
This subroutine goes through the file in stock number order and lists all those items where the current stock is below the reorder level. You can interrupt the process at any time by pushing a key:

```
REM DISPLAY ITEMS BELOW REORDER LEVEL
DEF PROC reorder
partno=1
REPEAT
    CLS
    PRINT "ITEMS BELOW REORDER LEVEL"
    line_count=2
    REPEAT
        flag=FN_getrec(partno, ASC"V")
        IF (flag AND stock<reord) THEN
            PRINT "Part Number ";partno
            PRINT desc$;"  Stock ";stock;"  Reorder Level ";reord
            PRINT
            line_count=line_count+3
        ENDIF
        partno=partno+1
        temp$=INKEY$(0)
    UNTIL partno>maxpartno OR line_count>20 OR temp$<""
    PRINT TAB(0,23);bell$;"Press any key to continue or E to end ";temp$=GET$
    UNTIL partno>maxpartno OR temp$="E" OR temp$="e"
    partno=0
ENDPROC
```

Deleted entries are not actually removed from the file, just marked as deleted. This subroutine makes it possible for you to correct the mistake you made by deleting data you really wanted. If you have never used this type of program seriously, you won't believe how useful this is:

```
REM RECOVER A DELETED ENTRY
DEF PROC recover
REPEAT
    CLS
    PRINT "RECOVER DELETED RECORDS"
    partno=FN_getpartno
    flag=FN_getrec(partno, ASC"D")
    PROC_display(flag)
    PRINT
    IF flag THEN
        PRINT "If you wish to recover this entry type Y ";temp$=GET$
        IF temp$="Y" OR temp$="y" PROC_write(partno,flag,ASC"V")
    ENDIF
    PRINT return$;bell$;"Do you wish to recover another record ? ";
    UNTIL (GET AND &5F)=ASC"N"
```
This subroutine lists all the deleted entries so you can check you really don't want the data:

REM LIST DELETED ENTRIES
DEF PROClistdeleted
partno=1
REPEAT
  CLS
  PRINT "DELETED ITEMS"
  line_count=2
  REPEAT
    flag=FN_getrec(partno,ASC"D")
    IF flag THEN
      PRINT "Part Number ";partno
      PRINT "Description ";desc$
      line_count=line_count+3
    ENDIF
    partno=partno+1
    temp$=INKEY$(0)
  UNTIL partno>maxpartno OR line_count>20 OR temp$<>"
  PRINT TAB(0,23);bell$;"Press any key to continue or E to end ";
  UNTIL partno>maxpartno OR (GET AND &5F)=ASC"E"
partno=0
ENDPROC

REM REINITIALISE THE INVENTORY DATA FILE
DEF PROCnew
CLS
PRINT TAB(0,3);bell$;"Are you sure you want to set up a new inventory?"
PRINT "You will DESTROY ALL THE DATA YOU HAVE ACCUMULATED so far."
PRINT "If you are SURE you want to do it, enter YES"
INPUT "Otherwise, just hit Return ",temp$
IF temp$="YES" PROC_setup(inventory)
ENDPROC

This is where all the variables that you usually write as CHR$ (#) go. Then you can find them if you want to change them:

REM INITIALISE ALL THE VARIOUS PRESETS ETC :
DEF PROC_initialise
CLS
bell$=CHR$(7)
return$=CHR$(13)
rec_length=47
partno=0
It's a new file, so we won't go through the warning bit.

REM SET UP THE FILE
:REM Ask for the maximum part number required, write
REM it as the first record and then write 0 in to
REM the first byte of every record.
:DEF PROC_setup(fnum)
REPEAT
PRINT TAB(0,12);bell$;:PROC_cteos
INPUT "What is the highest part number required (Max 4999) ",maxpartno
UNTIL maxpartno>0 AND maxpartno<5000 AND INT(maxpartno)=maxpartno
PTR#fnum=0
PRINT#fnum,maxpartno
FOR partno=1 TO maxpartno
  PTR#fnum=FN_ptr(partno)
  BPUT#fnum,0
NEXT
partno=0
ENDPROC

Ask for the required part number. If a null is entered, make the next part number one more than the last:

REM GET AND RETURN THE REQUIRED PART NUMBER
:DEF FN_getpartno
REPEAT
   PRINT TAB(0,5);bell$;:PROC_cteos
   PRINT "Enter a Part Number Between 1 and ";maxpartno
   IF partno<>maxpartno THEN
      PRINT "The Next Part Number is ";partno+1;
      PRINT " Just hit Return to get this"
   ENDIF
   INPUT "What is the Part Number You Want ",partno$
   IF partno$<>"" THEN
      partno=VAL(partno$)
   ELSE
      IF partno=maxpartno partno=0 ELSE partno=partno+1
   ENDIF
   PRINT TAB(35,9);partno;:PROC_cteol
   UNTIL partno>0 AND partno<maxpartno+1 AND INT(partno)=partno

REM GET THE RECORD FOR THE PART NUMBER
REM Return TRUE if the record exists and FALSE if not
REM If the record does not exist, load desc$ with "No Record"
REM The remainder of the record is set to 0
DEF FN_getrec(partno,type)
   stock=0
   reord=0
   price=0
   PTR#inventry=FN_ptr(partno)
   test=BGET#inventry
   IF test=0 desc$="No Record": =FALSE
   IF test<>type THEN
      IF type=86 desc$="Record Deleted" ELSE desc$="Record Exists"
      =FALSE
   ENDIF
   INPUT#inventry,desc$
   INPUT#inventry,stock,reord,price
   =TRUE

Part numbers run from 1 up. The record for part number 1 starts at byte 5 of the file. The start position could have been calculated as (part-no -1) *record_length + 5. The expression below works out to the same thing, but it executes quicker:

REM CALCULATE THE VALUE OF PTR FOR THIS RECORD
DEF FN_ptr(partno)=partno*rec_length+5-rec_length

This function amends the record as required and returns with flag=TRUE if any amendment has taken place. It also sets the record type indicator (valid deleted or no record) to ASC"V" or ASC"D" as appropriate:
REM AMEND THE RECORD
:
DEF FN_amend(partno)
PRINT return$,:PROC_cteol:PRINT TAB(0,4); PRINT "Please Complete the Details for Part Number ";partno
PRINT "Just hit Return to leave the entry as it is"
flag=FALSE
type=ASC"V"
INPUT "Description - Max 30 Chars " temp$
IF temp$="DELETE" type=ASC"D" =TRUE
temp$=LEFT$(temp$,30)
IF temp$<>"" desc$=temp$:flag=TRUE
IF desc$="No Record" OR desc$="Record Deleted" =FALSE
INPUT "Current Stock Level " temp$
IF temp$<>"" stock=VAL(temp$):flag=TRUE
INPUT "Reorder Level " temp$
IF temp$<>"" reord=VAL(temp$):flag=TRUE
INPUT "Unit Price " temp$
IF temp$<>"" price=VAL(temp$):flag=TRUE
=flag

Write the record to the file if necessary (flag=TRUE):

REM WRITE THE RECORD
:
DEF PROC_write(partno,flag,type)
IF NOT flag ENDPROC
PTR#inventry=FN_ptr(partno)
BPUT#inventry,type
PRINT#inventry,desc$,stock,reord,price
ENDPROC

Print the record details to the screen. If the record is not of the required type (V or D) or it does not exist, stop after printing the description. The description holds "Record Exists" or "Record Deleted" or valid data as set by FN_getrec:

REM DISPLAY THE RECORD DETAILS
:
DEF PROC_display(flag)
PRINT TAB(0,5),:PROC_cteoes
PRINT "Part Number ";partno
PRINT "Description ";desc$
IF NOT flag ENDPROC
PRINT "Current Stock Level ";stock
PRINT "Reorder Level ";reord
PRINT "Unit Price ";price
ENDPROC
The following two procedures rely on there being 80 characters per line and 25 lines (for example when MODE 3 or 16 is selected):

REM There are no 'native' clear to end of line/screen
REM vdu procedures. The following 2 procedures clear to
REM the end of the line/screen.
REM
DEF PROC_cteol
LOCAL x,y
x=POS:y=VPOS
REPEAT VDU 32 : UNTIL POS=0
PRINT TAB(x,y);
ENDPROC

DEF PROC_cteos
LOCAL x,y
x=POS:y=VPOS
WHILE VPOS<24 VDU 32 : ENDMETHOD
REPEAT VDU 32 : UNTIL POS=0
PRINT TAB(x,y);
ENDPROC
Indexed data files

Deficiencies of random access files

As you will see if you examine a random file, a lot of space is wasted. This is because all the records must be allocated the same amount of space, otherwise you could not calculate where the record started. For large data files, over 50% of the space can be wasted. Under these circumstances it is possible to save space by using two files, one as an index to the other. In order for this to work efficiently, you must have complete control over the file pointer. Not many version of BASIC allow this control, but it is quite simple with BBC BASIC.

The address book program

The final program is an example of an indexed file. It is a computer implementation of the address book discussed way back at the beginning of these notes. Two files are used, one as an index to the other. Both are serial and no space is wasted between records.

File organisation

The files are organised as shown below:

<table>
<thead>
<tr>
<th>NAME .NDX</th>
<th>(index file)</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxrec</td>
<td>5 bytes</td>
</tr>
<tr>
<td>length</td>
<td>5 bytes</td>
</tr>
<tr>
<td>index$(1)</td>
<td>1 to 31 bytes</td>
</tr>
<tr>
<td>index(1)</td>
<td>5 bytes</td>
</tr>
<tr>
<td>index$(2)</td>
<td>1 to 31 bytes</td>
</tr>
<tr>
<td>index(2)</td>
<td>5 bytes</td>
</tr>
<tr>
<td>etc.</td>
<td></td>
</tr>
</tbody>
</table>

Where index(n) points to a record in the data file as follows:

<table>
<thead>
<tr>
<th>ADDRESS .DTA</th>
<th>(data file)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phone Num</td>
<td>1 to 31 bytes</td>
</tr>
<tr>
<td>Address 1</td>
<td>1 to 31 bytes</td>
</tr>
<tr>
<td>Address 2</td>
<td>1 to 31 bytes</td>
</tr>
<tr>
<td>Address 3</td>
<td>1 to 31 bytes</td>
</tr>
<tr>
<td>Address 4</td>
<td>1 to 31 bytes</td>
</tr>
<tr>
<td>Post Code</td>
<td>1 to 31 bytes</td>
</tr>
</tbody>
</table>
maxrec Is the maximum number of records permitted in this file. The practical limit is governed by the amount of memory available for the index arrays which are held in memory. If you want to write a disk access and sort program for the index - the best of luck. And please can I have a copy?

length Is the number of entries in the index.

index(n) Is the value of PTR# datanum just prior to the first byte of the data for this entry being written to it. In the Random File examples this value was calculated and it increased by a constant amount for every record.

Program organisation

The example looks horribly long and complicated. However the actual file handling bits are quite simple. The rest is, as usual, required for tidy input and output of data. The meat of the program is in the procedures and functions for putting and deleting index entries and finding the right place in the index. The latter uses a routine called a 'binary chop' (you could get arrested for that). This looks simple, and it is - when it works. If you are interested there is a flow chart and a brief explanation of how it works at the end of these notes. For the faithful, just use it. It takes considerably less time than any other method to search an ordered list.

The index

The index is read into memory at the start and written back at the end. In memory, it consists of two arrays called index$() and index(). Alternatively an array of structures could be used.

Indexed database example

REM F-INDEX
REM EXAMPLE OF AN INDEXED FILE
:
REM Written by Doug Mounter - Feb 1982
REM Modified for BBCBASIC(86) - Dec 1985
:
REM This is a simple address book filing system. It will
REM accept names, telephone numbers and addresses and store
REM them in a file called ADDRESS.DTA. The index is in REM name order and is kept in a file called NAME.NDX.
REM All the fields are character and the maximum length REM of any field is 30.
:
PROC_initialise
PROC_open_files
ON ERROR IF ERR<>17 PRINT:REPORT:PRINT" At line ";ERL:END
REPEAT
CLS
PRINT TAB(5,3);"If you want to: - "'
PRINT TAB(10);"End This Session";TAB(55);"Type 0"
PRINT TAB(10);"Enter Data";TAB(55);"Type 1"
PRINT TAB(10);"Search For/Delete an Entry";TAB(55);"Type 2"
PRINT TAB(10);"List in Alphabetical Order";TAB(55);"Type 3"
PRINT TAB(10);"Reorganize the Data File and Index";TAB(55);"Type 4"
REPEAT
PRINT TAB(5,11);
PRINT "Please enter the appropriate number (0 to 4) ";
function$=GET$
PRINT return$;:PROC_cteol
UNTIL function$>"/" AND function$<"5"
function=VAL(function$)
PRINT TAB(54,function$+5);"<====<"
ON function PROC_enter,PROC_search,PROC_list,PROC_reorg ELSE
UNTIL function=0
CLS
PROC_close_files
*ESC ON
PRINT "Address Book Files Closed"'
END :
:
REM ENTER DATA :
:
DEF PROC_enter
flag=TRUE
temp$=""
i=1
REPEAT
REPEAT
PROC_get_data
CASE TRUE OF
WHEN length=maxrec OR data$(1)="": flag=FALSE
WHEN data$(1)="+" OR data$(1)="-": PROC_message("Bad Data")
OTHERWISE:
i=FN_find_place(0,data$(1))
IF i>0 PROC_message("Duplicate Record")
PRINT ",Is this data correct ? ";
temp$=FN_yesno
IF temp$="N" PROC_message("Data NOT Accepted")
ENDCASE
UNTIL NOT flag OR temp$<!"N"
PROC_cteos
IF flag THEN
PROC_put_index(i, data$(1), PTR#datanum)
 FOR i=2 TO 7
 PRINT#datanum, data$(i)
 NEXT
 ENDIF
 UNTIL NOT flag
 ENDP
:
:
REM SEARCH FOR AN ENTRY
:
DEF PROC_search
 i=0
 REPEAT
 PRINT TAB(0,11);:PROC_cteol
 INPUT "What name do you want to look for ",name$
 IF name$<>"" THEN
   IF name$="DELETE" PROC_delete(i) ELSE i=FN_display(i,name$)
 ENDIF
 UNTIL name$=""
 ENDP
:
:
REM LIST IN ALPHABETICAL ORDER
:
DEF PROC_list
 entry=1
 REPEAT
 CLS
 line_count=0
 REPEAT
 PRINT TAB(0, line_count);
 PROC_read_data(entry)
 PROC_print_data
 entry=entry+1
 line_count=line_count+8
 temp$=INKEY$(0)
 UNTIL entry>length OR line_count>16 OR temp$<>""
 PROC_message("Push any key to continue or E to end ")
 UNTIL entry>length OR (GET AND &5F)=ASC"E"
 ENDP
:
:
REM REORGANIZE THE DATA FILE AND INDEX
:
DEF PROC_reorg
 entry=1
 PRINT TAB(0,13);"Reorganizing the Data File "
 newdata=OPENOUT"ADDRESS.BAK"
 REPEAT
 PROC_read_data(entry)
 index(entry)=PTR#newdata
 FOR i=2 TO 7
   PRINT#newdata, data$(i)
 NEXT
entry=entry+1
UNTIL entry>length
CLOSE#newdata : CLOSE#datanum
PRINT "Re-naming the Data File" '
*REN ADDRESS.BAK ADDRESS.$$'
PRINT "**";
*REN ADDRESS.DTA ADDRESS.BAK
PRINT "**";
*REN ADDRESS.$$ ADDRESS.DTA
PRINT "**";
 datanum=OPENUP "ADDRESS.DTA"
ENDPROC
:
:
REM INITIALISE VARIABLES AND ARRAYS
:
DEF PROC_initialise
CLS
*ESC OFF
esc$=CHR$(27)
bell$=CHR$(7)
return$=CHR$(13)
maxrec=100
OSCLI "CD """"+@usr$+""
:
REM The maximum record number, maxrec, is read in
REM PROC_read_index if the file already exists.
:
DIM message$(7)
FOR i=1 TO 7
  READ message$(i)
NEXT
DATA Name,Phone Number,Address,-- " --,-- " --,-- " --,Post Code
:
DIM data$(7)
FOR i=1 TO 7
  data$(i)=STRING$(30," ")
NEXT
temp$=STRING$(255," ")
temp$=
:
ENDPROC
:
:
REM OPEN THE FILES
:
DEF PROC_open_files
indexnum=OPENUP"NAME.NDX"
datanum=OPENUP"ADDRESS.DTA"
IF indexnum=0 OR datanum=0 PROC_setup ELSE PROC_read_index
PTR#datanum=EXT#datanum
ENDPROC
:
:
REM SET UP NEW INDEX AND DATA FILES
DEF PROC_setup
CLS
PRINT TAB(0,13);"Setting Up Address Book"
indexnum=OPENOUT"NAME.NDX"
datanum=OPENOUT"ADDRESS.DTA"
length=0
PRINT#indexnum,maxrec,length
CLOSE#indexnum
DIM index$(maxrec+1),index(maxrec+1)
index$(0)=""
index(0)=0
index$(1)=CHR$(&FF)
index(1)=0
ENDPROC
:
:
REM READ INDEX AND LENGTH OF DATA FILE
:
DEF PROC_read_index
CLS
INPUT#indexnum,maxrec,length
DIM index$(maxrec+1), index(maxrec+1)
index$(0)=""
index(0)=0
FOR i=1 TO length
   INPUT#indexnum,index$(i),index(i)
NEXT
CLOSE#indexnum
index$(length+1)=CHR$(&FF)
index(length+1)=0
ENDPROC
:
:
REM WRITE INDEX AND CLOSE FILES
:
DEF PROC_close_files
indexnum=OPENOUT"NAME.NDX"
PRINT#indexnum,maxrec,length
FOR i=1 TO length
   PRINT#indexnum,index$(i),index(i)
NEXT
CLOSE#0
ENDPROC
:
:
REM WRITE A MESSAGE AT LINE 23
:
DEF PROC_message(line$)
LOCAL x,y
x=POS
y=VPOS
PRINT TAB(0,23);:PROC_cteol:PRINT bell$;line$;
PRINT TAB(x,y);
ENDPROC
REM GET A Y/N ANSWER
DEF FN_yesno
LOCAL temp$
temp$=GET$
IF temp$="y" OR temp$="Y" ="Y"
IF temp$="n" OR temp$="N" ="N"
=

REM CLEAR 9 LINES FROM PRESENT POSITION
DEF PROC_clear9
LOCAL x,y,n
x=POS:y=VPOS:PRINT return$;
FOR n=1 TO 9 : REPEAT VDU 32 : UNTIL POS=0 : NEXT n
PRINT TAB(x,y);
ENDPROC

REM GET INPUT DATA - LIMIT TO 30 CHAR
DEF PROC_get_data
LOCAL i
PRINT TAB(0,13);
PROC_clear9
IF length=maxrec PROC_message("Address Book Full")
FOR i=1 TO 7
PRINT TAB(10);message$(i);TAB(25);
INPUT temp$
data$(i)=LEFT$(temp$,30)
IF data$(1)="" i=7
NEXT
ENDPROC

REM FIND AND DISPLAY THE REQUESTED DATA
DEF FN_display(i,name$)
PRINT TAB(0,12);:PROC_cteos
i=FN_find_place(i,name$)
IF i<0 PROC_message("Name Not Known - Next Highest Given")
PROC_read_data(i)
PRINT
PROC_print_data
=i

Move everything below the entry you want deleted up one and subtract 1 from the length:
REM DELETE THE ENTRY FROM THE INDEX
:
DEF PROC_delete(i)
INPUT "Are you SURE ", temp$
PRINT TAB(0, VPOS-1);: PROC_cteos
IF temp$<>"YES" ENDPROC
IF i<0 i=-i
FOR i=i TO length
    index$(i)=index$(i+1)
    index(i)=index(i+1)
NEXT
length=length-1
ENDPROC

Get the start of the position of the start of the data record for entry 'i' in the index and read it into the buffer array data$(()). Save the current value of the data file pointer on entry and restore it before leaving:

REM READ DATA FOR ENTRY i
:
DEF PROC_read_data(i)
PTRdata=PTR#datanum
IF i<0 i=-i
PTR#datanum=index(i)
data$(1)=index$(i)
FOR i=2 TO 7
    INPUT#datanum, data$(i)
NEXT
PTR#datanum=PTRdata
ENDPROC

REM PRINT data$(()) ON VDU
:
DEF PROC_print_data
LOCAL i
FOR i=1 TO 7
    IF data$(i)<="" PRINT TAB(10); message$(i); TAB(25); data$(i)
    IF data$(1)=CHR$(&FF) i=7
NEXT
ENDPROC

Move all the directory entries from position i onwards down the index (in fact you have to start at the end and work back). Slot the new entry in the gap made at position i and add 1 to the length:

REM PUT A NEW ENTRY IN INDEX AT POSITION i
:
DEF PROC_put_index(i, entry$, ptr)

LOCAL j
IF i<0 i=-i
FOR j=length+1 TO i STEP -1
  index$(j+1)=index$(j)
  index(j+1)=index(j)
NEXT
index$(i)=entry$
index(i)=ptr
length=length+1
ENDPROC

This function looks in the index for the string entry$. If it finds it it returns with i set to its position in the index. If not, i is set to minus the position of the next highest string (in other words, the position you wish to put the a new entry). Thus if a part of the index looked like:

(34) BERT
(35) FRED
(36) JOHN

and you entered with FRED, it would return 35. However if you entered with GEORGE, it would return -36.

The function consists of two parts. The first looks at the entry$ to see if it should just up or down the entry number by 1, taking account of wrap-around at the start and end of the index. The second part is the binary chop advertised with such telling wit in the introduction to indexed files. Since we enter this function with the entry pointer i set to its previous value, we must cater for a negative value.

REM FIND ENTRY IN INDEX OR PLACE TO PUT IT :
DEF FN_find_place(i,entry$)
LOCAL top,bottom
IF i<0 i=-i
IF entry$="+" AND i<length =i+1
IF entry$="+" AND i=length =1
IF entry$="-" AND i>1 =i-1
IF entry$="-" AND i<2 =length

Here, at last, **THE BINARY CHOP**

top=length+1
bottom=0
i=(top+1) DIV 2
IF entry$<>index$(i) i=FN_search(entry$)
REPEAT
  IF entry$=index$(i-1) i=i-1
UNTIL entry$<>index$(i-1)
IF entry$=index$(i) =i ELSE =-i
: : :
REM DO THE SEARCHING FOR FN_find_place
: :
DEF FN_search(entry$)
REPEAT
  IF entry$>index$(i) bottom=i ELSE top=i
    i=(top+bottom+1) DIV 2: REM round
  UNTIL entry$=index$(i) OR top=bottom+1
  =i

This bit moves the pointer up the index to the first of any duplicate entries.

The two following procedures rely on mode 3 or 7 being selected. They will not work properly if a graphics mode has been selected or if some characters on the screen have attributes set.

REM There are no 'native' clear to end of line/screen
REM vdu procedures. The following 2 procedures clear to
REM the end of the line/screen.
: :
DEF PROC_cteol
LOCAL x,y
x=POS:y=VPOS
REPEAT VDU 32 : UNTIL POS=0
PRINT TAB(x,y);
ENDPROC
: :
DEF PROC_cteos
LOCAL x,y
x=POS:y=VPOS
WHILE VPOS<24 VDU 32 : ENDWHILE
REPEAT VDU 32 : UNTIL POS=0
PRINT TAB(x,y);
ENDPROC

Well, that's it. Apart from the following notes on the binary chop you have read it all.
The Binary Chop

The quickest way to find an entry in an ORDERED list is not to search through it from start to end, but to continue splitting the list in two until you reach the entry you are looking for. You begin by setting one pointer to the bottom of the list, another to the top, and a third to mid-way between bottom and top. Then you compare the entry pointed to by this third pointer with the number you are searching for. If your number is bigger you make the bottom equal the pointer, if not make the top equal to it. Then you repeat the process.

Let's try searching the list of numbers below for the number 14.

```
bottom> (1) 3  Set bottom to the lowest position in the list, and top to the highest. Set the pointer to (top+bottom)/2. Is that entry 14? No it's more, so set top to the current value of pointer and repeat the process.
  (2) 6
  (3) 8
  (4) 14
pointer> (5) 19
  (6) 23
  (7) 34
  (8) 45
top> (9) 61
```

```
bottom> (1) 3  Set the pointer to (top+bottom)/2. Is that entry 14? No it's less, so set bottom to the current value of pointer and try again.
  (2) 6
pointer> (3) 8
  (4) 14
top> (5) 19
  (6) 23
  (7) 34
  (8) 45
  (9) 61
```
3. Set the pointer to \((\text{top} + \text{bottom})/2\). Is that entry 14? Yes, so exit with the pointer set to the position in the list of the number you are looking for.

As you can imagine, things are not always as simple as this carefully chosen example. You have to cater for the number not being there, and for the list being empty. There are a number of ways of doing this, but the easiest is to add two numbers of your choice to the list. Make the first entry the most negative number the computer can hold, and the last entry the most positive. This will prevent you ever trying to search outside the list. Preventing a perpetual loop when the number you want is not in the list is quite simple, just exit when 'top' is equal to 'bottom'+1. If you have not found the number by then, it's not in the list.

You can use this routine to add numbers to the list in order. If you can't find the number, you exit with the position it should go in the list. Just move all the numbers under it down one slot and put the new number in. This works just as well when the list is empty except for your two 'end markers'.

Have a look at the flow chart below and work through a couple of dry runs with a short list of numbers. You may think that it's not worth doing it this way and that a 'linear search' would be as quick. Try it with a list of 100 numbers. It should take you no more than 7 goes to find the number. The AVERAGE number of comparisons required for a linear search would be 50.
ENTRY = 'THING' BEING SEARCHED FOR

LENGTH = NO OF 'PROPER' ENTRIES IN THE ARRAY. THE FIRST 'PROPER' ENTRY IS IN ARRAY(1).

ENTRY > ARRAY(PONITER) ?

Y

ENTRY = ENTRY

N

TOP = POINTER

BOTTOM = POINTER

POINTER = (TOP + BOTTOM + 1) DIV 2

UNTIL TOP = BOTTOM + 1 OR ARRAY(PONITER) = ENTRY

Y

EXIT

N

REPEAT

EXIT

ARRAY(0) HOLDS MOST -VE NO.
ARRAY(LENGTH + 1) HOLDS MOST +VE
THIS ROUNDS UP.
(TOP + BOTTOM) DIV 2 WOULD ROUND DOWN
Introduction to the assembler

_BBC BASIC for Windows_ includes an 80386/80486 assembler (it also accepts some Pentium instructions). This assembler is similar to the 6502 assembler on the BBC Micro and the Z80 assembler in BBC BASIC(Z80) and it is entered in the same way. That is, '[' enters assembler mode and ']' exits assembler mode. Unlike the 6502 or Z80 assemblers, the 80x86 assembler attempts to detect multiply-defined labels. If a label is found to have an existing non-zero value during the first pass of an assembly (OPT 0, 1, 8, 9), a 'Multiple label ' error is reported (error code 3).

Assembler statements

An assembly language statement consists of three elements; an optional label, an instruction opcode and an operand. A comment may follow the operand field. If an instruction opcode follows a label they must be separated by at least one space. Similarly, the operand must also be separated from the instruction opcode by a space. Opcodes are not case sensitive.

Assembly language statements are terminated by a colon (:) or end of line (<RET>). When terminated by a colon, it is necessary to leave a space between the colon and a preceding segment register name otherwise it may be misinterpreted as a segment override. See the Segment Override sub-section for details.

Labels

Labels are defined by preceding them with a full stop (.). When the assembler encounters such a label, a numeric variable is created containing the current value of the Program Counter (P%). Such variables are accessible in the normal way outside of the assembler.

In the example shown later under the heading The assembly process, two labels are defined and used. Labels have the same rules as standard _BBC BASIC for Windows_ variable names; they should start with a letter and not start with a keyword.

Note that although the run-time assembler will allow you to use an _array element_ as a label, this is not valid BBC BASIC syntax and it is _not_ accepted by the compiler (cruncher). You _can_ use a _structure member_ as a label, so long as it starts with a letter.
Comments

You can insert comments into assembly language programs by preceding them with a semi-colon (;). In assembly language, a comment ends at the end of the statement. Thus, the following example will work (but it's a bit untidy):

```plaintext
[;start assembly language program
etc
MOV EAX,ECX ;In-line comment : POP EBX ;start add
JNZ loop ;Go back if not finished : RET ;Return
etc
;end assembly language program:]
```
Differences from Intel syntax

The assembler generally conforms to Intel assembly language syntax. However, there are a number of minor differences which are described below.

Jumps, calls and returns

Unconditional jumps, calls and returns are assumed to be within the current code segment. Short (8 bit displacement) jumps, far (inter segment) calls, far jumps and far returns must be explicitly specified by using the following mnemonics:

- **Short jump**: `JMP S` or `JMP SHORT`
- **Far call**: `CALLF` or `CALL FAR`
- **Far jump**: `JMPF` or `JMP FAR`
- **Far return**: `RETF`

Note that since *BBC BASIC for Windows* is a 32-bit program, and the assembler will normally be used to generate 32-bit code in a 'flat' address space, the segment size is $2^{32}$ bytes (4 Gbytes!). You are therefore most unlikely to want to perform inter-segment jumps or calls.

Conditional jumps are assumed to be short (8-bit displacement). Near conditional jumps (32-bit displacement) must be explicitly specified by adding the NEAR prefix, for example:

- `JZ NEAR dest`
- `JNC NEAR label`

Note that the LOOP and JECXZ instructions (and their variants) can use only 8-bit displacements. You must ensure that the destination is within range.

Memory operands

Memory operands must be placed in square brackets in order to distinguish them from immediate operands. For example,

- `MOV EAX, [store]`
will load the EAX register with the contents of memory location 'store'. However,

\[
\text{MOV EAX, store}
\]

will load the EAX register with the 32 bit value of BASIC variable 'store', i.e. the address of the memory location.

**String operations**

The string operations must have the data size (byte, word or double-word) explicitly specified in the instruction mnemonic as listed below.

- Compare memory - byte: `CMPSB`
- Compare memory - word: `CMPSW`
- Compare memory - double-word: `CMPSD`
- Compare AL (byte): `SCASB`
- Compare AX (word): `SCASW`
- Compare EAX (double-word): `SCASD`
- Load from memory - byte: `LODSB`
- Load from memory - word: `LODSW`
- Load from memory - double-word: `LODSD`
- Store to memory - byte: `STOSB`
- Store to memory - word: `STOSW`
- Store to memory - double-word: `STOSD`
- Move byte: `MOVSB`
- Move word: `MOVSW`
- Move double-word: `MOVSD`

**Segment override**

When segment overrides are necessary, they must always be entered explicitly. The assembler will not insert them automatically. For example,

\[
\text{MOV EAX, CS:[data]}
\]
will load the EAX register with the contents of the address 'data' in the code segment. Since *BBC BASIC for Windows* is a 32-bit program and the assembler will normally be used to generate 32-bit code in a 'flat' address space, segment overrides will very rarely be required.

When assembly language statements are separated by colons, it is necessary to leave a space between the colon and a preceding segment register name. If the space is missing, the assembler will misinterpret the colon as a segment override. For example,

```
PUSH CS:MOV EAX,0
```

will give rise to an error, but

```
PUSH CS :MOV EAX,0
```

will be accepted.

**Data size ambiguities**

Some assembly language instructions are ambiguous as to whether a byte, word or double-word value is to be acted upon. When this is so, an explicit 'byte ptr', 'word ptr' or 'dword ptr' operator must be used. These can be simplified to 'byte', 'word' or 'dword' respectively. For example:

```
INC BYTE PTR [EBX]
MOV WORD PTR [count],0
ADD DWORD [ESI],offset
```

If this operator is omitted, *BBC BASIC for Windows* will issue a 'Size needed ' error message (error code 2).

**Loop instructions**

The 'loop' instructions (loop, loope, loopne, loopnz, loopz) by default decrement (and test) the 32-bit ECX register. To specify that they should instead decrement the 16-bit CX register use the opcodes loopw, loopew, loopnew, loopnzw and loopzw.
For completeness, the opcodes loopd, looped, loopned, loopnzd and loopzd are also accepted; these behave in an identical fashion to the opcodes without the final 'd' (they use the ECX register).

```
LOOP label
LOPW label
LOPFD label
```

**Based-indexed operands**

The 16-bit composite based-indexed operands are only accepted in the preferred forms with the base register specified first. For example,

```
[bp+di], [bp+si], [bx+di], [bx+si]
```

are accepted, but

```
[di+bp], [si+bp], [di+bx], [si+bx]
```

are not.

This restriction does not apply to the 32-bit memory operands. For example, all the following are accepted:

```
[eax+2*ecx], [edx+ebx*4], [eax*3], [ebp+esi]
```

**Indexed memory operands**

Indexed memory operands with constant offsets are accepted in the following formats:

```
[index]+offset
[index+offset]
offset[index]
```

Where 'index' is an index or base register such as 'ebx', 'ebp+esi', etc, and 'offset' is a numeric expression.
Floating-point operands

Locations in the floating-point register stack are referred to as ST0, ST1, ST2, ST3, ST4, ST5, ST6 and ST7. When a stack operand is required it must always be explicitly specified.

In addition to dword (or dword ptr) the data-size modifiers qword (or qword ptr) and tbyte may also be specified. These refer to a 64-bit (double float or long long integer) or 80-bit (temporary float or 18-digit BCD) data size respectively. The only instructions which can take a tbyte operand are fbld, fbstp, fld and fstp.
Numeric and string constants

You can store constants within your assembly language program using the define byte (DB), define word (DW) and define double-word (DD) pseudo-operation commands. These will create 1 byte, 2 byte and 4 byte items respectively. Define byte (DB) may alternatively be followed by a string operand. In which case, the bytes comprising the string will be placed in memory at the current assembly location. As discussed later, this will be governed by P% or O% depending on the OPT value used.

Be careful if you use DB, DW or DD to define locations in which to store variable data rather than constants. On some modern processors writing data to a memory location in close proximity to the code can dramatically reduce execution speed (this is to support self-modifying code). If speed is important ensure that any data storage locations to which you write frequently are at least 2 Kbytes away from the code accessing them.

Define byte - DB

Byte constant

DB can be used to set one byte of memory to a particular value. For example,

```
.data DB 15
  DB 9
```

will set two consecutive bytes of memory to 15 and 9 (decimal). The address of the first byte will be stored in the variable 'data'.

String constant

DB can be used to load a string of ASCII characters into memory. For example,

```
JMPS continue; jump round the data
.string DB "This is a test message"
  DB &D
  .continue; and continue the process
```
will load the string 'This is a test message' followed by a carriage-return into memory. The address of the start of the message is loaded into the variable 'string'. This is equivalent to the following program segment:

```
JMPS continue; jump round the data
.string; leave assembly and load the string
$P%="This is a test message" REM starting at P%
P%=P%+LEN($P%)+1 REM adjust P% to next free byte
OPT opt%; reset OPT
.continue; and continue the program
```

**Define word - DW**

DW can be used to set two bytes of memory to a particular value. The first byte is set to the least significant byte of the number and the second to the most significant byte. For example,

```
.data DW &90F
```

will have the same result as the Byte constant example above.

**Define double-word - DD**

DD can be used to set four bytes of memory to a particular value. The first byte is set to the least significant byte of the number and the fourth to the most significant byte. For example,

```
.data DD &90F0D10
```

will have the same result as,

```
.data DB 16
DB 13 or DB &D
DB 15 DB &F
DB 9 DB &9
```

```
.data DB 610
```
Opcodes

The following opcodes are accepted by the assembler. Opcodes are not case-sensitive, they may be given in capitals or lower-case:

aaa   aad   aam   aas
adc   add   and   bound
bsf   bsr   bswap  bt
btc   btr   bts   call
cbw   cdq   clc   cld
cli   cmc   cmp   cmpsb
cmpsd cmpsw cmpxchg cpuid
cwd   cwde  daa   das
dec   div   enter  hlt
idiv  imul  in    inc
insb  insd  insw  int
into  invd  invlpg  iret
iretd jae   ja    jbe
jb    jc    je    jge
jg    jle   jl    jmp
jnae  jna   jnbe  jnb
jnc   jne   jnge  jng
jnle  jnl   jno   jnp
jns   jnz   jo    jpe
jpo   jp    js    jz
lahf  lds   lea   leave
les   lfs   lgs   lock
lodsb lodsd lodsw  loop
loope loopne loopnz  loopz
loopd looped loopned loopnzd
loopzd loopw loopew  loopnew
loopnzw loopzw lss   mov
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Instruction</th>
<th>Instruction</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>movsb</td>
<td>movsd</td>
<td>movsw</td>
<td>movsx</td>
</tr>
<tr>
<td>movzx</td>
<td>mul</td>
<td>neg</td>
<td>nop</td>
</tr>
<tr>
<td>not</td>
<td>or</td>
<td>out</td>
<td>outsb</td>
</tr>
<tr>
<td>outsd</td>
<td>outsw</td>
<td>pop</td>
<td>popa</td>
</tr>
<tr>
<td>popad</td>
<td>popf</td>
<td>popfd</td>
<td>push</td>
</tr>
<tr>
<td>pusha</td>
<td>pushad</td>
<td>pushf</td>
<td>pushfd</td>
</tr>
<tr>
<td>rcl</td>
<td>rcr</td>
<td>rdtsc</td>
<td>rep</td>
</tr>
<tr>
<td>repe</td>
<td>repne</td>
<td>repnz</td>
<td>repz</td>
</tr>
<tr>
<td>ret</td>
<td>retf</td>
<td>retn</td>
<td>rol</td>
</tr>
<tr>
<td>ror</td>
<td>sahf</td>
<td>sal</td>
<td>sar</td>
</tr>
<tr>
<td>sbb</td>
<td>scasb</td>
<td>scasd</td>
<td>scasw</td>
</tr>
<tr>
<td>setae</td>
<td>seta</td>
<td>setbe</td>
<td>setb</td>
</tr>
<tr>
<td>setc</td>
<td>sete</td>
<td>setge</td>
<td>setg</td>
</tr>
<tr>
<td>setle</td>
<td>setl</td>
<td>setnae</td>
<td>setna</td>
</tr>
<tr>
<td>setnbe</td>
<td>setnb</td>
<td>setnc</td>
<td>setne</td>
</tr>
<tr>
<td>setnge</td>
<td>setng</td>
<td>setnle</td>
<td>setnl</td>
</tr>
<tr>
<td>setno</td>
<td>setnp</td>
<td>setns</td>
<td>setnz</td>
</tr>
<tr>
<td>seto</td>
<td>setpe</td>
<td>setpo</td>
<td>setp</td>
</tr>
<tr>
<td>sets</td>
<td>setz</td>
<td>shl</td>
<td>shld</td>
</tr>
<tr>
<td>shr</td>
<td>shrd</td>
<td>stc</td>
<td>std</td>
</tr>
<tr>
<td>sti</td>
<td>stosb</td>
<td>stosd</td>
<td>stosw</td>
</tr>
<tr>
<td>sub</td>
<td>test</td>
<td>wait</td>
<td>wbinvd</td>
</tr>
<tr>
<td>xadd</td>
<td>xchg</td>
<td>xlat</td>
<td>xor</td>
</tr>
</tbody>
</table>
Floating-point opcodes

The following floating-point opcodes are accepted by the assembler:

f2xm1  fabs  fadd  faddp
fbld  fbstp  fchs  fclex
fcom  fcomi  fcomip  fcomp
fcompp  fcos  fdecstp  fdiv
fdivp  fdivr  fdivr p  ffree
fiadd  ficom  ficomp  fidiv
fidivr  fild  fimul  fincstp
finit  fist  fistp  fisub
fisubr  fld  fld1  fldl2e
fldl2t  fldlg2  fldln2  fldpi
fldz  fldcw  fldenv  fmul
fmulp  fncl lex  fninit  fnop
fnsave  fnstcw  fnstenv  fnstsw
fpatt n  fprem  fprem1  fptan
frndint  frstow  fsave  fscale
fsin  fsincos  fsqrt  fst
fstp  fstcw  fstenv  fstsw
fsub  fsubp  fsnbr  fsubrp
ftst  fucom  fucom p  fucomi
fucomip  fucompp  fxam  fxch
fxtract  fyl2x  fyl2xp1

Note that fcomi, fcomip, fucomi and fucomip are available in BBC BASIC for Windows version 6.00a or later only.
MMX opcodes

The following MMX opcodes are accepted by the assembler:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>emms</td>
<td></td>
</tr>
<tr>
<td>movq</td>
<td></td>
</tr>
<tr>
<td>paddb</td>
<td></td>
</tr>
<tr>
<td>paddsw</td>
<td></td>
</tr>
<tr>
<td>pandn</td>
<td></td>
</tr>
<tr>
<td>pcmpeqw</td>
<td></td>
</tr>
<tr>
<td>pcmpeqd</td>
<td></td>
</tr>
<tr>
<td>pcmptd</td>
<td></td>
</tr>
<tr>
<td>pmovmskb</td>
<td></td>
</tr>
<tr>
<td>por</td>
<td></td>
</tr>
<tr>
<td>pslld</td>
<td></td>
</tr>
<tr>
<td>psrlw</td>
<td></td>
</tr>
<tr>
<td>psubw</td>
<td></td>
</tr>
<tr>
<td>psubusb</td>
<td></td>
</tr>
<tr>
<td>pxor</td>
<td></td>
</tr>
<tr>
<td>maskmovq</td>
<td></td>
</tr>
<tr>
<td>movd</td>
<td></td>
</tr>
<tr>
<td>movntq</td>
<td></td>
</tr>
<tr>
<td>packssdw</td>
<td></td>
</tr>
<tr>
<td>packsswb</td>
<td></td>
</tr>
<tr>
<td>packuswb</td>
<td></td>
</tr>
<tr>
<td>padd</td>
<td></td>
</tr>
<tr>
<td>paddd</td>
<td></td>
</tr>
<tr>
<td>paddsb</td>
<td></td>
</tr>
<tr>
<td>paddusw</td>
<td></td>
</tr>
<tr>
<td>pand</td>
<td></td>
</tr>
<tr>
<td>pavgw</td>
<td></td>
</tr>
<tr>
<td>pcmpeqb</td>
<td></td>
</tr>
<tr>
<td>pcmpeqd</td>
<td></td>
</tr>
<tr>
<td>pcmptb</td>
<td></td>
</tr>
<tr>
<td>pcmptw</td>
<td></td>
</tr>
<tr>
<td>pextrw</td>
<td></td>
</tr>
<tr>
<td>pinsrw</td>
<td></td>
</tr>
<tr>
<td>pmaddwd</td>
<td></td>
</tr>
<tr>
<td>pmaxub</td>
<td></td>
</tr>
<tr>
<td>pminsw</td>
<td></td>
</tr>
<tr>
<td>pminub</td>
<td></td>
</tr>
<tr>
<td>pmulhuw</td>
<td></td>
</tr>
<tr>
<td>pmulhw</td>
<td></td>
</tr>
<tr>
<td>pmullw</td>
<td></td>
</tr>
<tr>
<td>psadbw</td>
<td></td>
</tr>
<tr>
<td>pshufw</td>
<td></td>
</tr>
<tr>
<td>psllw</td>
<td></td>
</tr>
<tr>
<td>psllq</td>
<td></td>
</tr>
<tr>
<td>psraw</td>
<td></td>
</tr>
<tr>
<td>psrad</td>
<td></td>
</tr>
<tr>
<td>psrlq</td>
<td></td>
</tr>
<tr>
<td>psubb</td>
<td></td>
</tr>
<tr>
<td>psubd</td>
<td></td>
</tr>
<tr>
<td>psubsb</td>
<td></td>
</tr>
<tr>
<td>psubsw</td>
<td></td>
</tr>
<tr>
<td>unpckhbw</td>
<td></td>
</tr>
<tr>
<td>unpckhwd</td>
<td></td>
</tr>
<tr>
<td>unpckhdq</td>
<td></td>
</tr>
<tr>
<td>unpcklbd</td>
<td></td>
</tr>
<tr>
<td>unpckldq</td>
<td></td>
</tr>
</tbody>
</table>

Note that pavgb, pavgw, pextrw, pinsrw, pmaxsw, pmaxub, pminsw, pminub, pmovmskb, pmulhuw, psadbw and pshufw are not strictly speaking MMX opcodes, as they were added with the **Streaming SIMD Extensions (SSE)**. However since they operate on the integer MMX registers they logically extend the MMX instruction set.
Using BASIC input/output

An assembly language program may access some of BASIC's input/output routines (e.g. the VDU drivers) by calling the following routines by name:

- `CALL "osbget"`; Read byte from file to AL, EBX contains channel number
- `CALL "osbput"`; Write byte from AL to file, EBX contains channel number
- `CALL "osrdch"`; Read keyboard character to AL
- `CALL "osasci"`; Write AL to the VDU drivers (plus LF if CR)
- `CALL "osnewl"`; Write LF, CR
- `CALL "oswrch"`; Write AL to the VDU drivers
- `CALL "osword"`; Read character dot pattern, EDX addresses buffer
- `CALL "osbyte"`; Read character at cursor position to AL
- `CALL "oscli"`; Various OS commands, EDX addresses string
- `CALL "oskey"`; Equivalent to INKEY, EAX contains timeout value
- `CALL "osline"`; Read a line from the console, EDX addresses buffer (DL=0)
- `CALL "oshut"`; Close a file, EBX = channel number
- `CALL "getptr"`; Read file pointer, EBX = channel number, result in EAX
- `CALL "setptr"`; Set file pointer, EBX = channel number, EAX = value
- `CALL "getext"`; Read file length, EBX = channel number, result in EAX
- `CALL "setext"`; Set file length, EBX = channel number, EAX = value
- `CALL "osopen"`; Open a file, EDX addresses filename, AL = 0 (read), 1 (create) or 2 (update), channel number returned in EAX

In **BBC BASIC for Windows version 6.00a or later** the EDX register contains the most-significant 32-bits of the file pointer or file size, in the case of the `getptr`, `setptr`, `getext` and `setext` routines.

All strings are CR-terminated. In the case of 'oskey' the carry flag is cleared if no key was pressed within the timeout period. In the case of 'osbget' the carry flag is cleared if at end-of-file.

Note that you should ensure that the **direction flag** is cleared before calling any of these routines. If in doubt, add a CLD instruction.

The following assembly-language program would clear the screen (text viewport):

```
.clrscn
CLD
MOV AL,12 ; VDU 12 is CLS
CALL "oswrch"
RET
```
Calling the Windows API

An assembly language program may call Windows™ API routines by name. For example, the following program would generate the system warning sound:

```
.beep
push 48 ; Put the parameter on the stack
call "MessageBeep"
ret
```

When passing multiple parameters you must be careful to push them in 'reverse order' so they end up in the correct sequence on the stack. So for example:

```
SYS "SetWindowPos", @hwnd%, 0, xpos%, ypos%, 0, 0, 5
```

would become in assembly language:

```
push 5
push 0
push 0
push ypos%
push xpos%
push 0
push @hwnd%
call "SetWindowPos"
```

Note that you should ensure that the direction flag is cleared before calling any Windows API function. If in doubt, add a CLD instruction.
Reserving memory

The program counter

Machine code instructions are assembled as if they were going to be placed in memory at the addresses specified by the program counter, P%. Their actual location in memory is determined by P% or O% depending on the value of OPT used. You must make sure that P% (or O%) is pointing to a free area of memory before your program begins assembly. In addition, you need to reserve the area of memory that your machine code program will use so that it is not overwritten at run time. You can reserve memory by using a special version of the DIM statement.

P% and O% are initialised to zero. Using the assembler without first setting P% or O%, as appropriate, is liable to crash BBC BASIC for Windows.

Using DIM to reserve memory

Using the special version of the DIM statement to reserve an area of memory is the simplest and safest way (see the keyword DIM for more details). For example:

```
DIM code% 20 : REM Note the absence of brackets
```

will reserve 21 bytes of code (byte 0 to byte 20) and load the variable 'code%' with the start address of the reserved area. You can then set P% (or O%) to the start of that area.

The example below reserves an area of memory 100 bytes long, sets P% to the first byte of the reserved area and sets L% to the end of the reserved area (as is required when bit 3 of OPT is set):

```
size% = 100
DIM code% size%-1
FOR opt% = 8 TO 10 STEP 2
   PROCassemble(opt%, code%, code%+size%)
NEXT opt%
...
DEF PROCassemble(opt%, P%, L%)
[OPT opt%]
...
```

Page 614
Efficient cache usage

All modern processors cache code and data memory for efficiency. Performance of assembly-language code can be seriously impaired if data is written to the same block of memory as that in which the code is executing, because it may cause the CPU repeatedly to reload the instruction cache (this is to ensure self-modifying-code works correctly). To avoid this, you should ensure that code and (writable) data do not occupy the same 2 Kbyte block of memory. One way of achieving that is to allocate the memory for the code as follows:

```
size% = 2048
DIM code% NOTEND AND 2047, code% size%-1
```

This allocates 2 Kbytes of memory for the code. The use of the variable `code%` twice is not a mistake, it ensures an accurate alignment of the address on a 2K 'boundary'. If there is insufficient room for the code, you can increase the size in multiples of 2 Kbytes (e.g. `size% = 4096` will allocate 4 Kbytes).

It is safe to incorporate items of data within your code using the DB, DD and DW pseudo-ops, but only if the data is constant (read-only). Writable data should be stored outside the block of memory containing the code. One way of guaranteeing that is as follows:

```
[OPT opt%
 .data1 DB 0 : DB 0
 .data2 DW 0
 .data4 DD 0
 ]
P% = (P% + 2047) AND -2048
[OPT opt%
 .codestart
```

The code and data will then occupy separate 2 Kbyte blocks.

Length of reserved memory

You must reserve an area of memory which is sufficiently large for your machine code program before you assemble it, but you may have no real idea how long the
program will be until after it is assembled. How then can you know how much memory to reserve? Unfortunately, the answer is that you can't. However, you can add to your program to find the length used and then change the memory reserved by the DIM statement to the correct amount.

In the example below, a large amount of memory is initially reserved. To begin with, a single pass is made through the assembly code and the length needed for the code is stored in S%. After a CLEAR, the correct amount of memory is reserved and a further two passes of the assembly code are performed as usual. Your program should not, of course, subsequently try to use variables set before the CLEAR statement. If you use a similar structure to the example and place the program lines which initiate the assembly function at the start of your program, you can place your assembly code anywhere you like and still avoid this problem.

```plaintext
DIM code% HIMEM-END-2048
S% = FNassemble(0, code%, 0) - code%
CLEAR
size% = S%
DIM code% NOTEND AND 2047, code% size%-1
S% = FNassemble(8, code%, code%+size%)
S% = FNassemble(10, code%, code%+size%)
     Put the rest of your program here.
     DEF FNassemble(opt%, P%, L%)
       [OPT opt%]
       Assembler code.
       ]
       = P%
```
The assembly process

OPT

The only assembly directive is OPT. As with the 6502 assembler, 'OPT' controls the way the assembler works, whether a listing is displayed and whether errors are reported. OPT should be followed by a number in the range 0 to 15. The way the assembler functions is controlled by the four bits of this number in the following manner.

Bit 0 - LSB

Bit 0 controls the listing. If it is set, a listing is displayed.

Bit 1

Bit 1 controls error reporting. If it is set, the No such variable and Jump out of range errors are reported as normal, otherwise they are suppressed. This bit should be reset in the first pass and set in the second pass.

Bit 2

Bit 2 controls where the assembled code is placed. If bit 2 is set, code is placed in memory starting at the address specified by O%. However, the program counter (P%) is still used by the assembler for calculating the instruction addresses.

Bit 3

Bit 3 controls limit checking. If bit 3 is set the code address (P% or O% as appropriate) is checked against the current value of L%. If greater than or equal to L% the Address out of range error results. For example:

```
DIM P% 99, L% -1
[OPT 8
```

will cause the assembler to issue an error if the code size exceeds 100 bytes.
Assembly at a different address

In general, machine code will only run properly if it is in memory at the addresses for which it was assembled. With *BBC BASIC for Windows* the memory addresses occupied by your program and data (and thus your assembly-language program) are allocated by Windows™ each time BASIC is run, so you cannot assume that they will always be the same. Thus the option of assembling to a different area of memory will rarely be needed. However, this facility has been retained for compatibility with other versions of BBC BASIC and for special purposes (e.g. for assembling code which will be copied into another process's address space).

**OPT summary**

<table>
<thead>
<tr>
<th>OPT value</th>
<th>Limit check</th>
<th>Code stored at</th>
<th>Errors reported</th>
<th>Listing generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No</td>
<td>P%</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>No</td>
<td>P%</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>No</td>
<td>P%</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>3</td>
<td>No</td>
<td>P%</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>No</td>
<td>O%</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>5</td>
<td>No</td>
<td>O%</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>6</td>
<td>No</td>
<td>O%</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>7</td>
<td>No</td>
<td>O%</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>8</td>
<td>Yes</td>
<td>P%</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>9</td>
<td>Yes</td>
<td>P%</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>10</td>
<td>Yes</td>
<td>P%</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>11</td>
<td>Yes</td>
<td>P%</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>12</td>
<td>Yes</td>
<td>O%</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>13</td>
<td>Yes</td>
<td>O%</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>14</td>
<td>Yes</td>
<td>O%</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>15</td>
<td>Yes</td>
<td>O%</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**How the assembler works**

The assembler works line by line through the assembly-language code. When it finds a label declared it generates a *BBC BASIC for Windows* variable with that
name and loads it with the current value of the program counter (P%). This is fine all the while labels are declared before they are used. However, labels are often used for forward jumps and no variable with that name would exist when it was first encountered. When this happens, a 'No such variable' error occurs. If error reporting has not been disabled, this error is reported and BBC BASIC for Windows returns to the direct mode in the normal way. If error reporting has been disabled the current value of the program counter is used in place of the address which would have been found in the variable, and assembly continues. By the end of the assembly process the variable will exist (assuming the code is correct), but this is of little use since the assembler cannot 'back track' and correct the errors. However, if a second pass is made through the assembly code, all the labels will exist as variables and errors will not occur. The example below shows the result of two passes through a (completely futile) demonstration program. Twelve bytes of memory are reserved for the program (if the program was run, it would 'doom-loop' from line 50 to 70 and back again). The program disables error reporting by using OPT 1.

10 DIM code 12
20 FOR opt=1 TO 3 STEP 2
30 P%=code
40 [OPT opt
50 .jim JMP fred
60 DW &2345
70 .fred JMP jim
80 ]
90 NEXT

This is the first pass through the assembly process (note that the 'JMP fred' instruction jumps to itself):

RUN
030A18A9                        OPT opt
030A18A9 E9 FB FF FF FF       .jim JMP fred
030A18AE 45 23             DW &2345
030A18B0 E9 F4 FF FF FF       .fred JMP jim

This is the second pass through the assembly process (note that the 'JMP fred' instruction now jumps to the correct address):

030A18A9                        OPT opt
030A18A9 E9 02 00 00 00       .jim JMP fred
030A18AE 45 23             DW &2345
030A18B0 E9 F4 FF FF FF       .fred JMP jim
Generally, if labels have been used, you must make two passes through the assembly language code to resolve forward references. This can be done using a FOR ...NEXT loop. Normally, the first pass should be with OPT 0 (or OPT 4, 8, 12) and the second pass with OPT 2 (or 6, 10, 14). If you want a listing, use OPT 3 (or 7, 11, 15) for the second pass. During the first pass, a table of variables giving the address of the labels is built. Labels which have not yet been included in the table (forward references) will generate the address of the current op-code. The correct address will be generated during the second pass.
Conditional assembly and macros

Introduction

Most machine code assemblers provide conditional assembly and macro facilities. The assembler does not directly offer these facilities, but it is possible to implement them by using other features of BBC BASIC for Windows.

Conditional assembly

You may wish to write a program which makes use of special facilities and which will be run on different types of computer. The majority of the assembly code will be the same, but some of it will be different. In the example below, different output routines are assembled depending on the value of 'flag'.

```
DIM code 200
FOR pass=0 TO 3 STEP 3
  [OPT pass
   .start     - - -
   - - - code - - -
   - - - :]
  : IF flag [OPT pass: - code for routine 1 - :]
  IF NOT flag [OPT pass: - code for routine 2 - :]
  : [OPT pass
   .more_code - - -
   - - - code - - -
   - - - :]
  NEXT
```

Macros

Within any machine code program it is often necessary to repeat a section of code a number of times and this can become quite tedious. You can avoid this repetition by defining a macro which you use every time you want to include the code. The example below uses a macro to pass a character to the screen or the auxiliary output. Conditional assembly is used within the macro to select either the screen or the auxiliary output depending on the value of op_flag.

It is possible to suppress the listing of the code in a macro by forcing bit 0 of OPT to
zero for the duration of the macro code. This can most easily be done by AND ing the value passed to OPT with 14. This is illustrated in PROC_screen and PROC_aux in the example below.

```basic
DIM code 200
op_flag=TRUE
FOR pass=0 TO 3 STEP 3
    [OPT pass
        .start  - - -
        - - - code - - -
        - - -
    :
        OPT FN_select(op_flag); Include code depending on op_flag
          - - -
          - - - code - - -
          - - -:
    NEXT
END :
:
REM Include code depending on value of op_flag :
DEF FN_select(op_flag)
IF op_flag PROC_screen ELSE PROC_aux
=pass
REM Return original value of OPT. This is a REM bit artificial, but necessary to insert REM some BASIC code in the assembly code. :
DEF PROC_screen
[OPT pass AND 14
  ...code...
]
ENDPROC :
DEF PROC_aux
[OPT pass AND 14
  ...code...
]
ENDPROC
```

The use of a function call to incorporate the code provides a neat way of incorporating the macro within the program and allows parameters to be passed to it. The function should return the original value of OPT.
## Table of ASCII codes

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
<th>Dec</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>&amp;0</td>
<td>0</td>
<td>^@ NUL</td>
</tr>
<tr>
<td>00000001</td>
<td>&amp;01</td>
<td>1</td>
<td>^A SOH Start of Heading</td>
</tr>
<tr>
<td>00000010</td>
<td>&amp;02</td>
<td>2</td>
<td>^B STX Start of Text</td>
</tr>
<tr>
<td>00000011</td>
<td>&amp;03</td>
<td>3</td>
<td>^C ETX End of Text</td>
</tr>
<tr>
<td>00000100</td>
<td>&amp;04</td>
<td>4</td>
<td>^D EOT End of Transmit</td>
</tr>
<tr>
<td>00000101</td>
<td>&amp;05</td>
<td>5</td>
<td>^E ENQ Enquiry</td>
</tr>
<tr>
<td>00000110</td>
<td>&amp;06</td>
<td>6</td>
<td>^F ACK Acknowledge</td>
</tr>
<tr>
<td>00000111</td>
<td>&amp;07</td>
<td>7</td>
<td>^G BEL Bell - Audible Signal</td>
</tr>
<tr>
<td>00001000</td>
<td>&amp;08</td>
<td>8</td>
<td>^H BS Back Space</td>
</tr>
<tr>
<td>00001001</td>
<td>&amp;09</td>
<td>9</td>
<td>^I HT Horizontal Tab</td>
</tr>
<tr>
<td>00001010</td>
<td>&amp;0A</td>
<td>10</td>
<td>^J LF Line Feed</td>
</tr>
<tr>
<td>00001011</td>
<td>&amp;0B</td>
<td>11</td>
<td>^K VT Vertical Tab</td>
</tr>
<tr>
<td>00001100</td>
<td>&amp;0C</td>
<td>12</td>
<td>^L FF Form Feed</td>
</tr>
<tr>
<td>00001101</td>
<td>&amp;0D</td>
<td>13</td>
<td>^M CR Carriage Return</td>
</tr>
<tr>
<td>00001110</td>
<td>&amp;0E</td>
<td>14</td>
<td>^N SO Shift Out</td>
</tr>
<tr>
<td>00001111</td>
<td>&amp;0F</td>
<td>15</td>
<td>^O SI Shift In</td>
</tr>
<tr>
<td>00010000</td>
<td>&amp;10</td>
<td>16</td>
<td>^P DLE Data Link Escape</td>
</tr>
<tr>
<td>00010001</td>
<td>&amp;11</td>
<td>17</td>
<td>^Q DC1 X On</td>
</tr>
<tr>
<td>00010010</td>
<td>&amp;12</td>
<td>18</td>
<td>^R DC2 Aux On</td>
</tr>
<tr>
<td>00010011</td>
<td>&amp;13</td>
<td>19</td>
<td>^S DC3 X Off</td>
</tr>
<tr>
<td>00010100</td>
<td>&amp;14</td>
<td>20</td>
<td>^T DC4 Aux Off</td>
</tr>
<tr>
<td>00010101</td>
<td>&amp;15</td>
<td>21</td>
<td>^U NAK Negative Acknowledge</td>
</tr>
<tr>
<td>00010110</td>
<td>&amp;16</td>
<td>22</td>
<td>^V SYN Synchronous Idle</td>
</tr>
<tr>
<td>00010111</td>
<td>&amp;17</td>
<td>23</td>
<td>^W ETB End of Transmitted Block</td>
</tr>
<tr>
<td>00011000</td>
<td>&amp;18</td>
<td>24</td>
<td>^X CAN Cancel</td>
</tr>
<tr>
<td>00011001</td>
<td>&amp;19</td>
<td>25</td>
<td>^Y EM End of Medium</td>
</tr>
<tr>
<td>00011010</td>
<td>&amp;1A</td>
<td>26</td>
<td>^Z SUB Substitute</td>
</tr>
<tr>
<td>00011011</td>
<td>&amp;1B</td>
<td>27</td>
<td>[ ESC Escape</td>
</tr>
<tr>
<td>00011100</td>
<td>&amp;1C</td>
<td>28</td>
<td>\ FS File Separator</td>
</tr>
<tr>
<td>00011101</td>
<td>&amp;1D</td>
<td>29</td>
<td>] GS Group Separator</td>
</tr>
<tr>
<td>00011110</td>
<td>&amp;1E</td>
<td>30</td>
<td>^^ RS Record Separator</td>
</tr>
<tr>
<td>00011111</td>
<td>&amp;1F</td>
<td>31</td>
<td>_ US Unit Separator</td>
</tr>
</tbody>
</table>
01000101 &45  69 E
01000110 &46  70 F
01000111 &47  71 G
01001000 &48  72 H
01001001 &49  73 I
01001010 &4A  74 J
01001011 &4B  75 K
01001100 &4C  76 L
01001101 &4D  77 M
01001110 &4E  78 N
01001111 &4F  79 O
01010000 &50  80 P
01010001 &51  81 Q
01010010 &52  82 R
01010011 &53  83 S
01010100 &54  84 T
01010101 &55  85 U
01010110 &56  86 V
01010111 &57  87 W
01011000 &58  88 X
01011001 &59  89 Y
01011010 &5A  90 Z
01011011 &5B  91 [
01011100 &5C  92 \
01011101 &5D  93 ]
01011110 &5E  94 ^
01011111 &5F  95 _

01100000 &60  96`
01100001 &61  97 a
01100010 &62  98 b
01100011 &63  99 c
01100100 &64 100 d
01100101 &65 101 e
01100110 &66 102 f
01100111 &67 103 g
01101000 &68 104 h
01101001 &69 105 i
01101010 &6A 106 j
01101011 &6B 107 k
01101100 &6C 108 l
01101101 &6D 109 m
01101110 &6E 110 n
01101111 &6F 111 o
01110000 &70 112 p
01110001 &71 113 q
01110010 &72 114 r
01110011 &73 115 s
01110100 &74 116 t
01110101 &75 117 u
01110110 &76 118 v
01110111 &77 119 w
01111000 &78 120 x
01111001 &79 121 y
01111010 &7A 122 z
01111011 &7B 123 { 
01111100 &7C 124 | 
01111101 &7D 125 } 
01111110 &7E 126 ~
01111111 &7F 127 DEL Delete
## Mathematical functions

*BBC BASIC for Windows* has more intrinsic mathematical functions than many other versions of BASIC. Some of those that are not provided may be calculated as shown below.

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SECANT</strong></td>
<td>DEF FNSEC(X) = 1/COS(X)</td>
</tr>
<tr>
<td><strong>COSECANT</strong></td>
<td>DEF FNCSC(X) = 1/SIN(X)</td>
</tr>
<tr>
<td><strong>COTANGENT</strong></td>
<td>DEF FNCOT(X) = 1/TAN(X)</td>
</tr>
<tr>
<td><strong>Inverse SECANT</strong></td>
<td>DEF FNARCSEC(X) = ACS(1/X)</td>
</tr>
<tr>
<td><strong>Inverse COSECANT</strong></td>
<td>DEF FNARCCSC(X) = ASN(1/X)</td>
</tr>
<tr>
<td><strong>Inverse COTANGENT</strong></td>
<td>DEF FNARCCOT(X) = PI/2 - ATN(X)</td>
</tr>
<tr>
<td><strong>Hyperbolic SINE</strong></td>
<td>DEF FNSINH(X) = (EXP(X) - EXP(-X))/2</td>
</tr>
<tr>
<td><strong>Hyperbolic COSINE</strong></td>
<td>DEF FNCOSH(X) = (EXP(X) + EXP(-X))/2</td>
</tr>
<tr>
<td><strong>Hyperbolic TANGENT</strong></td>
<td>DEF FNTANH(X) = EXP(-X)/(EXP(X) + EXP(-X)) * 2 + 1</td>
</tr>
<tr>
<td><strong>Hyperbolic SECANT</strong></td>
<td></td>
</tr>
</tbody>
</table>
DEF FNSECH(X) = 2/ (EXP(X) + EXP(-X))

Hyperbolic COSECANT

DEF FNCSCH(X) = 2/ (EXP(X) - EXP(-X))

Hyperbolic COTANGENT

DEF FCOTH(X) = EXP(-X) / (EXP(X) - EXP(-X)) * 2 + 1

Inverse Hyperbolic SIN

DEF FNARCSINH(X) = LN(X + SQR(X*X + 1))

Inverse Hyperbolic COSINE

DEF FNARCCOSH(X) = LN(X + SQR(X*X - 1))

Inverse Hyperbolic TANGENT

DEF FNARCTANH(X) = LN((1 + X) / (1 - X)) / 2

Inverse Hyperbolic SECANT

DEF FNARCSECH(X) = LN((SQR(-X*X + 1) + 1) / X)

Inverse Hyperbolic COSECANT

DEF FNARCCSCH(X) = LN((SGN(X) * SQR(X*X + 1) + 1) / X)

Inverse Hyperbolic COTANGENT

DEF FNARCCOTH(X) = LN((X + 1) / (X - 1)) / 2

LOGn(X)

DEF FNLOG(N, X) = LN(X) / LN(N)
DEF FNLOG(N, X) = LOG(X) / LOG(N)
**Keyword tokens**

*BBC BASIC for Windows* internal-format programs are *tokenised*, that is BASIC keywords are represented as single-byte values. This results in programs which are shorter and which execute faster; see the Format of data in memory section for more details. The following table lists the token value for each BBC BASIC keyword:

<table>
<thead>
<tr>
<th>Token</th>
<th>Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;80</td>
<td>AND</td>
</tr>
<tr>
<td>&amp;81</td>
<td>DIV</td>
</tr>
<tr>
<td>&amp;82</td>
<td>EOR</td>
</tr>
<tr>
<td>&amp;83</td>
<td>MOD</td>
</tr>
<tr>
<td>&amp;84</td>
<td>OR</td>
</tr>
<tr>
<td>&amp;85</td>
<td>ERROR</td>
</tr>
<tr>
<td>&amp;86</td>
<td>LINE</td>
</tr>
<tr>
<td>&amp;87</td>
<td>OFF</td>
</tr>
<tr>
<td>&amp;88</td>
<td>STEP</td>
</tr>
<tr>
<td>&amp;89</td>
<td>SPC</td>
</tr>
<tr>
<td>&amp;8A</td>
<td>TAB(3)</td>
</tr>
<tr>
<td>&amp;8B</td>
<td>ELSE</td>
</tr>
<tr>
<td>&amp;8C</td>
<td>THEN</td>
</tr>
<tr>
<td>&amp;8D</td>
<td>line number 1</td>
</tr>
<tr>
<td>&amp;8E</td>
<td>OPENIN</td>
</tr>
<tr>
<td>&amp;8F</td>
<td>PTR (right) 2</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;90</td>
<td>PAGE (right) 2</td>
</tr>
<tr>
<td>&amp;91</td>
<td>TIME (right) 2</td>
</tr>
<tr>
<td>&amp;92</td>
<td>LOMEM (right) 2</td>
</tr>
<tr>
<td>&amp;93</td>
<td>HIMEM (right) 2</td>
</tr>
<tr>
<td>&amp;94</td>
<td>ABS</td>
</tr>
<tr>
<td>&amp;95</td>
<td>ACS</td>
</tr>
<tr>
<td>&amp;96</td>
<td>ADVAL</td>
</tr>
</tbody>
</table>
ASC
ASN
ATN
BGET
COS
COUNT
DEG
ERL
ERR
EVAL
EXP
EXT
FALSE
FN
GET
INKEY
INSTR( ³ 
INT
LEN
LN
LOG
NOT
OPENUP
OPENOUT
PI
POINT( ³ 
POS
RAD
RND
SGN
&F1  PRINT
&F2  PROC
&F3  READ
&F4  REM
&F5  REPEAT
&F6  REPORT
&F7  RESTORE
&F8  RETURN
&F9  RUN
&FA  STOP
&FB  COLOUR 5
&FC  TRACE
&FD  UNTIL
&FE  WIDTH
&FF  OSCLI

&00  reserved
&01  CIRCLE
&02  ELLIPSE
&03  FILL
&04  MOUSE
&05  ORIGIN
&06  QUIT
&07  RECTANGLE
&08  SWAP
&09  SYS
&0A  TINT
&0B  WAIT
&0C  INSTALL
&0D  end of line
&0E  PRIVATE
&0F  BY
\&10  \text{EXIT}
\&11  \text{spare}
\&12  \text{spare}
\&13  \text{spare}
\&14  \text{spare}
\&15  \text{spare}
\&16  \text{spare}
\&17  \text{spare}
\&18  \text{reserved (fast FN/PROC)}^6
\&19  \text{reserved (fast byte)}^6
\&1A  \text{reserved (fast dword)}^6
\&1B  \text{reserved (fast double)}^6
\&1C  \text{reserved (fast variant)}^6
\&1D  \text{reserved (fast structure)}^6
\&1E  \text{reserved (fast qword)}^6
\&1F  \text{reserved (fast string)}^6

Notes:

1. Token value \&8D doesn't represent a keyword, but marks the presence of an encoded line number (e.g. following a GOTO, GOSUB or RESTORE).

2. The pseudo-variables PTR, PAGE, TIME, LOMEM and HIMEM each have two tokens, depending on whether they are being written (i.e. to the left of an equals sign) or read (to the right of an equals sign).

3. The tokens for TAB, INSTR, POINT, LEFT$, MID$, RIGHT$ and STRING$ include the opening bracket; there cannot be a space between the keyword and the bracket.

4. The keyword TOP does not have a token. It is represented by the token for TO followed by a \text{P}.

5. Both COLOUR and COLOR are accepted, but the token is always listed as COLOUR.
6. These tokens are used internally by the compiler and run-time-engine. They should not normally appear in a saved BASIC program.
Editor and compiler errors

The following error messages can be reported by the program editor, compiler and Integrated Development Environment:

**Cannot allocate clipboard memory**

There was insufficient memory to copy something to the clipboard. This should not happen in normal circumstances.

**Cannot allocate virtual memory**

The requested initial user memory could not be allocated. Try freeing up memory by closing other programs.

**Cannot allocate memory for file**

A file specified in a REM!Resource compiler directive could not be buffered in memory. This should not happen in normal circumstances.

**Cannot append program**

The Compile utility could not append your program to the executable file. Most likely the disk is full.

**Cannot create character map**

There was insufficient memory for the screen character map. This should not happen in normal circumstances.

**Cannot create file**

The specified file could not be created. Either the disk is full, a read-only or locked file of the same name exists, or you do not have write access to the relevant directory (folder).

**Cannot create palette**
The colour palette could not be created. This should not happen in normal circumstances.

**Cannot create screen bitmap**

There was insufficient memory for BASIC's screen bitmap. Try freeing up memory by closing other applications; if that fails, reboot your computer.

**Cannot create temporary buffer**

The memory required for a temporary buffer used during compilation could not be allocated. This should not happen in normal circumstances.

**Cannot crunch file**

The specified embedded file could not be crunched. To find out why not, try compiling it separately using the same crunch options.

**Cannot determine encoding**

The character encoding of the program being loaded could not be determined, for example both **ANSI** and **UTF-8** strings appear to be present; the required selection should be made manually. This warning will not be issued again if the program is re-loaded.

**Cannot embed file**

The Compile utility could not embed a specified resource file in the executable. Most likely the disk is full.

**Cannot find BBCTUTOR.CHM**

The Tutorial command could not locate the tutorial file BBCTUTOR.CHM. If necessary re-install *BBC BASIC for Windows* from your master disk.

**Cannot find BBCWIN.CHM**
The Help Topics command could not locate the help file BBCWIN.CHM. If necessary re-install *BBC BASIC for Windows* from your master disk.

**Cannot find icon in file**

The selected icon could not be found in the specified file.

**Cannot find manifest**

The Windows XP™ Visual Styles manifest could not be read. Suspect a corrupted BBCWIN.EXE or BBCWIN6.EXE file.

**Cannot open clipboard**

The clipboard could not be opened. This should not happen in normal circumstances.

**Cannot open file**

The specified file could not be opened. Most likely it does not exist in the specified place or you don't have read access to the relevant directory (folder).

**Cannot open icon file**

The file containing the selected icon could not be opened. Most likely it does not exist in the specified place or you don't have read access to the relevant directory (folder).

**Cannot open temporary file**

The specified temporary file could not be opened. Most likely it does not exist in the specified place or you don't have the necessary access rights.

**Cannot print program**

An attempt to print your program failed.

**Cannot process resource in file**
A file specified in a REM!Resource compiler directive contains an invalid resource that could not be added to the executable.

**Cannot read file**

The specified file could not be read. Suspect a corrupted or damaged disk.

**Cannot read icon file**

The file containing the selected icon could not be read. Suspect a corrupted or damaged disk.

**Cannot read icon image from file**

The selected icon resource could not be read. Suspect a corrupted or damaged file.

**Cannot run Sign Tool**

The code-signing utility could not be run; most likely the path or filename specified in the registry is incorrect. The relevant key is: \HKCU\Software\R T Russell\BBC BASIC for Windows\Settings\SignTool

**Cannot update resources in file**

The resources (e.g. icon) of the specified file could not be updated. Either the file does not exist, has the wrong format or could not be opened for writing.

**Cannot write file**

The specified file could not be written. Most likely the disk is full.

**Cannot write icon**

The Compile utility could not store the selected icon in the executable file. Most likely the disk is full.

**Crunch failed: assembler syntax error**
The crunch option of the Compile utility encountered a syntax error within some assembly language code.

**Crunch failed: bad variable name**

The crunch option of the Compile utility encountered a bad assembler label, for example ..label (two dots).

**Crunch failed: calculated line number**

The crunch option of the Compile utility encountered a calculated line number, e.g. GOTO (100+offset%). If you use calculated line numbers you must deselect the Concatenate lines option.

**Crunch failed: invalid fast variable**

The crunch option of the Compile utility encountered a REM!Fast compiler directive specifying an invalid variable name, for example containing an illegal character.

**Crunch failed: invalid keep variable**

The crunch option of the Compile utility encountered a REM!Keep compiler directive specifying an invalid variable name, for example with fewer than three characters.

**Crunch failed: missing \**

The crunch option of the Compile utility detected a line continuation character without another at the beginning of the next line.

**Crunch failed: mismatched quotes**

The crunch option of the Compile utility detected mismatched quotation marks ("'). There must always be an even number in the line.

**Crunch failed: statement not at start of line**
The crunch option of the Compile utility detected an ENDCASE, ENDIF, OTHERWISE or WHEN not at the start of a line.

**Directory was changed**

When selecting files for embedding in an executable (Compile utility) the directory was changed. Only files in the `@dir$` or `@lib$` directories, or their subdirectories, can be embedded.

**File is not a valid resource file**

A file specified in a `REM!Resource` compiler directive does not have the correct signature for a `.res` format file.

**HTML Help not installed**

The Help Topics or Tutorial command could not access the HTML Help system. On Windows™ 95 and NT 4.0 you may need to install Internet Explorer version 4 or later to rectify this error.

**Insufficient memory**

There is insufficient memory to CHAIN a program file, or to change the value of PAGE.

**Insufficient memory to crunch file**

There is insufficient memory to crunch the specified embedded file.

**Insufficient memory to crunch program**

There is insufficient memory to run the crunch option of the Compile utility. Increase the value of HIMEM or change the initial user memory setting and re-start BBC BASIC.

**Invalid file format**

The program file you are attempting to load does not have a valid tokenised or
ASCII format. The file may have been corrupted.

**Line too long**

An attempt to concatenate two program lines, by typing **backspace** when the cursor is at the beginning of the line, would cause the maximum line length to be exceeded.

**Line too long - truncated!**

When *tokenising* a program line, i.e. converting it from ASCII to internal format, the maximum line length was exceeded and the remainder of the line was discarded. This can happen when loading a program from a file or when typing (or pasting) it in.

**Not enough room for file**

There is insufficient memory to Load or Insert a file. Increase the value of HIMEM or change the initial user memory setting and re-start BBC BASIC.

**Program is too big for allocated memory**

There is insufficient memory for your BASIC program. Increase the value of HIMEM or change the initial user memory setting and re-start BBC BASIC.

**Sign Tool failed or aborted**

The code-signing utility returned a failure status; most likely the parameters specified in the registry are incorrect (where the path/filename of the executable to be signed is given as "%s" ), or the authorising server couldn't be contacted. The relevant key is: \HKCU\Software\R T Russell\BBC BASIC for Windows\Settings\SignParm

**Too many files selected**

You have attempted to add too many embedded files in one go. Add the files in smaller groups.
Run-time errors

The following error messages can be reported while your BASIC program is running:

**Trappable errors - BASIC**

<table>
<thead>
<tr>
<th>No</th>
<th>Error</th>
<th>No</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jump out of range</td>
<td>2</td>
<td>Size needed</td>
</tr>
<tr>
<td>3</td>
<td>Multiple label</td>
<td>4</td>
<td>Mistake</td>
</tr>
<tr>
<td>5</td>
<td>Missing ,</td>
<td>6</td>
<td>Type mismatch</td>
</tr>
<tr>
<td>7</td>
<td>Not in a function</td>
<td>8</td>
<td>Address out of range</td>
</tr>
<tr>
<td>9</td>
<td>Missing &quot;</td>
<td>10</td>
<td>Bad DIM statement</td>
</tr>
<tr>
<td>11</td>
<td>DIM space</td>
<td>12</td>
<td>Not in a FN or PROC</td>
</tr>
<tr>
<td>13</td>
<td>Not in a procedure</td>
<td>14</td>
<td>Bad use of array</td>
</tr>
<tr>
<td>15</td>
<td>Bad subscript</td>
<td>16</td>
<td>Syntax error</td>
</tr>
<tr>
<td>17</td>
<td>Escape</td>
<td>18</td>
<td>Division by zero</td>
</tr>
<tr>
<td>19</td>
<td>String too long</td>
<td>20</td>
<td>Number too big</td>
</tr>
<tr>
<td>21</td>
<td>Negative root</td>
<td>22</td>
<td>Logarithm range</td>
</tr>
<tr>
<td>23</td>
<td>Accuracy lost</td>
<td>24</td>
<td>Exponent range</td>
</tr>
<tr>
<td>25</td>
<td>Bad MODE</td>
<td>26</td>
<td>No such variable</td>
</tr>
<tr>
<td>27</td>
<td>Missing )</td>
<td>28</td>
<td>Bad hex or binary</td>
</tr>
<tr>
<td>29</td>
<td>No such FN/PROC</td>
<td>30</td>
<td>Bad call</td>
</tr>
<tr>
<td>31</td>
<td>Incorrect arguments</td>
<td>32</td>
<td>Not in a FOR loop</td>
</tr>
<tr>
<td>33</td>
<td>Can't match FOR</td>
<td>34</td>
<td>Bad FOR variable</td>
</tr>
<tr>
<td>35</td>
<td>STEP cannot be zero</td>
<td>36</td>
<td>Missing TO</td>
</tr>
<tr>
<td>37</td>
<td>Missing OF</td>
<td>38</td>
<td>Not in a subroutine</td>
</tr>
<tr>
<td>39</td>
<td>ON syntax</td>
<td>40</td>
<td>ON range</td>
</tr>
<tr>
<td>41</td>
<td>No such line</td>
<td>42</td>
<td>Out of data</td>
</tr>
<tr>
<td>43</td>
<td>Not in a REPEAT loop</td>
<td>44</td>
<td>WHEN/OTHERWISE not first</td>
</tr>
<tr>
<td>45</td>
<td>Missing #</td>
<td>46</td>
<td>Not in a WHILE loop</td>
</tr>
<tr>
<td>47</td>
<td>Missing ENDCASE</td>
<td>48</td>
<td>OF not last</td>
</tr>
</tbody>
</table>
49  Missing ENDIF
50  Bad MOUSE variable
51  No such system call
52  Bad library
53  Size mismatch
54  DATA not LOCAL
55  Missing \n56  Bad use of structure

**Trappable errors - OS**

<table>
<thead>
<tr>
<th>No</th>
<th>Error</th>
<th>No</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>189</td>
<td>Access denied</td>
<td>192</td>
<td>Too many open files</td>
</tr>
<tr>
<td>196</td>
<td>File exists</td>
<td>198</td>
<td>Disk full</td>
</tr>
<tr>
<td>199</td>
<td>Disk fault</td>
<td>201</td>
<td>Disk read only</td>
</tr>
<tr>
<td>202</td>
<td>Device fault</td>
<td>204</td>
<td>Bad name</td>
</tr>
<tr>
<td>205</td>
<td>Bad device</td>
<td>206</td>
<td>Bad directory</td>
</tr>
<tr>
<td>214</td>
<td>File or path not found</td>
<td>222</td>
<td>Invalid channel</td>
</tr>
<tr>
<td>244</td>
<td>No coprocessor</td>
<td>245</td>
<td>Device unavailable</td>
</tr>
<tr>
<td>246</td>
<td>No such font</td>
<td>248</td>
<td>No such printer</td>
</tr>
<tr>
<td>251</td>
<td>Bad key</td>
<td>253</td>
<td>Bad string</td>
</tr>
<tr>
<td>254</td>
<td>Bad command</td>
<td>255</td>
<td>Unknown error</td>
</tr>
</tbody>
</table>

**Untrappable errors**

The following untrappable ('fatal') errors have an error code of zero:

- No room
- ON ERROR not LOCAL
- Bad program

Strictly speaking 'Bad program' does not have an error code. It leaves ERR and ERL unchanged.
Access denied

This is something of a 'catch all' error message. It originates from Windows™ and occurs whenever the operating system thinks you are trying to illegally write to or delete a file or directory.

The most common case of this error is trying to write to a file that was opened for input with OPENIN. If you wish to write to an existing file, you should open it for read/write with OPENUP.

On a network system, this error may indicate that you are trying to access a file that has been locked by another user. For example, two users trying to open the same file with OPENUP will give rise to this error.

You will also get this error if you try to remove a directory which has files in it. If you wish to remove (delete) a directory you must first delete all the files in it.
Accuracy lost

Before *BBC BASIC for Windows* calculates trigonometric functions (SIN, COS, etc.) of very large angles the angles are reduced to ± PI radians. The larger the angle, the greater the inaccuracy of the reduction, and hence the result. When this inaccuracy becomes unacceptable, BASIC will issue an 'Accuracy lost' error message.

Selecting the *FLOAT 64* mode allows the use of larger angles before an 'Accuracy lost' error is generated.
Address out of range

Although BBC BASIC programs should always allocate memory 'properly' (e.g. using the DIM statement or by lowering HIMEM) many programs written for the BBC Micro (and some written for other versions of BBC BASIC) contain references to 'absolute' memory addresses. Such programs would be very likely to crash BBC BASIC for Windows, since the addresses usually aren't valid in Windows™. For this reason, attempts to access memory addresses below &10000 are trapped and result in the 'Address out of range' error rather than a crash.

The address is checked in the case of indirection (i.e. the ?, ! and $ operators), when setting the values of PAGE and LOMEM, in the CALL, SYS and USR statements, and in the *LOAD and *SAVE commands.

This error also results when the assembler detects that the code pointer has reached the specified limit value L%.
Bad call

This error indicates that a procedure or function has been called incorrectly, for example by adding a space between the keyword FN and the function name:

```plaintext
PRINT FN mean(5, 6)
DEF FNmean(A, B) = (A + B) / 2
```
Bad command

This error occurs when a 'star' (OSCLI ) command is not correctly structured, or when Windows™ cannot understand a filename or file specification. For example, if the memory address is omitted from a *LOAD command, the 'Bad command' error will result.

Note that if a 'star' command is not recognised by BBC BASIC it is passed to Windows™ for execution. Should it not be a valid GUI or console command (for example it was mis-typed) the 'Bad command' error will not be generated but instead the 'Bad command or filename' error will be displayed at the Command Prompt. This is not trappable by a BASIC program, and may appear on the screen only fleetingly, if at all.
Bad DIM statement

This error is produced when there is a mistake in a DIM statement. For example, if an array suffix is negative a 'Bad DIM statement' error will result:

```
DIM table(20,-10)
```

This error will also result if you attempt to re-dimension an array which has already been declared, or redeclare a structure which has already been declared:

```
100 DIM A(123)
110 DIM A(100)
120 DIM my_struct{x%,y%}
130 DIM my_struct{A,B}
```

Note that you can repeat an array or structure declaration so long as the dimensions or format respectively are identical to those specified when the array/structure was first declared. In this case the contents of the array or structure remain unchanged: they are *not* initialised.
Bad directory

This error indicates that Windows™ could not understand a directory (folder) name, for example it contained illegal characters. For example, the command

```
*CD \:\
```

would give rise to the 'Bad directory' error.
Bad device

This error is reported if Windows™ could not understand a device name.
Bad FOR variable

The variable in a FOR ...NEXT loop must be a numeric variable. If you use a constant or a string variable this error message will be generated. For example, the following statements are not legal.

```plaintext
20 FOR name$=1 TO 20
20 FOR 10=1 TO 20
```
Bad hex or binary

Hexadecimal numbers can only include the characters 0 to 9 and A to F; similarly binary numbers can only contain the digits 0 and 1. If you try to form a hexadecimal or binary number with other characters this error will occur. For example:

\[
\begin{align*}
    n &= \&OF \\
    n &= \%21
\end{align*}
\]

(in the first example a letter O was used where the figure 0 was intended).
Bad key

This error is generated if the key number in a *KEY command is greater than 24.
Bad library

This error occurs if a module loaded with the INSTALL or CALL statement has an incorrect format, i.e. is anything other than a valid *BBC BASIC for Windows* internal format (tokenised) program file. This is similar to the 'Bad program' error except that 'Bad library' is trappable with ON ERROR but 'Bad program' is not.
Bad MODE

This error is generated if an attempt is made to select a MODE number greater than 33.
**Bad MOUSE variable**

This error is generated if one or more variables specified in the MOUSE statement is invalid, for example is a string variable rather than a numeric variable.
Bad name

This error is generated if a file specification exceeds 255 characters in length (or would exceed 255 characters after a default extension, e.g. .BBC, is added).
Bad program

When a program is run, or is CHAIN ed from another program, *BBC BASIC for Windows* checks to see that it has the correct internal (tokenised) format; if not, the 'Bad program' error is generated.

This error can be caused by attempting to run a file with an inappropriate format (e.g. a program in ASCII text format) or by corrupting the program in memory (machine code programmers beware). 'Bad program' is an untrappable error and does not have an error code (by definition there is no program present which could make use of such a code).

The most likely cause is that the program file is in ASCII (text) or Acorn format, rather than the correct internal (tokenised) format. In that case you can convert the file to the wanted format by Loading it into the program editor and Saving it As a BBC BASIC (.BBC) file.
Bad string

File names (etc.) in 'star' (OSCLI ) commands may optionally be enclosed in quotes; this is mandatory if the filename includes a space. The 'Bad string' error will occur if the quotes are unmatched. The following example would give rise to this error:

*DEL "GRAPHS"

This error is also generated if there is insufficient space for the key string in a *KEY command.
Bad subscript

This error is generated if you attempt to access an element of an array whose subscript is negative or greater than the dimension specified in the DIM statement. Both lines 20 and 30 of the following example would give rise to this error:

```
10 DIM test(10)
20 test(-4)=20
30 test(30)=10
```
Bad use of array

This error occurs when an array is referenced but where the use of an array is inappropriate, or when an undimensioned array is referenced (for example after an array is made LOCAL or PRIVATE but before it is re-DIM ensioned).
Bad use of structure

This error occurs when a structure is referenced but where the use of a structure is inappropriate, or when an *undeclared* structure is referenced (for example after a structure is made LOCAL or PRIVATE but before it is redeclared with DIM). The following example would give rise to this error:

```
DIM mystruct{x,y,z}
mystruct() = 1
```
Can't match FOR

This error is generated when BBC BASIC for Windows is unable to find a FOR statement corresponding to the current NEXT statement. BBC BASIC will 'pop' FOR ... NEXT loops in an attempt to match the variable name specified after NEXT, but if no matching FOR is found the 'Can't match FOR' error results. For example:

```
10 FOR A% = 1 TO 10
20   FOR B% = 1 TO 10
30   NEXT C%
```

It is also generated if an EXIT FOR var statement is encountered and the specified variable does not match that of an enclosing FOR...NEXT loop.
DATA not LOCAL

This error is generated if BBC BASIC for Windows encounters a RESTORE DATA statement but is unable to match it with a corresponding LOCAL DATA statement. This can happen as a result of jumping out of a loop in the meantime.

Note that when LOCAL DATA is used inside a FOR ... NEXT, REPEAT ... UNTIL or WHILE ... ENDFOR loop, or inside a user-defined function, procedure or subroutine, the data pointer is automatically restored on exit (there is no need to use RESTORE DATA).
Device fault

This error is generated if a device (e.g. a disk drive) accessed by BBC BASIC for Windows appears not to be ready. This is most commonly caused by attempting to access a floppy disk when no disk is in the drive, or when the disk is seriously damaged.
This error is generated when an attempt is made to play a MIDI file using the *PLAY command, but there is no suitable MIDI device available, or when the SOUND statement is used and no suitable wave output device is available. This is most commonly caused by the device already being in use (for example as the result of a previous *PLAY command, or when another program is using the sound system) but may also indicate that no suitable sound card is installed. A currently-playing MIDI file can be aborted using SOUND OFF.

If you want to avoid this error, you can incorporate the following code at the start of your program:

```plaintext
SoundOK% = TRUE
ON ERROR LOCAL SoundOK% = FALSE
IF SoundOK% SOUND 1,0,0,1
RESTORE ERROR
```

This will set the variable **SoundOK%** to TRUE if it is safe to use the SOUND statement and to FALSE if not. All subsequent SOUND statements in your program should be made conditional on **SoundOK%** as follows:

```plaintext
IF SoundOK% SOUND ..........
```
This error is generated if there is insufficient room for an array or structure when you try to declare it, or if an attempt is made to reserve a negative amount of memory. For example,

```
DIM A% -2
```

If there is insufficient room for an array or structure, you may be able to increase the memory available to BBC BASIC for Windows by raising the value of HIMEM. Failing that see the Memory management section for hints on how to reduce memory usage.
Disk fault

This error is generated if a fault occurs while accessing a disk. It will occur, for example, if the contents of the disk have been corrupted and a file is unreadable.
Disk full

This error will occur if there is insufficient room on the disk for the file being written to it.
Disk read only

This error is generated if an attempt is made to write to a disk which is write protected, for example a floppy disk which has had its 'write protect' tab moved to the 'protect' position.
Mathematically, dividing by zero gives an infinitely large answer. The computer is unable to understand the concept of infinity (it's not alone) and this error is generated. If there is any possibility that the divisor might be zero, you should test for this condition before carrying out the division. For example:

```
200 IF divisor=0 THEN PROC_error ELSE...
```

This error is also generated by the DIV and MOD functions if the second value is zero.
This error is generated by pressing the <Esc> key. You can trap this, and other errors, by using the ON ERROR statement. You can inhibit the generation of the 'Escape' error by using *ESC OFF. The <Esc> key then returns the ASCII value of escape (&1B). *ESC ON restores the default action of the <Esc> key.
Exponent range

The EXP function is unable to cope with powers greater than about 88 (in *FLOAT 40 mode) or 709 (in *FLOAT 64 mode), since the result would be larger than the maximum number which can be handled by BBC BASIC for Windows. If you try to use a larger power, this error will be generated.
File exists

This error is generated if you try to rename a file, and a file with the new name already exists, or to create a directory, and the specified directory already exists.

It will also occur if you try to *SAVE , *SCREENSAVE or *SPOOL to a file in a directory (folder) which does not exist or is not writable by you (in particular under Windows™ Vista or Windows™ 7).

Finally, it can occur if you try to *CHDIR to a non-existent directory, *RMDIR a non-existent directory or *MKDIR in a directory which doesn't exist or which is not writable by you.

Note that this error is not generated if you try to open the file with OPENIN , OPENOUT or OPENUP . In these cases the function returns the value zero, but no error is generated.
Incorrect arguments

This error indicates that too many or too few arguments have been passed to a procedure or function, or an invalid parameter has been used. See the sub-section on Procedures and Functions for more details.

It is also generated if a SYS statement has too many parameters.
Invalid channel

This error is generated if a specified channel number does not correspond to an open file. This may be because the file has been closed, or was never opened successfully, possibly because the OPENIN, OPENOUT or OPENUP function failed.

The 'Invalid channel' error is also generated if PTR# or EOF# is used with a channel opened to a communications port (e.g. a serial port).
Jump out of range

This assembly language error is generated if you try to perform a relative (short) jump to a destination which is more than +127 or -128 bytes away. Except in the case of LOOP and JE CXZ statements (and their variants), you can solve this problem by specifying a near jump:

\[ jz \text{ near label} \]

This error is also generated (rather misleadingly) when an assembly language statement requires an 8-bit (one byte) operand but the value supplied won't fit in 8 bits. For example the statement:

\[ add \text{ al,1000} \]

would generate the error, because the decimal value 1000 is too big to fit in a single byte.
Logarithm range

Logarithms for zero and negative numbers do not exist. This error is generated if you try to calculate the LOG or LN of zero or a negative number, or raise a negative number to a non-integer power.
Missing ,

This error is generated if *BBC BASIC for Windows* was unable to find a comma where one was expected. The following example would give rise to this error.

20 PRINT TAB(10 5)
This error message is generated if *BBC BASIC for Windows* was unable to find a double-quote where one was expected. The following example would give rise to this error.

```
10 name$="Douglas"
```
This error is generated if *BBC BASIC for Windows* was unable to find a closing bracket where one was expected. The following example would give rise to this error.

```
10 PRINT SQR(num
```
Missing #

This error will occur if *BBC BASIC for Windows* is unable to find a hash symbol (a pound symbol on some computers) where one was expected. The following example would give rise to this error.

```
CLOSE 7
```
This error will occur if there is no backslash (\) symbol at the beginning of a continuation line. See the Line continuation section for details. The following example would give rise to this error.

```
FOR I% = 1 TO \nN% : PRINT I% : NEXT
```

If you receive this error as the result of using backslash as an assembler comment delimiter, change the backslash to a semicolon (;).
Missing ENDCASE

This error is generated if BBC BASIC for Windows cannot locate an ENDCASE statement to terminate the current CASE statement.
Missing ENDIF

This error is generated if *BBC BASIC for Windows* cannot locate an ENDIF statement to terminate the current multi-line IF ... THEN statement.
This error is generated if *BBC BASIC for Windows* encounters a CASE statement with the OF keyword missing.
This error message is generated if *BBC BASIC for Windows* encounters a FOR ...NEXT loop with the TO part missing.
This error is generated if a statement does not begin with a keyword, but neither is it an assignment to a variable; in other words *BBC BASIC for Windows* is unable to make any sense of the statement. This is most likely to result from mistyping a keyword.
Multiple label

This error is generated by the assembler if, during the first pass of an assembly (OPT 0, 1, 4, 5, 8, 9, 12 or 13), a label is found to have an existing non-zero value. This is most likely to be caused by the same label being used twice (or more times) in the program.
This error is generated if *BBC BASIC for Windows* attempts to calculate the square root of a negative number. It is possible for this error to occur with ASN and ACS as well as SQR.

```
90 num=-20
100 root=SQR(num)
```
No coprocessor

This error is generated if a *FLOAT 64 command is executed but no numeric data processor is available. This will only happen with older PCs using an 80386 or 80486SX CPU (without an 80387 or 80487SX coprocessor) since all later PC CPUs contain a numeric data processor as standard.
No room

This untrappable error indicates that all the memory available to BASIC was used up whilst a program was running. You may be able to increase the memory available to *BBC BASIC for Windows* by raising the value of HIMEM. Failing that see the Memory management section for hints on how to reduce memory usage.
No such FN/PROC

This error is generated if *BBC BASIC for Windows* encounters a function call (FN) or procedure call (PROC) but no function or procedure with that name has been defined (DEF).

It is also generated if an attempt is made to get a pointer to a PROC or FN before any conventional function or procedure calls have been made.
No such font

This error is generated if the font name given in a *FONT or *PRINTERFONT command cannot be recognised. In practice Windows™ may choose an alternative (possibly unsuitable) font rather than issuing this error.
No such line

This error is generated if BBC BASIC for Windows tries to GOTO, GOSUB, TRACE or RESTORE to a non-existent line number.
No such printer

This error is generated if the printer name given in a *PRINTER command cannot be recognised. Printer names are not case sensitive, but otherwise must be precise.
No such system call

This error is generated if a Windows™ API function called by name with a SYS statement (or called from assembler code) cannot be recognised. This is most likely to be caused by the function name being mistyped (function names are case-sensitive). Note, however, that only API functions in the following Dynamic Link Libraries may be called by name:

- ADVAPI32.DLL
- COMCTL32.DLL
- COMDLG32.DLL
- GDI32.DLL
- KERNEL32.DLL
- SHELL32.DLL
- USER32.DLL
- WINMM.DLL

Other DLLs must be loaded explicitly and functions they contain must be called by address rather than by name. For example to call the OleUIChangeIconA function in OLEDLG.DLL:

```systems
SYS "LoadLibrary", "OLEDLG.DLL" TO oledlg%
SYS "GetProcAddress", oledlg%, "OleUIChangeIconA" TO chicon%
SYS chicon%, ci% TO uint%
```
No such variable

This error is generated if you try to access the value of a variable which has not yet been created. See the Creation of variables section for details of how variables may be created.
This error is generated if you try to define a variable as LOCAL or PRIVATE outside of a procedure or function. If you forget to put multi-line function definitions out of harm's way at the end of the program you may get this error message (see the sub-section on Procedures and Functions for more details).
Not in a FOR loop

This error is generated if *BBC BASIC for Windows* encounters a NEXT statement but is unable to match it with a corresponding FOR statement. This can be caused by having 'jumped out of a loop' in the meantime (see the sub-section on Program Flow Control for more details).

It is also generated if an EXIT FOR statement is encountered when not in a FOR...NEXT loop.
Not in a function

This error is generated if *BBC BASIC for Windows* encounters an end-of-function statement (i.e. one beginning with an 'equals' sign) but is not currently within a function. If you forget to put multi-line function definitions out of harm's way at the end of the program you are very likely to get this error message (see the sub-section on Procedures and Functions for more details). This can also be caused by having 'jumped out of a loop' whilst within a function.
Not in a procedure

This error is generated if *BBC BASIC for Windows* encounters an ENDPROC but is not currently within a procedure. If you forget to put multi-line procedure definitions out of harm's way at the end of the program you are very likely to get this error message (see the sub-section on Procedures and Functions for more details). This can also be caused by having 'jumped out of a loop' whilst within a procedure.
This error is generated if *BBC BASIC for Windows* encounters an UNTIL statement but is unable to match it with a corresponding REPEAT statement. This can be caused by having 'jumped out of a loop' in the meantime (see the sub-section on Program Flow Control for more details).

It is also generated if an EXIT REPEAT statement is encountered when not in a REPEAT...UNTIL loop.
This error is generated if *BBC BASIC for Windows* encounters a RETURN statement but is unable to match it with a corresponding GOSUB or ON statement. This can be caused by having 'jumped out of a loop' in the meantime (see the subsection on Program Flow Control for more details).
This error is generated if *BBC BASIC for Windows* encounters an ENDWHILE statement but is unable to match it with a corresponding WHILE statement. This can be caused by having 'jumped out of a loop' in the meantime (see the sub-section on Program Flow Control for more details).

It is also generated if an EXIT WHILE statement is encountered when not in a WHILE...ENDWHILE loop.
Number too big

This error is generated if a number is entered or calculated which is too big for *BBC BASIC for Windows* to cope with. The range of values of a 32-bit integer variable (e.g. `a%`) is from -2147483648 to +2147483647 and the range of values of a 64-bit integer variable (e.g. `a%`%) is from -9223372036854775808 to +9223372036854775807.

The range of values of a 40-bit floating-point variable is approximately -3.4E38 to +3.4E38, the range of values of a 64-bit floating-point variable (e.g. `a#`) is approximately -1.7E308 to +1.7E308 and the range of values of an 80-bit variant variable is approximately -1.1E4932 to +1.1E4932.
OF not last

This error is generated if the keyword OF is not the last thing on the line in a CASE statement. OF cannot even be followed by a REMark.
ON ERROR not LOCAL

This error is generated if *BBC BASIC for Windows* encounters a RESTORE ERROR statement but is unable to match it with a corresponding ON ERROR LOCAL statement. This can be caused by having 'jumped out of a loop' in the meantime (see the sub-section on Program Flow Control for more details).
ON range

This error is generated if, in an ON GOTO/GOSUB/PROC statement, the control variable was less than one or greater than the number of entries in the ON list, and there was no ELSE option. The first example below will generate an 'ON range' error, whilst the second is correct:

```
10 num=4
20 ON num GOTO 100,200,300
10 num=4
20 ON num GOTO 100,200,300 ELSE 1000
```
ON syntax

This error is generated if an ON statement is misformed. For example:

```
20 ON x TIME=0
```
Out of DATA

This error is generated if your program tries to READ more items of data than there are in the DATA list(s). You can use RESTORE to return the data pointer to the first DATA statement (or to a particular line with a data statement) if you wish.
Size mismatch

This assembly language error is generated when an inappropriate mixture of 16-bit and 32-bit operands is used in the same instruction. For example whilst the following two instructions are both valid:

```
mov ax, bx
mov eax, ebx
```

the following (non-existent) instructions will give rise to the 'Size mismatch' error:

```
mov ax, ebx
mov eax, bx
```
Size needed

Certain assembly language instructions are ambiguous as to whether a byte (8-bit), word (16-bit) double-word (32-bit) or quad-word (64-bit) value is to be acted upon. In these cases the data size must be explicitly stated as **byte**, **word**, **dword** or **qword** (or **byte ptr**, **word ptr**, **dword ptr**, **qword ptr**). For example:

```assembly
    inc byte [ebx]
    mov dword [count], 0
    add word [sum], 3
```

If this is not done, a 'Size needed' error will be generated.
STEP cannot be zero

This error is generated if the STEP value in a FOR ... NEXT statement is zero. For example:

```
FOR N% = 1 TO 10 STEP 0
```
String too long

This error is generated if a string is longer than the allowed maximum, for example the maximum length of a filename or OSCLI (*) command is 255 characters; the maximum length of a string passed to VAL is 65535 characters.
Syntax error

This error is generated if a keyword was recognised, but the rest of the statement or function was meaningless or incomplete. It is also generated if an illegal character is encountered in a variable name or value.
Too many open files

This error is generated if you try to open more than eight files or four communications ports (with OPENIN, OPENOUT or OPENUP) at any one time. To avoid this error make sure that you CLOSE each file as soon as you have finished with it. See Opening files and Serial input/output for more details.
Type mismatch

This error is generated when a number was expected but a string was encountered, or vice-versa. For example:

```
10 TIME = "Midnight"
20 Length = LEN(A%)
```

This error can be generated when the actual parameters and the formal parameters of a function or procedure do not correspond (see Procedures and functions for more details).

The **Type mismatch** error will also result when array arithmetic is performed but the dimensions of the arrays are not compatible.
Unknown error

This error is generated if Windows™ reports an error condition which BBC BASIC for Windows does not recognise. You can discover the Windows error number (so long as no other error has occurred in the meantime) as follows:

```
SYS "GetLastError" TO ern%
PRINT ern%
```

If you prefer you can convert the error number to a string:

```
DEF FNerrorstring(ern%)
DIM buf% LOCAL 255
SYS "FormatMessage",&1000,0,ern%,0,buf%,255,0
= $$buf%
```
WHEN/OTHERWISE not first

This error is generated if a WHEN or OTHERWISE clause of a CASE statement does not appear at the very start of a program line.
Memory map

_BBC BASIC for Windows_ is a 32-bit program and uses 32-bit 'flat' (unsegmented) memory addressing. The user's program, data, stack and libraries occupy a contiguous block of memory, reserved by Windows™ when BASIC is started. Absolute memory addresses therefore cannot be guaranteed to be the same each time BASIC is run, and your programs should only ever access memory allocated with the DIM statement, allocated by a Windows™ API function or at addresses relative to PAGE, LOMEM, HIMEM etc.

The current BASIC program starts at PAGE and ends at the byte immediately below TOP (i.e. the length of the program is TOP-PAGE bytes). The 'dynamic data structures' (variables, arrays etc.) start at LOMEM which by default is set equal to TOP (i.e. they follow immediately after your program). This area of memory, which is called the 'heap', grows upwards as more variables are created. The stack grows downwards from HIMEM which by default is set a little less than one Megabyte (two Megabytes in _BBC BASIC for Windows version 6.00a or later_) above PAGE, but may be raised above this value by the user (memory permitting) either using a program statement or the Customize menu command.
As your program runs, the heap expands upwards towards the stack and the stack expands downwards towards the heap. If the two should meet, you get a 'No room' error. Fortunately, there is a limit to the amount by which the stack and the heap expand.

In general, the heap only expands whilst new variables, arrays or structures are being declared. However, altering the length of string variables can result in 'dead' string space which also causes the heap to expand.

In addition to storing the 'return addresses' and other information about the nested structures in your program (loops, procedures, functions etc.), the stack is also used 'internally' by the *BBC BASIC for Windows* interpreter. Its size fluctuates but, in general, it expands every time you increase the depth of nesting of your program structure and every time you increase the number of local variables in use.
Memory management

Since *BBC BASIC for Windows* can make much more memory available to user programs than was possible with *BBC BASIC (86)*, memory management is less of a concern. You will often not need to worry about running out of RAM: if the default amount allocated for the user's program and data is insufficient, the value of HIMEM can be increased.

However, there may still be situations where the amount of memory used by your program needs to be kept to a minimum. With careful design of your program, the size of both the stack and the heap can be reduced. Growth of the stack can be reduced by avoiding deeply nested structures and re-entrant routines. The number of LOCAL variables should be kept to a minimum (especially local arrays, which can use up a lot of stack space). Growth of the heap can be controlled by limiting the number of variables you use and by good string variable management. Again, arrays are particularly thirsty of memory.

Reducing stack usage

Some problems naturally lend themselves to a recursive solution. The classic example is the calculation of the factorial, which can be defined in a recursive fashion:

```plaintext
DEF FN_Factorial(N)
IF (N = 1) THEN = 1 ELSE = N * FN_Factorial(N-1)
```

Each time the function 'calls itself' the size of the stack increases, so the larger the number whose factorial is required the greater the amount of stack usage. By restructuring the problem in a non-recursive way the amount of stack used can be reduced, but at the expense of the size and readability of the program:

```plaintext
DEF FN_Factorial(N)
LOCAL I,F
F = 1
FOR I = N TO 1 STEP -1
   F = F * I
NEXT
= F
```
It must be stressed that this example is illustrative only. It is very unlikely that the amount of stack used in this case would be significant.

**Limiting the number of variables**

Each new variable occupies room on the heap. Restricting the length of the names of variables and limiting the number of variables used will limit the size of the heap. However, of the techniques available to you, this is the least rewarding. In addition, it leads to incomprehensible programs because your variable names become meaningless. If you compile your program to a standalone EXE, variable names are (by default) automatically abbreviated, so you gain the benefits without needing to modify your program.
Program storage in memory

The program is stored in memory in the format shown below. The first program line commences at PAGE.

<table>
<thead>
<tr>
<th>length</th>
<th>LS</th>
<th>MS</th>
<th>token</th>
<th>:</th>
<th>token</th>
<th>&amp;0D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

↑ Keyword Tokens ↑ CR

Line No ← Program Line →

Line length

The line length includes the line length byte itself. The address of the start of the next line is found by adding the line length to the address of the start of the current line. The end of the program is indicated by a line length of zero and a line number of &FFFF.

Line number

The line number is stored in two bytes, LSB first. The end of the program is indicated by a line number of &FFFF and a line length of zero. Valid line numbers are from 1 (&0001) to 65535 (&FFFF); a line number of zero signifies that the line is unnumbered.

Statements

Statements begin with a keyword, one of the symbols ‘*’, ‘=’ or ‘[’, or a variable name (in the case of an implied LET). Keywords are encoded as one-byte tokens wherever they occur; these have values in the ranges &80 to &FF and &01 to &10. Statements within a line are separated by colons.

Line terminator

Each program line is terminated by a carriage-return (&0D).
Variable storage in memory

Variables are held within memory as linked lists (chains). The first variable in each chain is accessed via an index which is maintained by BBC BASIC for Windows. There is an entry in the index for each of the characters permitted as the first letter of a variable name. Each entry in the index has a double-word (four bytes) address field which points to the first variable in the linked list with a name starting with its associated character. If there are no variables with this character as the first character in the name, the pointer is zero. The first four bytes of all variables holds the address of the next variable in the chain. The address in the last variable in the chain is zero. All addresses are held in the standard 80x86 format - LSB first.

The first variable created for each starting character is accessed via the index and subsequently created variables are accessed via the index and the chain. Consequently, there is some speed advantage to be gained by arranging for all your variables to start with a different character. If you compile your program to a standalone EXE, variable names are (by default) automatically distributed across the alphabet, so you gain the benefits without needing to modify your program.

Integer variable storage

Integers are held in two's complement format. They occupy 4 bytes or 8 bytes with the LSB first. Bit 7 of the MSB is the sign bit. To make up the complete variable, the address (link), the name and a separator (zero) byte are added to the value. The format of the memory occupied by a 32-bit integer variable called 'NUMBER%' is shown below. Note that since the first character of the name is found via the index, it is not stored with the variable.

```
<table>
<thead>
<tr>
<th>LS</th>
<th>MS</th>
<th>U</th>
<th>M</th>
<th>B</th>
<th>E</th>
<th>R</th>
<th>% &amp;00</th>
<th>LS</th>
<th>MS</th>
</tr>
</thead>
<tbody>
<tr>
<td>←</td>
<td>←</td>
<td>←</td>
<td>←</td>
<td>←</td>
<td>←</td>
<td>←</td>
<td>Value</td>
<td>←</td>
<td>←</td>
</tr>
</tbody>
</table>
```

Address of next variable starting with the same letter

The format of the memory occupied by a 64-bit integer variable called 'NUMBER%%' is shown below.
The smallest amount of space is taken up by a variable with a single letter name. The static integer variables, which are not included in the variable chains, use the names A% to Z%. Thus, the only single character names available for dynamic integer variables are a% to z% plus _% and ` % (CHR$(96)). As shown below, 32-bit integer variables with these names will occupy 10 bytes:

```
<table>
<thead>
<tr>
<th>LS</th>
<th>MS</th>
<th>%</th>
<th>&amp;00</th>
<th>LS</th>
<th>MS</th>
</tr>
</thead>
</table>
```

```
Address of next variable starting with the same letter
```

**Byte variable storage**

Byte variables are unsigned; they occupy one byte. To make up the complete variable, the address (link), the name and a separator (zero) byte are added to the value. The format of the memory occupied by a byte variable called 'NUMBER&' is shown below. Note that since the first character of the name is found via the index, it is not stored with the variable.

```
<table>
<thead>
<tr>
<th>LS</th>
<th>MS</th>
<th>U</th>
<th>M</th>
<th>B</th>
<th>E</th>
<th>R</th>
<th>&amp;</th>
<th>&amp;00</th>
<th>value</th>
</tr>
</thead>
</table>
```

```
Address of next variable starting with the same letter
```

**Variant numeric storage (40 bit)**

*BBC BASIC for Windows version 5.95a or earlier only*
Real numbers are held in binary floating point format. In the default (40-bit) mode the mantissa is held as a 4 byte binary fraction in sign and magnitude format. Bit 7 of the MSB of the mantissa is the sign bit. When working out the value of the mantissa, this bit is assumed to be 1 (a decimal value of 0.5). The exponent is held as a single byte in 'excess 127' format. In other words, if the actual exponent is zero, the value stored in the exponent byte is 127. To make up the complete variable, the address word, the name and a separator (zero) byte are added to the number. The format of the memory occupied by a real variable called 'NUMBER' is shown below.

<table>
<thead>
<tr>
<th>LS</th>
<th>MS</th>
<th>U</th>
<th>M</th>
<th>B</th>
<th>E</th>
<th>R</th>
<th>&amp;00</th>
<th>LS</th>
<th>MS</th>
<th>exp</th>
</tr>
</thead>
<tbody>
<tr>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td>←</td>
<td>→</td>
<td>←</td>
<td>Mantissa</td>
<td>→</td>
<td>↑</td>
<td>Exponent</td>
</tr>
</tbody>
</table>

Address of next variable starting with the same letter

As with integer variables, variables with single character names occupy the least memory (however, the names A to Z are available for variant numeric variables). Whilst a real variable requires an extra byte to store the number, the '%' character is not needed in the name. Thus, integer and real variables with the same name occupy the same amount of memory. However, this does not hold for arrays, since the name is only stored once.

In the following examples, the bytes are shown in the more human-readable manner with the MSB on the left.

The value 5.5 would be stored as shown below.

**Mantissa**

<table>
<thead>
<tr>
<th>.0011 0000 0000 0000 0000 0000 0000 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>↑ Sign Bit</td>
</tr>
<tr>
<td>&amp;30 00 00 00 &amp;82</td>
</tr>
</tbody>
</table>

Because the sign bit is assumed to be 1, this would become:

**Mantissa**

<table>
<thead>
<tr>
<th>.1011 0000 0000 0000 0000 0000 0000 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;B0 00 00 00 &amp;82</td>
</tr>
</tbody>
</table>

**Exponent**

<table>
<thead>
<tr>
<th>1000 0010</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;82</td>
</tr>
</tbody>
</table>
The equivalent in decimal is:

\[(0.5+0.125+0.0625) \times 2^{(130-127)}\]
\[= 0.6875 \times 2^3\]
\[= 0.6875 \times 8\]
\[= 5.5\]

*BBC BASIC for Windows* uses **variant** numeric variables which can hold either integers or floating-point values, allowing the faster integer arithmetic routines to be used if appropriate. The presence of an integer value in a variant numeric variable is indicated by the stored exponent being zero. Thus, if the stored exponent is zero, the 4 byte mantissa holds the number in normal integer format.

Depending on how it is put there, an integer value can be stored in a variant numeric variable in one of two ways. For example,

```
number=5
```

will set the exponent to zero and store the integer &00 00 00 05 in the mantissa. On the other hand,

```
number=5.0
```

will set the exponent to &82 and the mantissa to &20 00 00 00.

The two ways of storing an integer value are illustrated in the following four examples.
Example 1
\[ \text{number}=5 \quad & 00 \ 00 \ 00 \ 00 \ 05 \quad \text{Integer 5} \]

Example 2
\[ \text{number}=5.0 \quad & 82 \ 20 \ 00 \ 00 \ 00 \quad \text{Real 5.0} \]

This is treated as
\[ & 82 \ A0 \ 00 \ 00 \ 00 \]
\[ = (0.5+0.125)*2^{(130-127)} \]
\[ = 0.625*8 \]
\[ = 5 \]

because the sign bit is assumed to be 1.

Example 3
\[ \text{number}=-5 \quad & 00 \ FF \ FF \ FF \ FB \]

The 2's complement gives
\[ & 00 \ 00 \ 00 \ 00 \ 05 \quad \text{Integer -5} \]

Example 4
\[ \text{number}=-5.0 \quad & 82 \ A0 \ 00 \ 00 \ 00 \quad \text{Real -5.0} \]

(The sign bit is already 1)
\[ = (0.5+0.125)*2^{(130-127)} \]
\[ = 0.625*8 \]

Magnitude = 5

If all this seems a little complicated, try using the program below to accept a number from the keyboard and display the way it is stored in memory. The program displays the 4 bytes of the mantissa in 'human readable order' followed by the exponent byte. Look at what happens when you input first 5 and then 5.0 and you will see how this corresponds to the explanation given above. Then try -5 and -5.0 and then some other numbers. The program is an example of the use of the byte indirection
operator. See the Indirection section for details.

The layout of the variable 'NMBR' in memory is shown below.

```
| LS | MS | M | B | R | &00 | LS | MS | exp |
```

A%   A%+3   A%+4

REPEAT
   INPUT "Enter a number: " NMBR
   PRINT "& ";
   A% = ^NMBR
   REM Step through mantissa from MSB to LSB
   FOR I% = 3 TO 0 STEP -1
      REM Look at value at address A%+I%
      num$ = STR$~(A%?I%)
      IF LEN(num$)=1 num$="0"+num$
      PRINT num$; " ";
   NEXT
   REM Look at exponent at address A%+4
   num$ = STR$~(A%?4)
   IF LEN(num$)=1 num$="0"+num$
   PRINT " & +"+num$''
UNTIL NMBR=0

**Variant numeric storage (64 bit)**

*BBC BASIC for Windows version 5.95a or earlier only*

In *FLOAT 64 mode* variant variables are stored as 64-bit numbers in 8 bytes of memory (bits 0-7 in the first byte and bits 56-63 in the last byte). Bit 63 is the sign bit (0 for positive, 1 for negative). Bits 62 to 52 inclusive are the 11-bit exponent in offset-binary (‘excess 1024’) format, thus a value of 1024 (&400) represents an exponent of zero; exponent values of 0 (&000) and 2047 (&7FF) are not permitted. Bits 51 to 0 inclusive are the least-significant 52 bits of the mantissa (the MSB of the mantissa, bit 52, is not stored and is assumed to be a '1').

To make up the complete variable, the address word, the name and a separator (zero) byte are added to the number. The format of the memory occupied by a 64-bit real variable called 'NMBR' is shown below.
64-bit variant numeric variables are distinguished from 40-bit variant numeric variables by the addition of a # suffix character. This character is added automatically in *FLOAT 64 mode, but can be explicitly specified by the user when references to 64-bit variables are made in *FLOAT 40 mode.

If the most-significant 24 bits of the 64-bit value are all zero then the variable is assumed to contain a 40-bit real number. If the most-significant 32-bits of the 64-bit value are all zero then the variable is assumed to contain a 32-bit signed integer. In these cases the data format accords with those described earlier for 40-bit variants or 32-bit integers respectively.

**Variant numeric storage (80 bit)**

In *BBC BASIC for Windows version 6.00a or later* variant variables are stored as 80-bit numbers occupying 10 bytes of memory. Floating-point values are stored in 'extended precision' format (64-bits mantissa with an explicit MSB, 15-bits 'excess 16383' exponent and a sign bit). If the most-significant 16 bits (exponent plus sign bit) are all zero then the variable is assumed to contain a 64-bit signed integer in the 'mantissa'.

To make up the complete variable, the address word, the name and a separator (zero) byte are added to the number. The format of the memory occupied by an 80-bit real variable called 'NUMBR' is shown below.
String variable storage

String variables are stored as the string of characters. Since the length of the string is stored in memory an explicit terminator for the string is unnecessary. As with numeric variables, the first double-word is the address of the next variable starting with the same character. However, since BBC BASIC for Windows needs information about the length of the string and the address in memory where it starts, the overheads for a string are more than for a numeric. The format of a string variable called 'NAME$' is shown below.

**BBC BASIC for Windows version 5.95a or earlier:**

<table>
<thead>
<tr>
<th>LS</th>
<th>MS</th>
<th>U</th>
<th>M</th>
<th>B</th>
<th>R</th>
<th>&amp;00</th>
<th>LS</th>
<th>MS</th>
<th>LS</th>
<th>MS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rest of name</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Address of next variable starting with the same letter

BBC BASIC for Windows version 6.00a or later:

<table>
<thead>
<tr>
<th>LS</th>
<th>MS</th>
<th>A</th>
<th>M</th>
<th>E</th>
<th>$</th>
<th>&amp;00</th>
<th>LS</th>
<th>MS</th>
<th>LS</th>
<th>MS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rest of name</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>String start address</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Length</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Address of next variable starting with the same letter

The amount of memory allocated for the string depends on the current length of the string, according to the following formula:
allocated_length = 2^\text{INT}(\log_2 (\text{current_length})+1)-1

So long as the length of the string is compatible with the allocated length, the string will be stored at the same address. If the variable is set to a string longer than this maximum length there will be insufficient room in the original position for the characters of the string. When this happens, the new string will be placed in a block of the correct length taken from the string free list or, failing that, on the top of the heap; its new start address will be loaded into the address bytes. The previous space occupied by the string is added to the free list. The same thing happens when the length of the string is reduced below the minimum value corresponding to the allocated length, otherwise wasted memory ('garbage') would result.

For example if the allocated length is 2047 bytes \((2^{11}-1)\) the string will be stored at the same address so long as its current length remains between 1024 and 2047 characters. If its length is reduced below 1024 characters or increased above 2047 characters the string will be moved to a new address and the 2047-byte block of memory added to the string free list.

**Structure storage**

The format of a structure in memory consists of three parts:

- A header containing the name of the structure (excluding the first character) and pointers to the format and data blocks.
- A format block containing the list of structure members.
- The structure's data.

The format of the structure header is very similar to that of an ordinary variable, except that instead of a data value it includes two 32-bit pointers (links) to the format and data blocks. The header of a structure called STRU\{ \} would be stored as follows:

```
<table>
<thead>
<tr>
<th>LS</th>
<th>MS</th>
<th>T</th>
<th>R</th>
<th>U</th>
<th>{ &amp;00</th>
<th>LS</th>
<th>MS</th>
<th>LS</th>
<th>MS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Address of next variable starting with the same letter</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td>←</td>
<td>Rest of Name</td>
<td>←</td>
<td>Format block address</td>
<td>Data block address</td>
<td></td>
</tr>
</tbody>
</table>
```

Page 737
Note that the stored name includes the left brace and it is this which identifies the heap entry as a structure.

The format block is of variable length, and consists of a 32-bit value containing the total data size (in bytes) followed by a linked-list of structure members. The linked list has the same format as the main variable lists (chains) except that instead of actual data values it contains 32-bit offsets into the data block. The format block for the structure STRU{a,b} would be stored in memory as follows (in BBC BASIC for Windows version 6.00a or later):

```
| 20 | 0 | 0 | 0 | LS | MS | &61 &00 | 0 | 0 | 0 | 0 |

- Total data size  ←  Link to next  ←  Name  ←  Offset into data  →
```

```
&00 &00 &00 &62 &00 0 0 0 0

- Final link  ←  Name  ←  Offset into data  →
```

### Fixed strings

You can place a string starting at a given location in memory using the indirection operator '$'. For example,

```basic
DIM S% 256
$S% = "This is a string"
```

would place &54 (T) at address S%, &68 (h) at address S%+1 etc. Because the string is placed at a predetermined location in memory it is called a 'fixed' string. Fixed strings are not included in the variable chains and they do not have the overheads associated with a string variable. However, since the length of the string is not stored, an explicit terminator (&0D) is used. Consequently, in the above example, byte S%+16 would be set to &0D. Fixed strings are restricted in length to 65535 bytes (65536 bytes including the terminating &0D).

If you use $$ rather than $ the string in memory is NUL-terminated (&00) rather than CR-terminated (&0D). So for example:

```basic
DIM S% 256
$$S% = "This is a string"
```
would set byte S%+16 to &00.

Array storage

The format of an array in memory consists of three parts:

- A header containing the name of the array (excluding the first character) and a pointer to the parameter block.
- A parameter block containing the number of dimensions and the size of each dimension.
- The array data (integer, byte, variant numeric or string).

The format of the array header is very similar to that of an integer variable, except that instead of a 32-bit data value it includes a 32-bit pointer (link) to the parameter block. The header of an array called ARRAY%() would be stored as follows:

```
| LS | MS | R | R | A | Y | % | ( | &00 | LS | MS |
```

Address of next variable starting with the same letter

Note that the stored name includes the left bracket (parenthesis) and it is this which identifies the heap entry as an array.

The parameter block is of variable length, and consists of a single byte containing the number of dimensions (suffixes) and four bytes for each of the dimensions, containing the size of that dimension (number of rows, columns etc). The parameter block for the array ARRAY%(10,20) would be stored in memory as follows:

```
2 11 0 0 0 21 0 0 0
```

Number of dimensions

Note that the size of each dimension is equal to one greater than the suffix specified in the DIM statement, since the index can take any value from zero to the specified maximum suffix.
The array data follows immediately after the parameter block, the number of elements being equal to the product of the sizes of each dimension. For example in the case of ARRAY%(10,20) the data consists of 11*21 = 231 values. Each data value consists of either one byte (in the case of a byte array), four bytes (in the case of a 32-bit integer numeric array), five bytes (in the case of a 40-bit variant numeric array), six bytes (in the case of a version 5 string array), eight bytes (in the case of a 64-bit integer array, 64-bit variant/double numeric array, structure array or version 6 string array) or ten bytes (in the case of an 80-bit variant numeric array).
Introduction to the Windows API

*BBC BASIC for Windows* allows you to access the Windows™ Application Program Interface (API) by means of the SYS statement. There are many hundreds of API functions, the vast majority of which are unlikely to be of use to a BASIC program, and no attempt will be made to provide a comprehensive list here. Further details can be found in Microsoft documentation and on their web site.

The SYS statement allows you to call an API function either by name or by specifying its memory address. Only the most commonly used functions can be called by name, specifically those which are present in the following Dynamic Link Libraries, which are automatically loaded into memory when *BBC BASIC for Windows* is run:

- ADVAPI32.DLL
- COMCTL32.DLL
- COMDLG32.DLL
- GDI32.DLL
- KERNEL32.DLL
- SHELL32.DLL
- USER32.DLL
- WINMM.DLL

Functions in other DLLs must be explicitly loaded into memory, and must be called by address rather than by name. For example to call the function *OleUIChangeIconA* which is in *OLEDLG.DLL* you must perform the following steps:

```basic
SYS "LoadLibrary", "OLEDLG.DLL" TO oledlg%
SYS "GetProcAddress", oledlg%, "OleUIChangeIconA" TO chicon%
SYS chicon%, ci% TO uint%
```

Once you have finished with them, it is important to 'release' any DLLs which you have loaded:

```basic
SYS "FreeLibrary", oledlg%
```

To ensure that the DLLs are released however your program exits, include code
similar to the following in a 'cleanup' routine called from your ON CLOSE and ON ERROR handlers:

```plaintext
oledlg% += 0 : IF oledlg% THEN SYS "FreeLibrary", oledlg%
```
Changing the window title

Normally, the text in the title bar of BASIC's output window is set to the name of the BASIC program itself. You can change the title by using the SetWindowText function:

```
title$ = "New title"
SYS "SetWindowText", @hwnd$, title$
```
Flashing the title bar

You can control whether the title bar of BASIC's output window is highlighted (which usually means the window will accept keyboard input) or not by using the FlashWindow function. This can be used, for example, to alert the user that your program requires some input:

```basic
SYS "FlashWindow", @hwnd%, 1
WAIT 20
SYS "FlashWindow", @hwnd%, 0
```

If the final parameter is 1 the state of the title bar is inverted (if it was highlighted it is un-highlighted; if it was not highlighted it becomes highlighted). If the final parameter is 0 the title bar is returned to its original state.

You will probably want to flash the title bar more than once to attract attention.
Finding the display size

You may wish to know the width and height of the desktop area in order to select an appropriate screen mode for your program. You can do that using the GetSystemMetrics function:

```
SYS "GetSystemMetrics", 0 TO xscreen%
SYS "GetSystemMetrics", 1 TO yscreen%
```

The above program segment will load the variable `xscreen%` with the width of the desktop and the variable `yscreen%` with the height of the desktop, both in pixels.
Displaying a message box

If you want to display a message in its own 'message window', you can use the MessageBox function:

```vbscript
message$ = "Test message"
caption$ = "Test caption"
SYS "MessageBox", @hwnd%, message$, caption$, 0
```

The final numeric value determines what kind of symbol is displayed and what options are presented to the user:

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>Stop symbol</td>
</tr>
<tr>
<td>32</td>
<td>Question mark</td>
</tr>
<tr>
<td>48</td>
<td>Exclamation mark</td>
</tr>
<tr>
<td>64</td>
<td>Information symbol</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>User options</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>OK</td>
</tr>
<tr>
<td>1</td>
<td>OK and Cancel</td>
</tr>
<tr>
<td>2</td>
<td>Abort, Retry and Ignore</td>
</tr>
<tr>
<td>3</td>
<td>Yes, No and Cancel</td>
</tr>
<tr>
<td>4</td>
<td>Yes and No</td>
</tr>
<tr>
<td>5</td>
<td>Retry and Cancel</td>
</tr>
<tr>
<td>6</td>
<td>Cancel, Try Again and Continue (Windows 2000 or later only)</td>
</tr>
</tbody>
</table>

The 'symbol' value and the 'options' value should be added together. When more than one option is offered, you can tell which was selected by storing the return value as follows:

```vbscript
SYS "MessageBox", @hwnd%, message$, caption$, 0 TO result%
```

The value of result% will be one of the following:
**Value Selection**

1. OK
2. Cancel
3. Abort
4. Retry
5. Ignore
6. Yes
7. No
10. Try Again
11. Continue

*Note*: If a dialogue box is open when you display the message, you should normally specify as the first parameter not `@hwnd%` but the window handle of the dialogue box itself, for example:

```
SYS "MessageBox", !dlg%, message$, caption$, 0
```

where `dlg%` is the value returned from `FN_newdialog` (note the exclamation point).
Updating the screen

* BBC BASIC for Windows doesn't always update the screen immediately a change is made (for example if a line is drawn). If several things are being plotted in quick succession it is more efficient to wait until all the changes have been made and then update the screen. This is normally perfectly satisfactory, but there may be special circumstances (for example if displaying an animated graphic) when you need to force the display to update immediately. You can do that by calling the UpdateWindow function:

```
SYS "UpdateWindow", @hwnd%
```

See also the *REFRESH command.
Producing a warning sound

Although you can produce your own sounds using the SOUND and ENVELOPE statements, your program can also produce one of the standard Windows™ warning sounds by using the MessageBeep function:

```plaintext
beep% = 0
SYS "MessageBeep", beep%
```

The sound is determined by the value of `beep%`, as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Sound</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Default sound</td>
</tr>
<tr>
<td>16</td>
<td>Critical stop</td>
</tr>
<tr>
<td>32</td>
<td>Question</td>
</tr>
<tr>
<td>48</td>
<td>Exclamation</td>
</tr>
<tr>
<td>64</td>
<td>Asterisk</td>
</tr>
</tbody>
</table>

The actual sound produced will depend on the user's `sound scheme` as selected in the Windows™ Control Panel.
Playing WAV files

If you want to play a sound which is stored in a standard WAV file, you can do that using the PlaySound function:

```plaintext
SND_ASYNC = 1
SND_FILENAME = &20000
wave$ = "\windows\media\tada.wav"
SYS "PlaySound", wave$, 0, SND_FILENAME + SND_ASYNC
```

You can control the playback volume as follows:

```plaintext
SYS "waveOutSetVolume", -1, volume% + (volume% << 16)
```

where `volume%` is in the range 0 (minimum) to 65535 (maximum). Note that the volume change will affect all subsequent wave audio output.

Alternatively you can load the WAV file into memory and then play the sound from there. You can load the file using the following code:

```plaintext
wave$ = "\windows\media\tada.wav"
file% = OPENIN(wave$)
size% = EXT#file%
CLOSE #file%
DIM tada% size%-1
OSCLI "LOAD "+wave$+" "+STR$~tada%
```

Then you can play the sound as many times as you like with:

```plaintext
SND_ASYNC = 1
SND_MEMORY = 4
SYS "PlaySound", tada%, 0, SND_MEMORY + SND_ASYNC
```

You can abort a sound which is already playing as follows:

```plaintext
SYS "PlaySound", 0, 0, 0
```

The PlaySound function also allows you to play 'system sounds':
SND_ASYNC = 1
SND_ALIAS = &10000
SYS "PlaySound", "SystemAsterisk", 0, SND_ALIAS + SND_ASYNC
SYS "PlaySound", "SystemExclamation", 0, SND_ALIAS + SND_ASYNC
SYS "PlaySound", "SystemExit", 0, SND_ALIAS + SND_ASYNC
SYS "PlaySound", "SystemHand", 0, SND_ALIAS + SND_ASYNC
SYS "PlaySound", "SystemQuestion", 0, SND_ALIAS + SND_ASYNC
SYS "PlaySound", "SystemStart", 0, SND_ALIAS + SND_ASYNC

Note that the SND_ALIAS signifies that a system sound should be played, whereas
SND_FILENAME signifies that a WAV file should be played and SND_MEMORY signifies
a sound in memory.

The PlaySound function may not work if your program has previously executed a
SOUND statement. In that case, use SOUND OFF before calling PlaySound.
Checking for a sound card

If no suitable sound card is fitted, using the SOUND statement will result in the trappable Device unavailable error. Although it is rare to find a modern PC without a sound card, you may wish your programs to work even if one is not installed. You can test for the presence of a sound card using the waveOutGetNumDevs function:

```
SYS "waveOutGetNumDevs" TO ndevs%
```

If `ndevs%` is non-zero it should be safe to use the SOUND statement, although it can still fail if (for example) another program is currently using the sound system.
Timing program execution

You can discover how long Windows™ has been running by calling the GetTickCount function; this can be a useful adjunct to the built in TIME pseudo-variable:

    SYS "GetTickCount" TO tick%

The value of **tick%** will be set to the number of milliseconds since Windows was started (it wraps around to zero if Windows has been running continuously for approximately 49 days and 17 hours!).
Pausing a program

A common way of pausing a program under software control is to use the INKEY function:

```vbnet
pause = INKEY(delay%)
```

or to use the TIME pseudo-variable:

```vbnet
TIME = 0
REPEAT UNTIL TIME >= delay%
```

However both of these methods have their disadvantages. The INKEY delay can be truncated by pressing a key, which may be undesirable, and the TIME method keeps the processor fully occupied, so other applications will run slowly whilst your program is paused. A better method is to use WAIT:

```vbnet
WAIT delay%
```

This is probably the best method for long delays, but an alternative is to use the Sleep function:

```vbnet
SYS "Sleep", delay%
```

The program will pause for approximately delay% milliseconds. Note that during this time the ESCape key is not tested, nor can the window be closed. Therefore this method should be used only for short delays.
Reading the command line

You can discover the command which was issued to execute *BBC BASIC for Windows* (or your executable file if generated using the Compile utility) by calling the GetCommandLine function:

```
SYS "GetCommandLine" TO cmdline%
cmdline$ = $$cmdline%
```

The command line is returned in memory and is terminated with a NUL character (CHR$0). The $$ indirection operator converts it to a normal BASIC string.

If you simply want to know the command line 'tail' (i.e. everything after the filename) you can use the system variable @cmd$.

Reading a NUL-terminated string

*BBC BASIC for Windows* versions 5.10a and later support NUL-terminated strings directly using the $$ indirection operator. Earlier versions can use the following function to convert a NUL-terminated string in memory to a normal BASIC string:

```
DEF FNnulterm$(P%)
LOCAL A$
WHILE ?P%
   A$ += CHR$?P%
   P% += 1
ENDWHILE
= A$
```
Finding the filename

If, rather than the entire command line, you just want know the path and filename of *BBC BASIC for Windows* (or of your executable program) you can use the GetModuleFileName function:

```basic
DEF FNgetmodulefilename
LOCAL filename%
DIM filename% LOCAL 260
SYS "GetModuleFileName", 0, filename%, 260
= $$filename%
```

To discover the directory from which your program was loaded you can use the `@dir$` system variable.

Discovering an 'unknown error'

If Windows reports an error condition which BASIC was not expecting, an Unknown error (error code 255) results. You can discover the true cause of the error by using the GetLastError and FormatMessage functions:

```
DEF FNwinerror
LOCAL message%, winerr%
DIM message% LOCAL 255
SYS "GetLastError" TO winerr%
SYS "FormatMessage", &1000, 0, winerr%, 0, message%, 255, 0
= $$message%
```
Repositioning the window

You can set the position of BASIC's output window without changing its size by calling the SetWindowPos function:

```plaintext
SWP_NOSIZE = 1
SWP_NOZORDER = 4
SYS "SetWindowPos", @hwnd%, 0, xpos%, ypos%, 0, 0, \ 
   SWP_NOSIZE + SWP_NOZORDER
```

The position is specified as the offset, in pixels, from the top-left corner of the desktop to the top-left corner of BASIC's output window. To change the window's size without moving it you can do the following:

```plaintext
SWP_NOMOVE = 2
SWP_NOZORDER = 4
SYS "SetWindowPos", @hwnd%, 0, 0, 0, width%, height%, \ 
   SWP_NOMOVE + SWP_NOZORDER
VDU 26
```

The width and height are specified in pixels, including the border and the title bar. If want to specify the dimensions excluding the border and title bar, i.e. of the region usable by BASIC, then the best way of doing it is to select a user-defined mode with VDU 23,22... . However an alternative method is as follows:

```plaintext
DIM rc{l%,t%,r%,b%}
rc.l% = 0
rc.t% = 0
rc.r% = width%
rc.b% = height%
SYS "AdjustWindowRect", rc{}, &CF0000, 0
SYS "SetWindowPos", @hwnd%, 0, 0, 0, rc.r%-rc.l%, rc.b%-rc.t%, 6
VDU 26
```

If you want to alter both the size and the position you can use the MoveWindow function:

```plaintext
SYS "MoveWindow", @hwnd%, xpos%, ypos%, width%, height%, 1
VDU 26
```
Whenever you intentionally change the window size, you should reset BASIC's text and graphics clipping regions to the new size by issuing the VDU 26 command.

To discover the current size of the window you can use the GetClientRect function, which returns the width and height in pixels:

```basic
DIM rc{l%,t%,r%,b%}
SYS "GetClientRect", @hwnd%, rc{}
Width% = rc.r%
Height% = rc.b%
```

This gives the usable size of the window, i.e. excluding the title bar etc. Should you need to know the full size (and position) you can do the following:

```basic
DIM rc{l%,t%,r%,b%}
SYS "GetWindowRect", @hwnd%, rc{}
Xpos% = rc.l%
Ypos% = rc.t%
Width% = rc.r% - rc.l%
Height% = rc.b% - rc.t%
```

To find the 'normal' size of the window, even if it is maximised or minimised, you can use the GetWindowPlacement function:

```basic
DIM wp{length%,flags%,showcmd%, minpos{x%,y%}, \\ maxpos{x%,y%}, normal{l%,t%,r%,b%})
wp.length% = DIM(wp{})
SYS "GetWindowPlacement", @hwnd%, wp{}
Width% = wp.normal.r% - wp.normal.l%
Height% = wp.normal.b% - wp.normal.t%
```

Don't use the window position returned by GetWindowPlacement since it is in workspace (not screen ) coordinates and probably won't be what you want if the Taskbar is at the top of the screen.
Fixing the window size

Normally the user can re-size the output window by dragging the side or corner to a new position. However there may be circumstances when you would prefer to fix the window size, and prevent it from being changed by the user. You can do that as follows:

```
GWL_STYLE = -16
WS_THICKFRAME = &40000
WS_MAXIMIZEBOX = &10000
SYS "GetWindowLong", @hwnd%, GWL_STYLE TO ws%
SYS "SetWindowLong", @hwnd%, GWL_STYLE, ws% AND NOT WS_THICKFRAME \ 
\ AND NOT WS_MAXIMIZEBOX
MODE 8
```

If necessary change the **MODE 8** to whichever MODE is required, or replace it with a suitable VDU 23,22 command. It is important to execute a MODE or VDU 23 statement **after** changing the window style.
Minimising or maximising the window

You can minimise BASIC’s output window under program control using the ShowWindow function:

```
SW_MINIMIZE = 6
SYS "ShowWindow", @hwnd%, SW_MINIMIZE
```

This will have exactly the same effect as clicking on the minimise button in the title bar, or right-clicking on the title bar and selecting **Minimize** from the menu. The ShowWindow function can also be used to restore the window to its original size and position, maximise the window (equivalent to clicking on the maximise button) or hide the window (so it appears neither on the desktop nor in the taskbar):

```
SW_RESTORE = 9
SYS "ShowWindow", @hwnd%, SW_RESTORE
SW_MAXIMIZE = 3
SYS "ShowWindow", @hwnd%, SW_MAXIMIZE
SW_HIDE = 0
SYS "ShowWindow", @hwnd%, SW_HIDE
```

If you maximise the window you should normally use a VDU 26 to reset the text and graphics viewports so they fill the full area:

```
SW_MAXIMIZE = 3
SYS "ShowWindow", @hwnd%, SW_MAXIMIZE
VDU 26
```

If you hide the window, you are liable to confuse the **BBC BASIC for Windows** IDE when you exit your program. To prevent this you should execute the following code before quitting:

```
SW_NORMAL = 1
SYS "ShowWindow", @hwnd%, SW_NORMAL
SYS "SetForegroundWindow", @hwnd%
```
Forcing the window to stay on top

You can set your window to be 'topmost', thus forcing it to appear on top of all other non-topmost windows, by using the SetWindowPos function:

```plaintext
HWND_TOPMOST = -1  
SWP_NOSIZE = 1  
SWP_NOMOVE = 2
SYS "SetWindowPos", @hwnd%, HWND_TOPMOST, 0, 0, 0, 0, SWP_NOSIZE+SWP_NOMOVE
```

The above example leaves the window position and size unchanged, but you can if you like change these at the same time:

```plaintext
SYS "SetWindowPos", @hwnd%, HWND_TOPMOST, x%, y%, cx%, cy%, 0
VDU 26
```

Here x%, y% is the position and cx%, cy% the size (using the same units as described previously for Repositioning the window).

If you don't need your window to stay on top, but simply want to move it to the top, you can use the BringWindowToTop function:

```plaintext
SYS "BringWindowToTop", @hwnd%
```
Removing the title bar

You can remove the title bar and system menu from the output window as follows:

```
GWL_STYLE = -16
WS_BORDER = &800000
SWP_NOSIZE = 1
SWP_NOMOVE = 2
SWP_NOZORDER = 4
SWP_FRAMECHANGED = &20
SYS "GetWindowLong", @hwnd%, GWL_STYLE TO ws%
SYS "SetWindowLong", @hwnd%, GWL_STYLE, ws% AND NOT WS_BORDER
SYS "SetWindowPos", @hwnd%, 0, 0, 0, 0, 0, SWP_NOSIZE +
\                     SWP_NOMOVE + SWP_NOZORDER + SWP_FRAMECHANGED
```

As this removes the close button, make sure you provide an obvious means for quitting the program (Alt-F4 will still work so long as you haven't used ON CLOSE).
Using the entire screen

If you want your program to run 'full screen', without even a title bar or borders, you can do that as follows:

```plaintext
GWL_STYLE = -16
HWND_TOPMOST = -1
WS_VISIBLE = &10000000
WS_CLIPCHILDREN = &2000000
WS_CLIPSIBLINGS = &4000000
SYS "GetSystemMetrics", 0 TO xscreen%
SYS "GetSystemMetrics", 1 TO yscreen%
SYS "SetWindowLong", @hwnd%, GWL_STYLE, WS_VISIBLE + \\
    WS_CLIPCHILDREN + WS_CLIPSIBLINGS
SYS "SetWindowPos", @hwnd%, HWND_TOPMOST, 0, 0, xscreen%, yscreen%, 0
VDU 26
```

When you run your program not even the Windows™ taskbar will be visible, nor will the user be able to re-size or move the window, so make sure you provide an obvious means for quitting the program (Alt-F4 will still work so long as you haven't used ON CLOSE).

Bear in mind that display resolutions vary, from 640 x 480 pixels to 1920 x 1440 pixels or more, and widescreen (16:9) displays are becoming increasingly common. Ideally you should write your program in such a way that it will behave sensibly with any display resolution and shape. If that isn't practical you can check that the dimensions are suitable (xscreen% and yscreen% in the above example contain the width and height in pixels) and prompt the user if not.

In the unlikely event that the screen dimensions exceed 1920 x 1440 pixels it is important that you do not issue the VDU 26 listed above. Instead you should use the code provided at Using windows larger than 1920 x 1440 pixels.
Drawing graphics to a printer

One way of outputting graphics to a printer is to first display them on the screen, then print them using *HARDCOPY. However this approach has a major disadvantage: the quality of the graphics is limited to the quality of the screen display (printers typically have resolutions of 600 or more dots-per-inch, compared to a typical value for the screen of 96 dpi). Whilst this can be partially overcome by creating a sufficiently large output 'canvas' (which can be bigger than your screen) it still imposes some limitations.

The solution is to output the graphics directly to the printer. Unfortunately you cannot use BASIC's built-in graphics statements to do this, you must use the Windows™ API instead. However this is quite easy, and there are equivalents for all of the normal graphics operations, for example:

<table>
<thead>
<tr>
<th>BASIC statement</th>
<th>Printer API equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE x,y</td>
<td>SYS &quot;MoveToEx&quot;,@prthdc%,x,y,0</td>
</tr>
<tr>
<td>DRAW x,y</td>
<td>SYS &quot;LineTo&quot;,@prthdc%,x,y</td>
</tr>
<tr>
<td>RECTANGLE x,y,dx,dy</td>
<td>SYS &quot;Rectangle&quot;,@prthdc%,x1,y1,x2,y2</td>
</tr>
<tr>
<td>ELLIPSE x,y,a,b</td>
<td>SYS &quot;Ellipse&quot;,@prthdc%,x1,y1,x2,y2</td>
</tr>
</tbody>
</table>

The main differences are that the coordinate system is different (vertical coordinates are measured downwards, with the origin at the top of the page) and the resolution in dots-per-inch varies between printers. If you only ever expect to output to a particular printer then you can use absolute coordinates, but it is far better to scale your graphics output so that it will look correct whatever the printer. There are two ways of achieving that, scaling to a particular size (in inches or cm) or scaling to fit the size of the paper.

To scale your printed graphics to a particular size you can find the resolution of the printer in dots-per-inch as follows:

```plaintext
_LOGPIXELSX = 88
_LOGPIXELSY = 90
SYS "GetDeviceCaps", @prthdc%, _LOGPIXELSX TO dpix%
SYS "GetDeviceCaps", @prthdc%, _LOGPIXELSY TO dpiy%
```

The two values will often be the same. To scale your graphics to fit the paper you can discover the coordinates of the left, right, top and bottom margins as follows:
As usual you need to output at least one conventional character to the printer at the start; you may want to print a title anyway, but if not you can simply output a space. The following code draws a straight line diagonally across the page, according to the current *MARGINS setting:

```
VDU 2,1,32,3
SYS "MoveToEx", @prthdc%, @vdu%!232, @vdu%!240, 0
SYS "LineTo", @prthdc%, @vdu%!236, @vdu%!244
VDU 2,1,12,3
```

You may find that the line is too thin, or you want to print it in a colour other than black. You can change the line thickness and colour as follows:

```
VDU 2,1,32,3
pcol% = red% + (green% << 8) + (blue% << 16)
SYS "CreatePen", 0, thickness%, pcol% TO pen%
SYS "SelectObject", @prthdc%, pen%
SYS "MoveToEx", @prthdc%, @vdu%!232, @vdu%!240, 0
SYS "LineTo", @prthdc%, @vdu%!236, @vdu%!244
VDU 2,1,12,3
```

The colour must be specified as the amounts of red, green and blue, in the range 0 (none) to 255 (maximum).
Printing coloured text

If you have a colour printer, you may want to print text in different colours. You can do this using the SetTextColor function:

\[
pcol\% = \text{red}\% + (\text{green}\% \ll 8) + (\text{blue}\% \ll 16)
\]

SYS "SetTextColor", @prthdc\%, pcol\%

This sets the colour of subsequently printed text according to the amounts of red, green and blue specified, in the range 0 (none) to 255 (maximum). For example:

SYS "SetTextColor", @prthdc\%, &FF0000

would result in blue text.

You should ensure that at least one text character is sent to the printer before issuing the SetTextColor call, otherwise it might not be effective. You may want to print some black text anyway but if not you can simply send an initial space character:

VDU 2,1,32,3
Printing in landscape format

You may want to produce your printed output in landscape orientation (i.e. with the long edge of the paper horizontal). You can achieve that by executing the following procedure before outputting data to the printer:

```
DEF PROC landscape
LOCAL psd%, dm%
DIM psd% LOCAL 83
!psd% = 84
psd%!16 = 1024
SYS "PageSetupDlg", psd%
SYS "GlobalLock", psd%!8 TO dm%
dm%!40 = 1
dm%!44 = 2
SYS "ResetDC", @prthdc%, dm%
SYS "GlobalUnlock", psd%!8
SYS "GlobalFree", psd%!8
SYS "GlobalFree", psd%!12
*MARGINS 10,10,10,10
ENDPROC
```

The *MARGINS command is necessary for BBC BASIC for Windows to take note of the changed page dimensions. If you want to set the margins to values other than the default (10 mm), specify the appropriate values in the command.

To set the format back to portrait, change the 2 to 1 in the ninth line:

```
dm%!44 = 1
```

As an alternative, and possibly more reliable, method you can present the user with the Print, Print Setup or Page Setup dialogue box which will allow him not only to select the required orientation but also the printer, printer options (e.g. print quality) and paper size.
Adding a menu bar

A rather more complicated, but very valuable, use for the Windows™ API functions is to add a menu bar to your program. Below is a complete program which uses the CreateMenu, SetMenu, AppendMenu and DrawMenuBar functions to implement a simple menu. Clicking on the menu items causes the background colour of the program window to change:

```sys
SYS "CreateMenu" TO hmenu%
SYS "SetMenu", @hwnd%, hmenu%
SYS "AppendMenu", hmenu%, 0, 0, "Black"
SYS "AppendMenu", hmenu%, 0, 1, "&Red"
SYS "AppendMenu", hmenu%, 0, 2, "&Green"
SYS "AppendMenu", hmenu%, 0, 3, "&Yellow"
SYS "AppendMenu", hmenu%, 0, 4, "&Blue"
SYS "AppendMenu", hmenu%, 0, 5, "&Magenta"
SYS "AppendMenu", hmenu%, 0, 6, "&Cyan"
SYS "AppendMenu", hmenu%, 0, 15, "&White"
SYS "DrawMenuBar", @hwnd%
:
Click% = -1
ON SYS Click% = @wparam% : RETURN
:
REPEAT
  WAIT 1
  click% = -1
  SWAP click%,Click%
  IF click%<>-1 THEN COLOUR 128+click% : CLS
UNTIL FALSE
END
```

The program creates a menu bar containing eight items, consisting of the colour names listed. In each case one letter of the name is preceded by an ampersand (&): this determines the keyboard shortcut letter associated with the menu item.

The ON SYS statement is activated whenever a menu item is selected (either by clicking with the mouse or using the keyboard shortcut). The system variable @wparam% contains a menu identifier which is equal to the third parameter of the relevant AppendMenu function (these values have been chosen to correspond directly to colour numbers), and this is copied to the global variable Click%.

The value of Click% is monitored by polling it within the program's main loop; when its value has been modified by the menu selection the background colour is changed.
If for any reason you want to remove the menu bar, you can do that as follows:

```c
SYS "SetMenu", hwnd%, 0
```

however in that case you should make sure you destroy the menu before quitting your program, so that the memory it uses is released:

```c
SYS "DestroyMenu", hmenu%
```
Adding popup and sub-menus

The foregoing example simply generates a menu bar with a number of clickable items. More usually these top-level items activate 'popup' (or 'drop down') menus each containing a set of selectable items (or their own sub-menus). The following example program is functionally compatible with the previous one, but the colour selection is provided with popup and sub-menus:

```
SYS "CreatePopupMenu" TO hsub%
SYS "AppendMenu", hsub%, 0, 0, "&Black"
SYS "AppendMenu", hsub%, 0, 15, "&White"

SYS "CreatePopupMenu" TO hpop1%
SYS "AppendMenu", hpop1%, 0, 1, "&Red"
SYS "AppendMenu", hpop1%, 0, 2, "&Green"
SYS "AppendMenu", hpop1%, 0, 4, "&Blue"

SYS "CreatePopupMenu" TO hpop2%
SYS "AppendMenu", hpop2%, 0, 3, "&Yellow"
SYS "AppendMenu", hpop2%, 0, 5, "&Magenta"
SYS "AppendMenu", hpop2%, 0, 6, "&Cyan"
SYS "AppendMenu", hpop2%, 16, hsub%, "&Others"

SYS "CreateMenu" TO hmenu%
SYS "AppendMenu", hmenu%, 16, hpop1%, "&Primary"
SYS "AppendMenu", hmenu%, 16, hpop2%, "&Secondary"
SYS "SetMenu", @hwnd%, hmenu%
SYS "DrawMenuBar", @hwnd%

Click% = -1
ON SYS Click% = @wparam% : RETURN

REPEAT
  WAIT 1
  click% = -1
  SWAP click%, Click%
  IF click%<>-1 THEN COLOUR 128+click% : CLS
UNTIL FALSE
END
```

Note that in the AppendMenu calls which specify popup or sub-menus the second parameter is set to 16 rather than to zero. In this case the third parameter is set to the handle of the popup menu (hpop1% or hpop2%) or sub-menu (hsub%) rather than an arbitrary ID number.

To insert one or more separators (horizontal dividing lines) within the popup or sub-
menu, add the following statement at the appropriate point(s):

SYS "AppendMenu", hpop2%, &800, 0, 0
Ticking a menu item

You can place (or remove) a tick mark against an item in a popup menu by using the CheckMenuItem function:

```
SYS "CheckMenuItem", hpopup%, itemid%, 8
SYS "CheckMenuItem", hpopup%, itemid%, 0
```

The first parameter is the handle of the popup menu (as returned by CreatePopupMenu), the second parameter is the menu item identifier (specified as the third parameter of AppendMenu) and the third parameter is 8 to add a tick mark or 0 to remove the tick mark.
Disabling a menu item

You can disable (or enable) an item in a popup menu by using the EnableMenuItem function:

```
SYS "EnableMenuItem", hpopup%, itemid%, 0
SYS "EnableMenuItem", hpopup%, itemid%, 1
```

The first parameter is the handle of the popup menu (as returned by CreatePopupMenu), the second parameter is the menu item identifier (specified as the third parameter of AppendMenu) and the third parameter is 0 to enable the item and 1 to disable the item (and change it to grey to indicate that it is disabled). You can also use the value 2 which disables the item but does not make it grey.

An alternative method of specifying the item you want to disable (or enable) is to give its position in the menu rather than its ID; this is particularly useful when the item is a sub-menu (which doesn't have an ID as such). To do this you add &400 to the third parameter:

```
SYS "EnableMenuItem", hpopup%, itempos%, &400
SYS "EnableMenuItem", hpopup%, itempos%, &401
```

If the menu item you are enabling or disabling is visible, you will need to update the display in order for its appearance to reflect its state (e.g. greyed-out when disabled):

```
SYS "DrawMenuBar", @hwnd%
```

You can discover the current state of a menu item using the GetMenuState function:

```
SYS "GetMenuState", hpopup%, itemid%, 0 TO state%
```

The returned variable `state%` will be a combination of one or more values, including:
1 Greyed
2 Disabled
8 Ticked
Deleting and inserting menu items

You can delete an item in a popup menu by using the DeleteMenu function:

```nim
SYS "DeleteMenu", hpopup%, itemid%, 0
```

The first parameter is the handle of the popup menu (as returned by CreatePopupMenu) and the second parameter is the menu item identifier (specified as the third parameter of AppendMenu).

An alternative method of specifying the item you want to delete is to give its position in the menu rather than its ID; this is particularly useful when the item is a sub-menu (which doesn't have an ID as such). To do this you specify &400 as the third parameter:

```nim
SYS "DeleteMenu", hpopup%, itempos%, &400
```

You can insert a new item into an existing menu using the InsertMenu function:

```nim
SYS "InsertMenu", hpopup%, position%, 0, itemid%, "Name"
SYS "InsertMenu", hpopup%, position%, 16, hsubmenu%, "Name"
```

The parameters are the same as for AppendMenu apart from position% which specifies where in the menu the new item should be inserted.
Creating a toolbar

The procedure for creating a toolbar is sufficiently complex that it has been provided in a Library; see FN_createtoolbar for details. You can, of course, cut and paste the relevant code from the library file into your own program if you prefer.
Creating a status bar

The procedure for creating a status bar is sufficiently complex that it has been provided in a Library; see FN_createstatusbar for details. You can, of course, cut and paste the relevant code from the library file into your own program if you prefer.
Using the clipboard

You may want your program to be able to access the clipboard, either to put text on the clipboard or to read text from the clipboard. The following program segment writes the string `text$` to the clipboard; if you want to include a new-line you should append a carriage-return, line-feed pair to the string as shown in the example:

```
text$ = "The five boxing wizards jump quickly"+CHR$13+CHR$10
SYS "GlobalAlloc", &2000, LEN(text$)+1 TO hdata%
SYS "GlobalLock", hdata% TO tmp%
$$tmp% = text$
SYS "GlobalUnlock", hdata%
SYS "OpenClipboard", @hwnd%
SYS "EmptyClipboard"
SYS "SetClipboardData", 1, hdata%
SYS "CloseClipboard"
```

To check whether the clipboard contains any text, you can use the following function:

```
SYS "IsClipboardFormatAvailable", 1 TO res%
```

which will set `res%` to 1 if there is text in the clipboard and to 0 otherwise. Once you have established that there is text available, you can read it with the following program segment:

```
SYS "OpenClipboard", @hwnd%
SYS "GetClipboardData", 1 TO hdata%
IF hdata% THEN
    SYS "GlobalLock", hdata% TO tmp%
    text$ = $$tmp%
    SYS "GlobalUnlock", hdata%
    REM Do something with text$
ENDIF
SYS "CloseClipboard"
```
Using dialogue boxes

The procedure for creating a custom dialogue box is sufficiently complex that it has been provided in a Library; see FN_newdialog for details. You can, of course, cut and paste the relevant code from the library file into your own program if you prefer.

Alternatively, and more easily, your BASIC program can utilise one or more of the standard dialogue boxes provided by Windows, for example the File Open (or File Save), Choose Colour, Choose Font, Browse for Folder, Print, Print Setup or Page Setup dialogue boxes. In each case you must create and initialise a data structure in memory before calling the Dialogue Box function. The examples given below illustrate only the simplest use of the dialogue boxes; there are many more options available which are beyond the scope of this manual but may be found in Microsoft documentation.

File Open and File Save

To use the File Open or File Save dialogue you must first create the data structure fs{} as follows:

```vbnet
DIM fs{lStructSize%, hwndOwner%, hInstance%, lpstrFilter%, \n    lpstrCustomFilter%, nMaxCustFilter%, nFilterIndex%, \n    nMaxFile%, lpstrFile%, \n    nMaxFileName%, lpstrFileName%, \n    flags%, nFileOffet{l&h&}, nFileExtension{l&h&}, \n    lpstrDefExt%, lpstrInitialDir%, lpstrTitle%, \n    lpTemplateName%}
DIM fp{t&(260)}
ff$ = "BMP files"+CHR$0+"*.BMP"+CHR$0
fs.lStructSize% = DIM(fs{})
fs.hwndOwner% = @hwnd%
fs.lpstrFilter% = ff$     
fs.lpstrFile% = fp{}
fs.nMaxFileName% = 260
fs.flags% = 6

The string ff$ specifies a file filter which tells the dialogue box function what file type(s) it should display by default. In the example shown the BMP file type is specified. You can specify more than one type by adding an appropriate description and extension for each, for example:
```
The string is terminated by *two* CHR$0 characters. If you do not supply a filter string, all file types will be displayed.

To list more than one file type under the same description, separate the extensions with semicolons:

```plaintext
ff$ = "Image files"+CHR$0+"*.BMP;*.GIF;*.JPG"+CHR$0+CHR$0
```

Once you have created and initialised the data structure you can call the File Open dialogue as follows:

```plaintext
SYS "GetOpenFileName", fs{} TO result%
IF result% filename$ = $$fp{}
```

If the returned value is non-zero, the selected filename can be found in memory at fp{}. If the returned value is zero, the file selection failed (for example the user selected Cancel).

You can call `GetSaveFileName` rather than `GetOpenFileName`. This will display the *Save As* title and will prompt the user if the selected file already exists.

**Choose Colour**

To use the Choose Colour dialogue ('colour picker') you must first create the data structure cc{} as follows:

```plaintext
DIM cc{1StructSize%, hwndOwner%, hInstance%,  
\    rgb{r&,g&,b&,z&}, lpCustColors%, flags%,  
\    lCustData%, lpfnHook%, lpTemplateName%}
DIM cb%(15)
cc.lStructSize% = DIM(cc{})
cc hwndOwner% = @hwnd%
cc.lpCustColors% = ^cb%(0)
```

Once you have created and initialised the data structure you can call the Choose Colour dialogue as follows:
SYS "ChooseColor", cc{} TO result%
IF result% COLOUR 1, cc.rgb.r&, cc.rgb.g&, cc.rgb.b&

If the returned value is non-zero, the selected colour can be found in memory (red at cc.rgb.r&, green at cc.rgb.g& and blue at cc.rgb.b&). In the example shown this is used to set the physical colour of logical colour 1 (see COLOUR for details). If the returned value is zero, the colour selection failed (for example the user selected Cancel).

Choose Font

To use the Choose Font dialogue you must first create the data structures lf{} and cf{} as follows:

```
DIM lf{Height%, Width%, Escapement%, Orientation%,
      Weight%, Italic&, Underline&, StrikeOut&,
      CharSet&, OutPrecision&, ClipPrecision&,
      Quality&, PitchAndFamily&, FaceName&(30)}
DIM cf{lStructSize%, hwndOwner%, hdc%, lpLogFont%,
      iPointSize%, flags%, rgbColors%, lCustData%,
      lpfnHook%, lpTemplateName%, hInstance%, lpszStyle%,
      nFontType{l&,h&}, pad{l&,h&}, nSizeMin%, nSizeMax%}
```

Once you have created and initialised the data structures you can call the Choose Font dialogue as follows:

```
SYS "ChooseFont", cf{} TO result%
```

If the returned value is non-zero, the selected font name can be found in memory at ^lf.FaceName&(0) and the font size (in decipoints) at cf.iPointSize%. If the returned value is zero, the font selection failed (for example the user selected Cancel). If you also want to know the font style (i.e. bold or italic) you can discover these as follows:

```
bold% = lf.Weight% >= 600
italic% = lf.Italic& <> 0
```
You can use this information to set the current font, size and style (see *FONT for details):

SYS "ChooseFont", cf{} TO result%
IF result% THEN
   font$ = lf.FaceName&()+","+STR$(cf.iPointSize%/10)
   style$ = ""
   IF lf.Weight% > 600 style$ += "B"
   IF lf.Italic& <> 0 style$ += "I"
   IF style$ <> "" font$ += "," + style$
   OSCLI "FONT "+font$
ENDIF

Browse for Folder

To use the Browse for Folder dialogue you must first create the data structure bi{} as follows, where title$ contains a title which can be used to prompt the user:

DIM bi{hOwner%, pidlRoot%, pszDisplayName%, 
   lpszTitle%, ulFlags%, lpfn%, lParam%, iImage%}
DIM folder{t&(260)}
titlez$ = title$+CHR$0
bi.hOwner% = @hwnd%
b.i pszDisplayName% = folder{}
b.i.lpszTitle% = !^titlez$

Once you have created and initialised the data structure you can call the Browse for Folder dialogue as follows:

SYS "SHBrowseForFolder", bi{} TO pidl%
IF pidl% SYS "SHGetPathFromIDList", pidl%, folder{}
folder$ = $$folder{}
SYS "SHGetMalloc", "malloc"
SYS !(!malloc%+20), malloc%, pidl% : REM. IMalloc::Free

If the value of pidl% is non-zero the path to the selected folder will be found (as a NUL-terminated string) at address folder%. In the above example this is converted to a standard BASIC string folder$ using the $$ indirection operator. If the user selected the Cancel button the value of pidl% will be zero.

Print dialogue
To use the Print dialogue you must first create the data structure `pd[]` as follows:

```basic
DIM pd{lStructSize%, hwndOwner%, hDevMode%, hDevNames%, hdc%, flags%, nFromPage{l&,h&}, nToPage{l&,h&}, \
    nMinPage{l&,h&}, nMaxPage{l&,h&}, nCopies{l&,h&}, hInstance%, lCustData%, lpfnPrintHook%, lpfnSetupHook%, \
    lpPrintTemplateName%, lpSetupTemplateName%, hPrintTemplate%, hSetupTemplate%}

pd.lStructSize% = DIM(pd{})
pd hwndOwner% = @hwnd%
pd nMinPage.l& = 1
pd nMaxPage.l& = 99
pd flags% = &100
```

Once you have created and initialised the data structure you can call the Print dialogue as follows:

```basic
SYS "PrintDlg", pd() TO ok%
IF ok% THEN
    SYS "DeleteDC", @prthdc%
    @prthdc% = pd hdc%
    *MARGINS 10,10,10,10
    REM. Add your conventional printing code here.
ENDIF
```

You should add your printing code where shown. You can use the values returned from the Print dialogue to control printing according to the user's choices, for example the chosen number of copies is in `pd.nCopies.l&` and the selected page range (if any) is in `pd.nFromPage.l&` and `pd.nToPage.l&`.

The *MARGINS command is necessary for *BBC BASIC for Windows* to take note of the (possibly) changed page dimensions. If you want to set the margins to values other than the default (10 mm), specify the appropriate values in the command.

**Print Setup dialogue**

Microsoft discourages the use of the Print Setup dialogue and instead recommends the Print or Page Setup dialogue in new applications. If you do want to use the Print Setup dialogue you must first create the data structure `pd[]` as follows:

```basic
DIM pd{lStructSize%, hwndOwner%, hDevMode%, hDevNames%, hdc%, flags%, nFromPage{l&,h&}, nToPage{l&,h&}, \
    nMinPage{l&,h&}, nMaxPage{l&,h&}, nCopies{l&,h&},
```
Once you have created and initialised the data structure you can call the Print Setup dialogue as follows:

```basic
SYS "PrintDlg", pd{} TO ok%
IF ok% THEN
  SYS "DeleteDC", @prthdc%
  @prthdc% = pd.hdc%
  *MARGINS 10,10,10,10
ENDIF
```

On exit from this routine, assuming the user has selected **OK** rather than **Cancel**, the current printer settings will have been changed to those chosen by the user. These may include the printer itself, any printer-specific options (e.g. print quality), the paper size and the page orientation (portrait or landscape). Subsequent output to the printer from your BBC BASIC program will use these settings.

If you need to know what the new settings are (for example the paper orientation and size) you can do that as follows:

```basic
SYS "GlobalLock", pd.hDevMode% TO dm%
  orientation% = dm%?44
  papersize% = dm%?46
  paperlength% = dm%!48 AND &FFFF
  paperwidth% = dm%!48 >>> 16
SYS "GlobalUnlock", pd.hDevMode%
```

This will set **orientation%** to 1 for portrait and to 2 for landscape and **papersize%** to 0 for a custom size or to one of the following constants for a standard size:
Other values are used for less common paper sizes. In the case of a custom size (only), `paperlength%` and `paperwidth%` are set to the dimensions in tenths of a millimetre.

The *MARGINS command is necessary for *BBC BASIC for Windows* to take note of the (possibly) changed page dimensions. If you want to set the margins to values other than the default (10 mm), specify the appropriate values in the command.

Note that this is a rare occasion when it is correct to *write* to a system variable (in this case `@prthdc%`).

**Page Setup dialogue**

To use the Page Setup dialogue you must first create and initialise the data structure `psd{}` as follows:

```plaintext
DIM psd{lStructSize%, hwndOwner%, hDevMode%, hDevNames%, \
 flags%, ptPaperSize{w%,h%}, rtMinMargin{l%,t%,r%,b%}, \
 rtMargin{l%,t%,r%,b%}, hInstance%, lCustData%, \ 
 lpfnPageSetupHook%, lpfnPagePaintHook%, \ 
 lpPageSetupTemplateName%, hPageSetupTemplate%}
psd.lStructSize% = DIM(psd({})
psd.hwndOwner% = @hwnd%
psd.flags% = 10
psd.rtMargin.l% = 1000 : REM left
psd.rtMargin.t% = 1000 : REM top
psd.rtMargin.r% = 1000 : REM right
psd.rtMargin.b% = 1000 : REM bottom
```
When you want to display the **Page Setup** dialogue (typically in response to a menu selection) execute the following code:

```autoit
SYS "PageSetupDlg", psd{} TO ok%
IF ok% THEN
    IF psd.hDevMode% THEN
        SYS "GlobalLock", psd.hDevMode% TO dm%
        SYS "ResetDC", @prthdc%, dm%
        SYS "GlobalUnlock", psd.hDevMode%
    ENDIF
    OSCLI "MARGINS "+STR$(psd.rtMargin.l%DIV100)+","+ \
    \    STR$(psd.rtMargin.b%DIV100)+","+ \
    \    STR$(psd.rtMargin.r%DIV100)+","+ \
    \    STR$(psd.rtMargin.t%DIV100)
ENDIF
```

The paper size, orientation and margins will be changed according to the user's selections.
Using system colours

For maximum authenticity, you might want your BASIC program to take account of system-wide settings, such as the default window background colour (set in Display Properties... Appearance... Item... Window). This is normally white, but can be set to any other colour by the user. The following program segment sets the background colour to the current Windows™ default, then clears the screen so that BASIC's output window is filled with that colour:

```
SYS "GetSysColor", 5 TO winbk%
COLOUR 15, winbk%, winbk% >> 8, winbk% >> 16
COLOUR 128+15
CLS
```

The parameter value 5 selects the window background colour. Other values which you might want to use with GetSysColor are given in the following table:

<table>
<thead>
<tr>
<th>Value</th>
<th>Colour returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Desktop background</td>
</tr>
<tr>
<td>4</td>
<td>Menu background</td>
</tr>
<tr>
<td>5</td>
<td>Window background</td>
</tr>
<tr>
<td>7</td>
<td>Menu text</td>
</tr>
<tr>
<td>8</td>
<td>Window text</td>
</tr>
</tbody>
</table>
Loading or saving part of a file

To load a file into memory you can use the *LOAD command, which is very fast. However, this only allows you to load an entire file. If you want to load part of a file into RAM, the only way to do it with BASIC statements is one byte at a time:

```
DIM store% size%-1
file% = OPENIN(filename$)
PTR#file% = offset%
FOR addr% = store% TO store%+size%-1
  ?addr% = BGET#file%
NEXT addr%
CLOSE #file%
```

The above program segment loads memory addresses `store%` to `store%+size%-1` inclusive with the part of the file starting at offset `offset%` from its beginning. This will work perfectly well, but is significantly slower than *LOAD. If you need to load a part of a file as quickly as possible, you can use the ReadFile API function:

```
DIM store% size%-1
file% = OPENIN(filename$)
PTR#file% = offset%
SYS "ReadFile", @hfile%(file%), store%, size%, ^temp%, 0
CLOSE #file%
```

Note the use of the system variable `@hfile%()` to discover the Windows™ file handle, and the need to allocate a 4-byte area of memory at `temp%` to hold data returned from the API function (the number of bytes loaded).

Similarly to write part of a file you can use the WriteFile function:

```
DIM store% size%-1
file% = OPENUP(filename$)
PTR#file% = offset%
SYS "WriteFile", @hfile%(file%), store%, size%, ^temp%, 0
CLOSE #file%
```

These functions are the Windows™ equivalent of the Acorn OSGBP function.

If you need to call this routine multiple times ensure that you move the DIM statement so that it will be executed only once. If you don't, you may eventually run
out of memory. Alternatively, consider using DIM LOCAL.
Aligning proportional-spaced text

Normally, centering or right-justifying text is straightforward, because the width of the text string can simply be determined from the number of characters (depending on the MODE in use, all characters are either 16, 32, or 64 BASIC graphics units wide). In MODE 8, for example, the display width of a string would be given by:

\[
\text{Width}\% = 16 \times \text{LEN}(\text{string}\$)
\]

However, if you have used *FONT to change to a proportional-spaced character font, things are not so simple. Because the characters are different widths, there is no straightforward way of calculating the total display width of a string. In that case you can use the GetTextExtentPoint32 function (see also the built-in WIDTH function for an alternative method when available):

```basic
DIM size{cx%, cy%}
*FONT "Arial", 24
SYS "GetTextExtentPoint32", @memhdc%, string$, LEN(string$), size{}
Width% = size.cx% * 2
Height% = size.cy% * 2
```

Note that the dimensions returned by GetTextExtentPoint32 are in pixels, so they are doubled to obtain values in BASIC graphics units (one pixel equals two graphics units).

Once you know the dimensions of the string you can calculate where it needs to be plotted to achieve the required alignment (e.g. centred or right-justified). Use MOVE to set the position then select VDU 5 mode before you PRINT the string to the screen.

If you want to achieve the same effect on the printer rather than the screen, it is slightly more complicated. Firstly, you must specify @prthdc% rather than @memhdc% in the GetTextExtentPoint32 function call; the dimensions returned will then be in printer units (and no doubling is required). Secondly, since there is no equivalent to VDU 5 mode for the printer, you must control the print position using the @vdu% system variable.

The following program segment prints a string centred on a particular column:
Here, \texttt{@vdu\%!-12} determines the horizontal print position, which is moved to the left by half the width of the string. To right-justify the string rather than centre it, delete the /2 at the end of the ninth line.
Displaying enhanced metafiles

Windows™ Enhanced Metafiles are files (usually with the extension .EMF) which contain pictures in a *device-independent* format. Unlike bitmaps (for example .BMP files) they do not have a fixed size in pixels. Metafiles can be displayed at different sizes, without the distortion sometimes associated with scaling a bitmap; they are also often smaller than the equivalent bitmap.

*BBC BASIC for Windows* has no built-in commands for displaying Enhanced Metafiles, but the following procedure illustrates how this may be achieved by means of calls to API functions:

```basic
DEF PROCenhmetafile(emffile$, left%, top%, right%, bottom%)
LOCAL rect{}, hemf%
DIM rect{l%, t%, r%, b%}
SYS "GetEnhMetaFile", emffile$ TO hemf%
IF hemf% = 0 ERROR 214, "File "+emffile$+" not found"
rect.l% = left%
rect.t% = top%
rect.r% = right%
rect.b% = bottom%
SYS "PlayEnhMetaFile", @memhdc%, hemf%, rect{}
SYS "InvalidateRect", @hwnd%, rect{}, 0
SYS "DeleteEnhMetaFile", hemf%
SYS "UpdateWindow", @hwnd%
ENDPROC
```

The position and size of the displayed image are determined by the values stored in the structure `rect`. The values of `left%`, `top%`, `right%` and `bottom%` must be specified in pixels, and are measured from the *top-left* corner of BASIC's output window. If necessary, these values must be converted from BASIC graphics units, taking into account that one pixel corresponds to two graphics units, and that BASIC's graphics origin is usually the *bottom-left* corner of the window.

You may also encounter an older kind of Windows™ Metafile with a .WMF extension. The program WMF2EMF, supplied with *BBC BASIC for Windows* in the EXAMPLES folder, can be used to convert a .WMF file to a .EMF file.
Using multimedia timers

For critical timing applications, Windows™ provides multimedia or interrupt timers. These are more accurate and provide a finer resolution that the normal timer services (e.g. the TIME pseudo-variable, the INKEY function, the WAIT statement and the Sleep API function). The multimedia timers are most easily accessed using the TIMERLIB library but an alternative method using assembly language code is listed below. For the purposes of illustration, the timer interrupt simply increments an integer value Count%.

```
DIM tc{PeriodMin%, PeriodMax%}
DIM P% 10
[.TimerProc
  inc dword [^Count%]
  ret 20
]
:
Interval% = 5
SYS "timeGetDevCaps", tc{}, 8 TO res%
IF res% ERROR 100, "Multimedia timer not available"
SYS "timeBeginPeriod", tc.PeriodMin%
:
SYS "timeSetEvent", Interval%, tc.PeriodMin%, TimerProc, 0, 1 TO TimerID%
IF TimerID% = 0 ERROR 100, "Could not start timer"
ON CLOSE SYS "timeKillEvent", TimerID% : QUIT
ON ERROR SYS "timeKillEvent", TimerID% : PRINT 'REPORT$ : END
:
REPEAT
  PRINT Count%
UNTIL FALSE
```

The value of Interval% determines the period of the timer interrupt, in milliseconds. In this case it is set to a period of 5 ms (a frequency of 200 Hz). The minimum period you can use is system dependent, and is returned from the timeGetDevCaps function as tc.PeriodMin% (in milliseconds). Note that interrupt timers are a scarce resource, so you must always free the timer with timeKillEvent when you have finished with it. You should also be aware that short-period timers can consume a significant portion of CPU time.

Because it is called as the result of an interrupt, there are restrictions on what you can do within the assembly-language timer routine. The details are beyond the scope of this document, but the safest thing to do is to restrict the code to performing calculations on, and modifying, data values stored in memory.
Changing the process priority

There may be rare occasions when your BASIC program has to perform a time-critical operation, during which time you don't want any other programs (or the operating system) to interrupt it. You can achieve this by setting the priority of your process to its highest possible value:

```
SYS "GetCurrentProcess" TO hprocess%
SYS "SetPriorityClass", hprocess%, &100
```

The following program segment restores the priority to its normal value:

```
SYS "GetCurrentProcess" TO hprocess%
SYS "SetPriorityClass", hprocess%, &20
```

You should use this facility with extreme care, because while the priority of your program is raised other programs may run very slowly or not at all. You may not even be able to stop or interrupt your own program! You should raise the priority only for the shortest possible period, and then return it to normal.
Using scroll bars

If your program outputs more data than will fit on the screen at any one time, you may want to incorporate a scroll bar to allow the user to view the region of interest. Although BBC BASIC for Windows has no built-in support for scroll bars, you can implement them relatively easily with API calls. Using a scroll bar involves four steps: creating the scroll bar, initialising the scroll bar, processing scroll commands and actually scrolling the display.

Creating a scroll bar

The following program segment will create and display a vertical scroll bar at the right-hand side of your program's output window:

```basic
GWL_STYLE = &H16
WS_HSCROLL = &H100000
WS_VSCROLL = &H200000
SYS "GetWindowLong", hwnd%, GWL_STYLE TO ws%
SYS "SetWindowLong", hwnd%, GWL_STYLE, ws% OR WS_VSCROLL
MODE md%
VDU 26
```

where md% is the screen mode you wish to use. Note the extra VDU 26 which causes the text and graphics clipping regions to be re-sized to take account of the presence of the scroll bar. To create a horizontal scroll bar rather than a vertical scroll bar change the WS_VSCROLL to WS_HSCROLL; to create both a vertical and a horizontal scroll bar change it to (WS_HSCROLL+WS_VSCROLL).

If you subsequently want to remove the scroll bar(s) you can do that as follows:

```basic
SB_BOTH = 3
SYS "ShowScrollBar", hwnd%, SB_BOTH, 0
VDU 26
```

To restore the scroll bar(s) thereafter you can do:

```basic
SB_BOTH = 3
SYS "ShowScrollBar", hwnd%, SB_BOTH, 1
VDU 26
```
This command restores both vertical and horizontal scroll bars. To restore just a vertical bar change the **SB_BOTH** to **SB_VERT** (1); to restore just a horizontal bar change it to **SB_HORZ** (0).

### Initialising the scroll bar

Once you have created the scroll bar you must set the *scroll range*. Normally this will correspond to the number of lines of text (in the case of a vertical scroll bar) or characters (in the case of a horizontal scroll bar) through which you want to scroll. This can be achieved for a vertical scroll bar as follows:

```plaintext
SB_VERT = 1
SYS "SetScrollRange", @hwnd%, SB_VERT, minscroll%, maxscroll%, 0
```

In most cases you would set **minscroll%** to zero and **maxscroll%** to the total number of data lines minus the number of lines displayed at any one time. So if the total number of data lines was 100 and the number displayed was 32 you would set **maxscroll%** to 68. For a horizontal scroll bar replace the **SB_VERT** with a **SB_HORZ** (0).

### Processing scroll commands

You can intercept scroll commands using the **ON MOVE** statement, as follows:

```plaintext
ON MOVE PROCscroll(@msg%,@wparam%) : RETURN
```

For a vertical scroll bar **@msg%** will have the value &115 and the low word of **@wparam%** will have one of the values 0 (scroll up one line), 1 (scroll down one line), 2 (scroll up one 'page'), 3 (scroll down one 'page') or 5 (scroll to the position indicated by the high word of **@wparam%**). A suitable **PROCscroll** would be something like the following:

```plaintext
DEF PROCscroll(msg%,wp%)
SB_VERT = 1
WM_VSCROLL = &115
IF msg% <> WM_VSCROLL ENDPIC
CASE wp% AND &FFFF OF
  WHEN 0: vscroll% -= 1
  WHEN 1: vscroll% += 1
```

Page 798
WHEN 2: vscroll% -= vpage%
WHEN 3: vscroll% += vpage%
WHEN 5: vscroll% = wp% >> 16
ENDCASE
IF vscroll% < minscroll% vscroll% = minscroll%
IF vscroll% > maxscroll% vscroll% = maxscroll%
SYS "SetScrollPos", @hwnd%, SB_VERT, vscroll%, 1
ENDPROC

Here vscroll% is the wanted scroll position and vpage% is the number of lines to scroll when the scroll bar is clicked above or below the thumb box (typically the number of displayed lines).

For a horizontal scroll bar change WM_VSCROLL to WM_HSCROLL (&114) and change the SB_VERT to SB_HORZ (0). Also replace vscroll% by hscroll% and vpage% by hpage% throughout.

If you want more control over the scroll bars, you can use the SetScrollInfo API.

**Scrolling the display**

The value of vscroll% identifies which data item should be the first to be displayed. Your program must respond to changes in this variable by scrolling the display appropriately. The method used to perform the actual scrolling will depend on the nature of the data to be displayed and how it is stored. One method, appropriate when the data is stored in a string array, can be found in the supplied example program VSCROLL.BBC (in the WINDOWS folder). Another, appropriate for scrolling over a large graphics canvas (up to 1920 x 1440 pixels), is illustrated in the example program SCROLL.BBC (in the GENERAL folder).

In the case of a vertical scroll bar you can take advantage of BASIC's VDU drivers to scroll the visible display (text viewport). To scroll the display down you should position the cursor on the top line of the text viewport and issue a VDU 11 command. Similarly to scroll the display up you should position the cursor in the bottom line and issue a VDU 10. Alternatively you can use VDU 23,7 to scroll the display in any direction.

When the screen is scrolled vertically a blank line is 'scrolled' into the top line or the bottom line respectively. To simulate scrolling through a greater amount of data than the display can show, your program must update the top or bottom line with the appropriate data item as soon as the scroll has taken place. Depending on the nature of your program, this data may either be read from a RAM buffer (e.g. an array),
read from a file or generated 'on the fly'. 
Displaying GIF and JPEG images

The built-in *DISPLAY command will only display images in Windows Bitmap (.BMP) format. If you want to display images in GIF or JPEG (.JPG) formats you can do that using the following procedure (remember that you can copy and paste this code from the help window into your program):

```plaintext
DEF PROC display(picture$, xpos%, ypos%, xsize%, ysize%)
LOCAL oleaut32%, olpp%, iid%, gpp%, hmw%, hmh%, picture%, res%
SYS "LoadLibrary", "OLEAUT32.DLL" TO oleaut32%
SYS "GetProcAddress", oleaut32%, "OleLoadPicturePath" TO olpp%
IF olpp%=0 ERROR 100, "Could not get address of OleLoadPicturePath"
DIM iid% LOCAL 15, picture% LOCAL 513
SYS "MultiByteToWideChar", 0, 0, picture$, -1, picture%, 256
iid%!0  = &7BF80980
iid%!4  = &101ABF32
iid%!8  = &AA00BB8B
iid%!12 = &AB0C3000
SYS olpp%, picture%, 0, 0, iid%, ^gpp%
IF gpp% = 0 ERROR 100, "OleLoadPicturePath failed"
SYS !(gpp%+24), gpp%, ^hmw : REM. IPicture::get_Width
SYS !(gpp%+28), gpp%, ^hmh : REM. IPicture::get_Height
SYS !(gpp%+32), gpp%, @memhdc%, xpos%, ypos%, xsize%, ysize%, 0, 
 \ hmh%, hmw%, -hmh%, 0 TO res%
IF res% ERROR 100, "IPicture::Render failed"
SYS !(gpp%+8), gpp% : REM. IPicture::Release
SYS "InvalidateRect", @hwnd%, 0, 0
SYS "UpdateWindow", @hwnd%
ENDPROC
```

This routine will also display Windows Bitmap (.BMP or .ICO) and Windows Metafile (.WMF or .EMF) images. Note that the required display position (xpos%, ypos%) must be specified in pixels, and is measured from the top-left corner of your program's output window to the top-left corner of the image. Note also that the file name must include the drive letter (e.g. C:) to distinguish it from a web URL, which can be used as an alternative means of specifying the image.
Listing the disk directory

Although you can list the contents of the current directory (folder) using the *DIR command, you have no control over the display format (other than what can be achieved by changing the font or the dimensions of the text viewport) nor can you determine file attributes other than read only.

If you need more control over the directory listing, you can use the FindFirstFile, FindNextFile and FindClose API functions. The following procedure displays the names of all the files in the current directory:

```
DEF PROC listdirectory
LOCAL dir%, sh%, res%
DIM dir% LOCAL 317
SYS "FindFirstFile", ",", dir% TO sh%
IF sh% <> -1 THEN
    REPEAT
        PRINT $$ (dir%+44)
        SYS "FindNextFile", sh%, dir% TO res%
        UNTIL res% = 0
    SYS "FindClose", sh%
ENDIF
ENDPROC
```

By adapting this routine you can display or otherwise process the disk directory in any way that you want. If you want to filter only certain files, specify an appropriate ambiguous file specification (e.g. "*.BBC") rather than "*" in the FindFirstFile line. As well as the filenames themselves, you can discover other properties of the files from the contents of the dir% structure, as follows:
dir%0 File attributes (see below)
dir%4 Time created (LS 32 bits)
dir%8 Time created (MS 32 bits)
dir%12 Time last accessed (LS 32 bits)
dir%16 Time last accessed (MS 32 bits)
dir%20 Time last modified (LS 32 bits)
dir%24 Time last modified (MS 32 bits)
dir%28 File size in bytes (MS 32 bits)
dir%32 File size in bytes (LS 32 bits)

The common file attribute values are 1 (read only), 2 (hidden), 4 (system), 16 (directory) and 32 (archive); other bits are used for specialised purposes. The values may be combined.

The time stamps are 64-bit values representing the number of 100-nanosecond intervals since 00:00 on 1st January, 1601. You can convert these values to a more conventional form using the FileTimeToSystemTime function, as follows:

```vbnet
DEF PROC filetimetosystime(dir%)
LOCAL systime%
DIM systime% LOCAL 15
SYS "FileTimeToSystemTime", dir%+20, systime%
year% = !systime% AND &FFFF
month% = systime%?2
weekday% = systime%?4
day% = systime%?6
hour% = systime%?8
minute% = systime%?10
second% = systime%?12
ENDPROC
```

This example converts the time last modified. To convert the time last accessed or time created replace the dir%+20 with dir%+12 or dir%+4 respectively. The times returned will be UTC, not your local clock time.
Opening a document file

If you want your BASIC program to open (for display or editing) a document file, you can use the ShellExecute function, as follows:

```basic
    document$ = "C:\Path\DocFile.doc"
    SYS "ShellExecute", @hwnd%, "open", document$, 0, 0, 1
```

For this to work the document type must be associated with an application (for example .doc files are usually opened with Microsoft Word™). ShellExecute can also be used to open a web page, by specifying a URL rather than a document name.

Beware that ShellExecute seems to misbehave with Windows™ 95, 98 and Me, especially when opening .doc and .pdf files. Use with care on these systems.

If you want to open a directory (folder) rather than a file you can do that very easily using Windows Explorer:

```basic
    *EXPLORER C:\Program Files\BBC BASIC for Windows
```
Discovering the Windows version

You may wish to know on which version of Windows™ (95, 98 etc.) *BBC BASIC for Windows* is running. For example, use of the mouse wheel is supported only in Windows 98 and later. You can do that using the GetVersionEx function, as follows:

```basic
DEF PROC getversion
LOCAL osvi()
DIM osvi{Size%, \ Size of structure
\ Major%, \ Major version number
\ Minor%, \ Minor Version number
\ Build%, \ Build number
\ Platform%, \ Platform ID
\ SP&(127) \ Service Pack string
\ }
osvi.Size% = 148
SYS "GetVersionEx", osvi()
MajorVersion% = osvi.Major%
MinorVersion% = osvi.Minor%
PlatformID% = osvi.Platform%
ENDPROC
```

From the **MajorVersion%**, **MinorVersion%** and **PlatformID%** values you can deduce the version of Windows, as follows:

<table>
<thead>
<tr>
<th>Windows</th>
<th>MajorVersion%</th>
<th>MinorVersion%</th>
<th>PlatformID%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows 95</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Windows 98</td>
<td>4</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Windows Me</td>
<td>4</td>
<td>90</td>
<td>1</td>
</tr>
<tr>
<td>Windows NT4</td>
<td>4</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>Windows 2000</td>
<td>5</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Windows XP</td>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Windows Vista</td>
<td>6</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Windows 7</td>
<td>6</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Windows 8</td>
<td>6</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Windows 8.1</td>
<td>6</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Windows 10</td>
<td>10</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>
If you want to find out whether the version of Windows is **32-bits** or **64-bits** you can use this function, which will return FALSE for 32-bits and TRUE for 64-bits:

```plaintext
DEF FNis64bit
LOCAL yes%
ON ERROR LOCAL = FALSE
SYS "IsWow64Process", -1, ^yes%
= yes% <> 0
```
Finding special folders

You may want to discover the location of one of the special Windows™ folders, for example Application Data or the Start Menu (you should not assume that they will always be in the same place). The following routine performs that function:

```plaintext
DEF FNspecialfolder(id%)
LOCAL ppidl%, folder%, malloc%
DIM folder% LOCAL 255
SYS "SHGetSpecialFolderLocation", @hwnd%, id%, ^ppidl%
SYS "SHGetPathFromIDList", ppidl%, folder%
SYS "SHGetMalloc", ^malloc%
SYS !(^malloc%+20), malloc%, ppidl% : REM. IMalloc::Free
= $$folder% + \\
```

The special folder returned depends on the parameter id% as follows:

0: Desktop
2: Start Menu\Programs
5: Documents
6: Favorites
7: StartUp
8: Recent files
9: Send To
11: Start Menu
20: Fonts
26: Application Data
32: Temporary Internet Files

To find the locations of the Windows directory and the System directory use the following routines:

```plaintext
DEF FNwindowsdirectory
LOCAL T%
DIM T% LOCAL 260
SYS "GetWindowsDirectory", T%, 260
```
The locations of the Temporary directory and the user's Documents folder are available in the system variables @tmp$ and @usr$ respectively.
Finding the current font name

As explained in the documentation for the *FONT command, the font selected by Windows™ may not be the same as that requested. For example if you specify the Helvetica font, and it is not installed on your system, you may find that the Arial font has been selected instead.

You can discover the font currently in use with the GetTextFace function:

```
DEF FNfont
LOCAL face%
DIM face% LOCAL 31
SYS "GetTextFace", @memhdc%, 32, face%
= $$face%
```

The above routine can be used not only to discover the actual font selected as the result of a *FONT command, but also to find the default font used following a MODE change.
Distributing the BBCWIN font

BBC BASIC for Windows is supplied with the special BBCWIN font, for use in 40-column screen modes (MODE s 1, 4, 6, 7, 9, 17 and 19). Depending on the version of Windows™ it may be selected automatically in these modes or you may need to specify it explicitly using the *FONT command:

*FONT BBCWIN

This is fine so long as you are running your program on a machine on which BBC BASIC for Windows is installed, but if you have compiled it to a stand-alone executable, for distribution to others, there is no guarantee that the BBCWIN font will be installed on the target machine.

If the BBCWIN font is not available Windows™ will select an alternative, and possibly unsuitable, font. In that case you might prefer to supply the BBCWIN.FON file (which can be found in your fonts folder) with your program to ensure the on-screen appearance is what you expect. You will then need to install the font on the target machine using the AddFontResource API function as follows:

SYS "AddFontResource", "BBCWIN.FON"
*FONT BBCWIN

This assumes that the file BBCWIN.FON is in the current directory (folder). In practice you will probably want to load it from the directory which contains your executable file. This can be achieved using the @dir$ system variable as follows:

SYS "AddFontResource", @dir$+"BBCWIN.FON"
*FONT BBCWIN

On exit from your program, or when you have finished with the font, release it as follows:

SYS "RemoveFontResource", @dir$+"BBCWIN.FON"
Checking for input focus

You may wish to know whether your program currently has the *input focus*, that is whether keyboard and mouse input is being directed to your program. For example, if your program directly checks the state of the mouse buttons with MOUSE or the state of the keyboard using INKEY (with a negative parameter), it may want to act upon these only when it has the focus. You can test that using the GetForegroundWindow function as follows:

```
SYS "GetForegroundWindow" TO hw%
IF hw% = @hwnd% THEN
   REM program has input focus
ENDIF
```
Displaying icon files

Icons are useful because, unlike standard bitmaps, they can have *transparent* backgrounds. When they are displayed the previous screen contents 'show through' the transparent region. To display an icon file use the LoadImage and DrawIconEx functions:

```plaintext
IMAGE_ICON = 1
LR_LOADFROMFILE = 16
DI_MASK = 1
DI_IMAGE = 2
SYS "LoadImage", 0, iconfile$, IMAGE_ICON, w%, h%, LR_LOADFROMFILE TO hicon%
SYS "DrawIconEx", @memhdc%, x%, y%, hicon%, w%, h%, 0, 0, DI_MASK OR DI_IMAGE
SYS "InvalidateRect", @hwnd%, 0, 0
```

Here \(w\%\) and \(h\%\) are the width and height of the icon in pixels, and \(x\%\) and \(y\%\) are the offsets in pixels from the top left-hand corner of the screen to the top left-hand corner of the icon. The maximum size of an icon is 256 x 256 pixels.

For simplicity the InvalidateRect function here updates the entire screen, not just the region where the icon was displayed. If speed is important specify an update rectangle as in Displaying enhanced metafiles.
Putting an icon in the SysTray

To display an icon in the SysTray (more correctly called the taskbar notification area) you first need to obtain a handle to the icon. If the icon image is present in a file (e.g. generated by the supplied ICONEDIT.BBC program) you can use the LoadImage function to return a handle:

```bash
IMAGE_ICON = 1
LR_LOADFROMFILE = 16
iconfile$ = @dir$+"resources\bbcmicro.ico"
SYS "LoadImage", 0, iconfile$, IMAGE_ICON, 16, 16, \
    LR_LOADFROMFILE TO hicon%
```

Next you must create a data structure in memory as follows:

```bash
NIF_MESSAGE = 1
NIF_ICON = 2
NIF_TIP = 4
WM_COMMAND = 273
DIM nid{cbSize%, hwnd%, uID%, uFlags%, uCallbackMessage%, \
    hIcon%, szTip&(63)}
    nid.cbSize% = DIM(nid{})
    nid.hwnd% = @hwnd%
    nid.uID% = 1234
    nid.uFlags% = NIF_ICON OR NIF_TIP
    nid.uCallbackMessage% = WM_COMMAND
    nid.hIcon% = hicon%
    nid.szTip&() = "Tooltip"
```

Here Tooltip is the string which is displayed if you 'hover' the mouse over the icon.

Finally to display or remove the icon you use the Shell_NotifyIcon function:

```bash
NIM_ADD = 0
SYS "Shell_NotifyIcon", NIM_ADD, nid{}
```

```bash
NIM_DELETE = 2
SYS "Shell_NotifyIcon", NIM_DELETE, nid{}
```

If you want your program to respond to clicking on the SysTray icon you should add NIF_MESSAGE to the nid.uFlags% value in the code above. This will cause messages
to be sent which can be intercepted with ON SYS, in exactly the same way as those from a menu. These messages will have an \@wparam\% value equal to the specified \nid.uID\% (1234 in the example above) and an \@lparam\% value dependent on the mouse operation, as follows:

<table>
<thead>
<tr>
<th>@lparam%</th>
<th>mouse operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>mouse move</td>
</tr>
<tr>
<td>513</td>
<td>left button down</td>
</tr>
<tr>
<td>514</td>
<td>left button up</td>
</tr>
<tr>
<td>515</td>
<td>left button double click</td>
</tr>
<tr>
<td>516</td>
<td>right button down</td>
</tr>
<tr>
<td>517</td>
<td>right button up</td>
</tr>
<tr>
<td>518</td>
<td>right button double click</td>
</tr>
<tr>
<td>519</td>
<td>middle button down</td>
</tr>
<tr>
<td>520</td>
<td>middle button up</td>
</tr>
<tr>
<td>521</td>
<td>middle button double click</td>
</tr>
</tbody>
</table>

You can expect to receive many 'mouse move' messages (@lparam% = 512) so in most circumstances you will want to ignore these in an efficient manner to avoid being 'flooded'.
Using the system registry

The registry is a repository of data maintained by Windows™, which an application can use to hold configuration or other information from one session to the next. The registry has many other uses which are outside the scope of this documentation, and the example below illustrates only the simplest way in which it can be utilised.

The registry has a hierarchical structure, not unlike a filing system. Analogous with directories or folders are keys, which identify locations within the registry. Within each key there can be a number of named values (analogous with files) each of which can contain binary data (typically a number or a string).

So for example to save a string called String$ and an integer value called Integer% in the registry under the key name Settings do the following:

```plaintext
Key$ = "Software\YourName\YourProgram\Settings"
SYS "RegCreateKeyEx", &80000001, Key$, 0, "", 0, &F003F, 0, ^K%, ^D% TO R%
IF R% = 0 THEN
    SYS "RegSetValueEx", K%, "String", 0, 1, String$, LEN(String$)+1
    SYS "RegSetValueEx", K%, "Integer", 0, 4, ^Integer%, 4
    SYS "RegCloseKey", K%
ENDIF
```

The key name and value names shown above are purely illustrative, but by adopting the style of key in the example (beginning Software\YourName\) you ensure that it is unique to your application.

To read the values back from the registry do the following:

```plaintext
DIM TempBuffer% 255
Key$ = "Software\YourName\YourProgram\Settings"
SYS "RegOpenKeyEx", &80000001, Key$, 0, &20001, ^K% TO R%
IF R% = 0 THEN
    L% = 255 : SYS "RegQueryValueEx", K%, "String", 0, ^T%, TempBuffer%, ^L% TO R%
    IF R% = 0 TempBuffer%?(L%-1) = 13 : String$ = $TempBuffer%
    L% = 4 : SYS "RegQueryValueEx", K%, "Integer", 0, ^T%, ^V%, ^L% TO R%
    IF R% = 0 Integer% = V%
SYS "RegCloseKey", K%
ENDIF
```

As always, ensure that you execute the DIM only once to avoid eventually running out of memory. Alternatively, consider using DIM LOCAL.
Creating a screen saver

To create a Windows™ screen saver using *BBC BASIC for Windows* carry out the following steps:

- Copy-and-paste the following code to the start of your program:

```basic
IF INSTR(@cmd$, "/c") THEN
    SYS "MessageBox", 0, "This screen saver has no options.", ",", 48
    QUIT
ENDIF
Preview% = INSTR(@cmd$, "/p")
IF Preview% THEN
    SYS "SetWindowLong", @hwnd%, -16, &50000000
    hparent% = VALMID$(@cmd$, Preview%+2)
    SYS "SetParent", @hwnd%, hparent%
    SYS "SetWindowLong", @hwnd%, -8, hparent%
    DIM rc{l%,t%,r%,b%}
    SYS "GetClientRect", hparent%, rc{}
    XScreen% = rc.r%
    YScreen% = rc.b%
    SYS "MoveWindow", @hwnd%, 0, 0, XScreen%, YScreen%, 0
ELSE
    SYS "SetWindowLong", @hwnd%, -16, &16000000
    SYS "GetSystemMetrics", 0 TO XScreen%
    SYS "GetSystemMetrics", 1 TO YScreen%
    SYS "SetWindowPos", @hwnd%, -1, 0, 0, XScreen%, YScreen%, 0
ENDIF
VDU 26 : COLOUR 15 : COLOUR 128 : CLS
SYS "ShowWindow", @hwnd%, 1
MOUSE MouseX%, MouseY%, MouseB%
```

- Write the rest of your program as normal, displaying whatever you want your screen saver to show, but use the values of `XScreen%` and `YScreen%` to determine the total width and height of the output in *pixels*. This will cope with both different screen resolutions and the need to create a *miniature preview* when requested by Windows. For example the following code would draw a disc just filling the height of the screen:

```basic
CIRCLE FILL XScreen%, YScreen%, YScreen%
```

- Do **not** include an ON CLOSE statement in your program as it will interfere with proper operation of the screen saver.

- Within your program's *main loop* (i.e. at a point which is executed frequently) add the following code to quit the program if the mouse is moved or a key is pressed:
IF Preview% = 0 THEN
   IF INKEY(0)<>-1 QUIT
   IF INKEY(-1) OR INKEY(-2) OR INKEY(-3) OR INKEY(-99) QUIT
   MOUSE X%,Y%,B%
   IF X%<>MouseX% OR Y%<>MouseY% OR B%<>MouseB% QUIT
ENDIF

Of course if you need to carry out any 'tidy up' processes, do them before each QUIT.

- Convert your program to an executable using the Compile command, specifying the Filename as 'Name Of Screen Saver.scr' (i.e. having the extension .scr rather than the usual .exe) and specifying the Initial window state as 'Hidden'.

- Copy the executable file to the folder on your PC containing the other screen savers (probably \Windows\System32 or \WINNT\System32). If you are not sure where this is you can search the disk for *.scr files.

Your screen saver should now appear in the list displayed on the Screen Saver tab of the Display Properties dialogue.
Drawing angled text

You may sometimes want to draw text at an angle other than horizontal. For example you might want to label the vertical axis of a graph or make a title screen more interesting. Using the procedure listed below you can draw text at any angle:

```
DEF PROCangled(font$, height%, weight%, text$, angle, X%, Y%)
LOCAL font%, oldfont%
X% = (X%+@vdu%!0)>>1
Y% = @vdu%!212-((Y%+@vdu%!4)>>1)
SYS "CreateFont", height%, 0, angle*10, angle*10, weight%, \
  0, 0, 0, 0, 0, 0, 0, font$ TO font%
SYS "SelectObject", @memhdc%, font% TO oldfont%
SYS "SetTextColor", @memhdc%, &1000000+@vdu%?70
SYS "SetBkColor", @memhdc%, &1000000+@vdu%?71
SYS "TextOut", @memhdc%, X%, Y%, text$, LEN(text$)
SYS "InvalidateRect", @hwnd%, 0, 0
SYS "SelectObject", @memhdc%, oldfont%
SYS "DeleteObject", font%
ENDPROC
```

The parameters are the name of the font, the height of the font (in pixels), the weight (400 is normal, 800 is extra bold), the text string you want to draw, the angle to the horizontal (anticlockwise, degrees) and an X,Y starting position in graphics coordinates. The text is drawn in the current text foreground and background colours, as set by COLOUR .

If you want to draw the text with a transparent (VDU 5 style) background rather than an opaque (VDU 4 style) background, replace the `SYS "TextOut"` line with the following:

```
SYS "SetBkMode", @memhdc%, 1
SYS "TextOut", @memhdc%, X%, Y%, text$, LEN(text$)
SYS "SetBkMode", @memhdc%, 2
```
Downloading files from the internet

The following code sample will download a file from the internet to your local disk, assuming you have an internet connection:

```plaintext
url$ = "http://www.bbcbasic.co.uk/index.html"
file$ = "c:\index.html"
SYS "LoadLibrary", "URLMON.DLL" TO urlmon%
SYS "GetProcAddress", urlmon%, "URLDownloadToFileA" TO UDTF%
SYS UDTF%, 0, url$, file$, 0, 0 TO fail%
IF fail% ERROR 100, "File download failed"
```

If you need to flush the cache to ensure you retrieve the latest version (e.g. it's a webcam picture) then it becomes slightly more complicated:

```plaintext
url$ = "http://www.bbcbasic.co.uk/index.html"
file$ = "c:\index.html"
SYS "LoadLibrary", "URLMON.DLL" TO urlmon%
SYS "GetProcAddress", urlmon%, "URLDownloadToFileA" TO UDTF%
SYS "LoadLibrary", "WININET.DLL" TO wininet%
SYS "GetProcAddress", wininet%, "DeleteUrlCacheEntryA" TO DUCE%
SYS DUCE%, url$
SYS UDTF%, 0, url$, file$, 0, 0 TO fail%
IF fail% ERROR 100, "File download failed"
```

The URL can specify either an HTTP (Hyper Text Transfer Protocol) document or an anonymous FTP (File Transfer Protocol) file.
Serial input/output

You can access serial communications ports from a BBC BASIC for Windows program by treating them as files. That is, you must first open a port (using OPENUP) then read and/or write data to that port using the statements and functions provided for file input/output (BGET, BPUT, INPUT#, PRINT# etc.) and finally close the port when you have finished with it (using CLOSE). There are some differences in the way these statements and functions are used when accessing a port rather than a file, which are described below.

Most PCs have either one or two serial ports, called COM1 and (if present) COM2. However it is possible to have additional ports (COM3, COM4 etc.) if required for special purposes. BBC BASIC for Windows can access these extra ports if they have a suitable Windows™ driver.

Serial ports can only be accessed by one application at a time, so if a port is currently in use by another program, BBC BASIC will not be able to open it (the OPENUP function will return the value zero).
Opening a serial port

To open a channel to a serial port you use the OPENUP function, but instead of providing a filename you should specify a string in the following form:

```
port% = OPENUP("COM1: baud=9600 parity=N data=8 stop=1")
```

The fields within the string have the following meanings:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>baud</td>
<td>Sets the baud rate (speed) of the port to the specified value. Permitted values are 75, 110, 150, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400 and 115200.</td>
</tr>
<tr>
<td>parity</td>
<td>Enables or disables parity generation and checking. Permitted values are N (signifying no parity), E (signifying even parity) and O (signifying odd parity).</td>
</tr>
<tr>
<td>data</td>
<td>Sets the number of data bits in each transmitted character. Permitted values are 7 and 8.</td>
</tr>
<tr>
<td>stop</td>
<td>Sets the number of stop bits. Permitted values are 1 and 2.</td>
</tr>
</tbody>
</table>

Optionally you can include the following parameters if supported by your PC:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>xon</td>
<td>Enables or disables XON/XOFF (software) flow-control. Permitted values are on and off.</td>
</tr>
<tr>
<td>odsr</td>
<td>Enables or disables output flow-control using the Data Set Ready (DSR) input. Permitted values are on and off.</td>
</tr>
<tr>
<td>octs</td>
<td>Enables or disables output flow-control using the Clear To Send (CTS) input. Permitted values are on and off.</td>
</tr>
<tr>
<td>dtr</td>
<td>Controls the Data Terminal Ready (DTR) output. Permitted values are on, off and hs (handshake).</td>
</tr>
<tr>
<td>rts</td>
<td>Controls the Request To Send output (RTS). Permitted values are on, off, hs (handshake) and tg (toggle).</td>
</tr>
<tr>
<td>idsr</td>
<td>Controls the DSR sensitivity. Permitted values are on and off.</td>
</tr>
</tbody>
</table>
You can also use a shorthand form of the string, where the parameter values are simply separated by commas:

```
port% = OPENUP("COM1: 9600,N,8,1")
```

If the open operation succeeds, OPENUP will return a non-zero channel number. Up to four ports may be open simultaneously, with channel numbers 1, 2, 3 and 4. If the open operation fails (for example because the port is already in use) OPENUP will return zero.
Writing to a serial port

You can write data to a serial port using any of the statements provided for writing to a file. i.e. BPUT and PRINT#. These statements will not return until the data has been sent, and depending on the baud rate this can take a significant time (at 9600 baud each character takes a little over one millisecond to transmit).
Reading from a serial port

You can read data from a serial port using any of the statements and functions provided for reading from a file, i.e. BGET, INPUT# and GET$#. These will wait until the requested amount of data has been received (one byte for BGET, multiple bytes for INPUT# and GET$#) which can, of course, be an indefinite period if no data is incoming.

To avoid your program 'hanging up' while waiting for data on a serial port, you can use the EXT# function to discover how many characters (if any) are waiting to be read. If you read only that number of characters you can guarantee that your program will never be waiting a long time for data to arrive:

```basic
port% = OPENUP("COM1: baud=9600 parity=N data=8 stop=1")
IF port% = 0 PRINT "Could not open COM1" : END
REPEAT
    chars% = EXT#port%
    IF chars% <> 0 THEN
        FOR count% = 1 TO chars%
            data% = BGET#port%
            REM. Do something with the data
        NEXT
        ENDIF
    REM. Do something useful here
UNTIL FALSE
```
Closing a serial port

Once you have finished with a serial port it should be closed, using the standard CLOSE statement. This makes the port available to another process. As usual, executing the END statement (or an untrapped error occurring) causes all open files and ports to be closed.

The EOF# function and the PTR# pseudo-variable are not meaningful in the case of a serial port, and if used will result in the Invalid channel error.
Transferring files from a BBC Micro

If you have a working *BBC Microcomputer* you can transfer files (either program files or data files) from it to your PC by means of a serial link. A software utility to make this as simple as possible is supplied with *BBC BASIC for Windows*.

**Preparation**

You will need a lead to connect the BBC Micro's RS423 serial port to the COM1 serial port on your PC. The appropriate connections between the BBC Micro's 5-pin DIN connector and the standard 25 way or 9 way D-type connector used on PCs are shown below. Note that the 5-pin DIN connector is the 360° 'domino' or 'type C' variety, not the normal 'audio' type (a suitable plug can be obtained from Electron Electronics as order code 100-456). Alternatively you may be able to purchase a ready-made lead from Brockhill Enterprises.

<table>
<thead>
<tr>
<th>BBC Micro</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIN Plug</td>
<td>D25 Socket</td>
</tr>
<tr>
<td>pin A</td>
<td>pin 2</td>
</tr>
<tr>
<td>pin B</td>
<td>pin 3</td>
</tr>
<tr>
<td>pin C</td>
<td>pin 7</td>
</tr>
<tr>
<td>pin D</td>
<td>\Connect</td>
</tr>
<tr>
<td>pin E</td>
<td>/ together</td>
</tr>
<tr>
<td>Connect /</td>
<td>pin 4</td>
</tr>
<tr>
<td>together \</td>
<td>pin 5</td>
</tr>
<tr>
<td>Connect /</td>
<td>pin 6</td>
</tr>
<tr>
<td></td>
<td>pin 8</td>
</tr>
<tr>
<td>together \</td>
<td>pin 20</td>
</tr>
</tbody>
</table>
Transferring the files

To transfer files from your BBC Micro to the PC carry out the following steps:

- Connect your PC and BBC Micro together using the lead previously described. If you have two or more serial ports on your PC, use COM1 (if for some reason you cannot use COM1, you will have to edit the BEEBXFER program accordingly).

- Insert the disk containing the file(s) you want to transfer in the BBC Micro's disk drive. If necessary, select the correct disk drive and side (*DRIVE command).

- Run the program BEEBXFER (which you will find in the TOOLS\EXAMPLES directory) and follow the on-screen instructions.

If the transfer fails, carry out the following checks:

- Ensure that the connecting cable has been wired correctly, according to the table above, and that the DIN plug is inserted with the correct orientation.

- If you are using a BBC Master computer, ensure that the correct serial data format is selected (*CONFIGURE DATA 5).

- Ensure that you are using the correct serial port on the PC (COM1).

Converting data files

Data files transferred from a BBC Micro may need to be converted before they can be successfully read by BBC BASIC for Windows (BASIC program files should not need conversion, since BBC BASIC for Windows can read Acorn-format program files).
Data files which are read entirely using BGET# or GET$# will not need conversion. However, files which are read using INPUT# or READ# must be converted using the **FCONVERT** utility supplied (it can be found in the EXAMPLES\TOOLS directory).

Data files read with a mixture of BGET# (or GET$#) and INPUT# (or READ#) are a nuisance. It may be possible to convert them, but you will get a number of 'Bad data' errors reported and you may confuse the FCONVERT utility. If this happens, you will need to write your own data conversion program and in order to do this you will need to know the structure of the data file and the way data is stored by the Acorn version of BBC BASIC.
Parallel input/output

In a similar way to the serial ports, you can open a channel to a parallel (printer) port as if it was a file. You might want to do that in order, for example, to control an unusual printer which has no Windows™ driver or if a device other than a printer is connected to the port. Note that a suitable device implementing the 'Centronics' interface protocol must be connected; you cannot use this method to output 'raw' data to the parallel port.

```plaintext
port% = OPENUP("LPT1:")
```

Unlike the serial ports, no parameters need to be specified. Normally you can only write to a parallel port, although you may be able to read from a bi-directional port in some circumstances.

Most PCs have just one parallel port, called LPT1. Parallel ports can only be accessed by one application at a time, so if a port is currently in use by another program, BBC BASIC will not be able to open it (the OPENUP function will return the value zero).

If a printer is already installed on the specified port, attempting to open the port may fail (depending on the operating system version and printer driver in use). In that case you should write to the printer using the normal methods, i.e. using VDU 2 etc.
Direct port input/output

In special circumstances you may wish to access input/output ports directly, for example to interface with and/or control external devices. Typically such devices will not have Windows™ drivers nor be supported by any operating system calls.

Input/output using InpOut32

Windows™ NT (including Windows 2000, XP, Vista, Windows 7, 8/8.1 and 10) prohibits direct low-level access to input/output ports from user programs; in so doing it provides better protection against applications interfering with each other or crashing the computer. For this reason you must use a special device driver to achieve such access; a suitable driver of this kind is **InpOut32** which can be downloaded from here.

You should copy the file **inpout32.dll** from the downloaded ZIP file (probably in the Win32 subdirectory) to your *BBC BASIC for Windows* library folder (usually C:\Program Files\BBC BASIC for Windows\LIB\). Under Windows Vista™, Windows 7™, Windows 8/8.1™ or Windows 10™ you will need administrator privileges to do that.

To use InpOut32 incorporate the following code in your BASIC program:

```basic
SYS "LoadLibrary", @lib$+"inpout32.dll" TO inpout32%
IF inpout32% = 0 ERROR 100, "Cannot load inpout32.dll"
SYS "GetProcAddress", inpout32%, "Inp32" TO `Inp32`
SYS "GetProcAddress", inpout32%, "Out32" TO `Out32`
```

Once this code has been executed, you can read from an input port using the following statement:

```basic
SYS `Inp32`, portaddress% TO portdata%```

Similarly, you can write to an output port using the following statement:

```basic
SYS `Out32`, portaddress%, portdata%```
If using **InpOut32** with a 'compiled' BASIC program you must ensure that inpout32.dll is available; this can conveniently be achieved using the following Compiler Directive:

\[
\text{REM!Embed @lib$+"inpout32.dll"}
\]

When **InpOut32** is used on Windows Vista™, Windows 7™, Windows 8/8.1™ or Windows 10™ *BBC BASIC for Windows* (or your compiled program) must be run in administrator mode. One way of doing that is to right-click on the desktop icon and select 'Run as administrator'. If you want to avoid the inconvenience of doing that, the **InpOut32** ZIP file contains the utility **InstallDriver.exe**; instructions for using it are in the file **ReadMe.txt**.

**Assembly language I/O**

Windows™ 95, 98 and Me permit direct access to the hardware ports using the processor's **in** and **out** instructions. Although *BBC BASIC for Windows* has no keywords to perform such access, you can easily achieve it by means of assembly language code. The simplest code to achieve this is as follows:

```assembly
DIM P% 6
[OPT 0
.inport in al,dx : movzx eax,al : ret
.outport out dx,al : ret : ]
```

Once this code has been executed, you can read from a port using the following program segment:

```basic
D% = address%
data% = USR(inport)
```

The static variable **D%** is first loaded with the port address (this gets copied into the processor's **edx** register), then the data is read from the port using the USR function.

Similarly, you can write to a port using the following program segment:

```basic
D% = address%
A% = data%
CALL outport
```
Here D% is loaded with the port address and A% is loaded with the data value (they get copied into the processor's edx and eax registers respectively) and the actual port write is performed using the CALL statement.

Note that, as stated previously, this method will only work with Windows™ 95, 98 and Me.
Modem control input/output

An alternative form of parallel input/output, and one which will work with all versions of Windows™ without the complication of a special driver, is the use of the modem control lines on a serial port. Each serial port (COM1, COM2 etc.) has a number of input/output signals in addition to the serial data lines; traditionally these are used to control, and receive status information from, a modem connected to the port. They may also be used for handshaking between two serial ports (for example, a receiving device might use one of these lines to cause the sending device to pause transmission).

When not used for these purposes, the modem control and status signals are available for user I/O. The only special feature to note is that the voltage levels on these lines (generally) correspond to the RS232 specification, that is they swing from a negative voltage (in the range -5 Volts to -12 Volts) to a positive voltage (+5 Volts to +12 Volts).

The signals available for this use, and the corresponding pins on the serial port connectors, are as follows:

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>D25 socket</th>
<th>D9 socket</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTS</td>
<td>output</td>
<td>pin 4</td>
<td>pin 7</td>
</tr>
<tr>
<td>CTS</td>
<td>input</td>
<td>pin 5</td>
<td>pin 8</td>
</tr>
<tr>
<td>DSR</td>
<td>input</td>
<td>pin 6</td>
<td>pin 6</td>
</tr>
<tr>
<td>CD</td>
<td>input</td>
<td>pin 8</td>
<td>pin 1</td>
</tr>
<tr>
<td>DTR</td>
<td>output</td>
<td>pin 20</td>
<td>pin 4</td>
</tr>
<tr>
<td>RI</td>
<td>input</td>
<td>pin 22</td>
<td>pin 9</td>
</tr>
<tr>
<td>GND</td>
<td>ground</td>
<td>pin 7</td>
<td>pin 5</td>
</tr>
</tbody>
</table>

Modem control output

The RTS and DTR output signals may be controlled as follows:

```
com% = OPENUP("COM1:9600,N,8,1")
IF com% = 0 ERROR 0,"Could not open serial port"
REM. Set RTS:
SYS "EscapeCommFunction", @hfile%(com%), 3
```
REM Clear RTS:
SYS "EscapeCommFunction", @hfile%(com%), 4
REM Set DTR:
SYS "EscapeCommFunction", @hfile%(com%), 5
REM Clear DTR:
SYS "EscapeCommFunction", @hfile%(com%), 6
CLOSE #com%

Modem status input

The CTS, DSR, CD and RI input signals may be tested as follows:

```
com% = OPENUP("COM1:9600,N,8,1")
IF com% = 0 ERROR 0,"Could not open serial port"
SYS "GetCommModemStatus", @hfile%(com%), ^modemstatus%
CLOSE #com%
```

The variable `modemstatus%` will contain a value corresponding to the states of the inputs as follows (the values may be combined):

<table>
<thead>
<tr>
<th>Signal</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTS</td>
<td>16</td>
</tr>
<tr>
<td>DSR</td>
<td>32</td>
</tr>
<tr>
<td>RI</td>
<td>64</td>
</tr>
<tr>
<td>CD</td>
<td>128</td>
</tr>
</tbody>
</table>
Introduction to libraries

The INSTALL statement allows you to load a library containing functions and procedures which can be called from within your program, without them appearing in the program listing. If you, or somebody else, has written a number of useful functions which you might want to call from several different programs, this provides a convenient way of packaging and distributing them.

Because the procedures and functions do not form part of your program the size of its file is reduced, and if a bug is found in one of the functions it can be corrected by updating the library without having to modify each of the programs in which it is used.

You can build up your own set of such files, but BBC BASIC for Windows is supplied with the following libraries:

- ARRAYLIB.BBC : Array and matrix functions
- WINLIB.BBC : Windows toolbars and status bars
- WINLIB2.BBC : Windows dialogue boxes
- WINLIB3.BBC : Windows trackbars and progress bars
- WINLIB4.BBC : Windows property sheets and wizards
- WINLIB5.BBC : Windows boxes and buttons
- SPRITELIB.BBC : Sprites
- FNUSING.BBC : Formatting and conversion
- MDILIB.BBC : The Multiple Document Interface
- DATELIB.BBC : Calendar functions
- D3DLIB.BBC, D3DLIBA.BBC, D3D9LIB.BBC, D3D9LIBA.BBC : Direct3D graphics
- ELLIPSE.BBC : Plotting angled ellipses
- SORTLIB.BBC : Sorting data arrays
- SOCKLIB.BBC : Socket (network) connections
- GDIPLIB.BBC : Antialiased graphics
- COMLIB.BBC : COM automation
- STRINGLIB.BBC : String manipulation
- MULTIWIN.BBC : Multiple output windows
- NOWAIT.BBC : No-wait function replacements
- CALLBACK.BBC : Callback routines
- SUBCLASS.BBC : Window subclassing
- TIMERLIB.BBC: High speed timers
- XMLLIB.BBC: Parsing XML files
- ASMLIB.BBC: Extending the assembler
- ASMLIB2.BBC: More assembler extensions
- HQSOUND.BBC: High quality sound patch

These can be found in the LIB sub-directory (folder) in the main BBC BASIC directory.
Array and matrix functions

The **ARRAYLIB** library contains a set of procedures and functions for performing arithmetic and matrix operations on 1- and 2-dimensional arrays. These include adding two matrices together, multiplying two matrices, transposing a matrix and inverting a matrix.

The library should be loaded from your program using the command:

```
INSTALL @lib$+"ARRAYLIB"
```

The functions contained are:

- PROC_add
- PROC_mul
- PROC_sum
- PROC_dot
- PROC_transpose
- PROC_invert
- FN_mod
- FN_det

In *BBC BASIC for Windows* version 3.00a and later several of these operations are incorporated within the interpreter. Using the built-in operations will be considerably faster than using the library routines; see the Array arithmetic section for details. The library routines which are not supported as built-in operations are PROC_transpose, PROC_invert and FN_det.

**PROC_add (A(), B)**

PROC_add adds a scalar value B to all the elements of 1D or 2D (numeric) array A(), and returns the result in A():

```bas
DIM N(3)
N() = 1, 2, 3, 4
PROC_add(N(), 5)
PRINT N(0),N(1),N(2),N(3)
```
PROC_mul (A(), B)

PROC_mul multiplies all the elements of 1D or 2D (numeric) array A() by the scalar value B, and returns the result in A():

```
DIM N(3)
N() = 1, 2, 3, 4
PROC_mul(N(), 2)
PRINT N(0), N(1), N(2), N(3)
```

When executed this program will print:

```
2 4 6 8
```

PROC_sum (A(), B())

PROC_sum adds 1D or 2D (numeric) arrays A() and B() together, and returns the result in A(). A() and B() must have the same dimensions.

```
DIM N(3), S(3)
N() = 1, 2, 3, 4
S() = 5, 6, 7, 8
PROC_sum(N(), S())
PRINT N(0), N(1), N(2), N(3)
```

When executed this program will print:

```
6 8 10 12
```

PROC_dot (A(), B(), C())

PROC_dot multiplies 2D matrices A() and B() together and returns the result in C(). The number of columns of A() must equal the number of rows of B(), the number of columns of C() must equal the number of columns of B() and the number of rows of C() must equal the number of rows of A().
DIM N(0,2), S(2,1), D(0,1)
N() = 1, 2, 3
S() = 4, 5, 6, 7, 8, 9
PROC_dot(N(), S(), D())
PRINT D(0,0) D(0,1)

When executed this program will print:

40    46

PROC_transpose (A(), B())

PROC_transpose transposes 2D matrix A() and returns the result in B(). The number of columns of A() must equal the number of rows of B() and the number of rows of A() must equal the number of columns of B().

DIM N(1,2), T(2,1)
N() = 1, 2, 3, 4, 5, 6
PROC_transpose(N(), T())
PRINT T(0,0) T(0,1)
PRINT T(1,0) T(1,1)
PRINT T(2,0) T(2,1)

When executed this program will print:

1    4
2    5
3    6

PROC_invert (A())

PROC_invert inverts square matrix A() and returns the result in A().

DIM M(2,2)
M() = 2,0,6,8,1,-4,0,5,7
PROC_invert(M())
PRINT M(0,0) M(0,1) M(0,2)
PRINT M(1,0) M(1,1) M(1,2)
PRINT M(2,0) M(2,1) M(2,2)

When executed this program will print:
**FN_mod (A())**

**FN_mod** returns the modulus (the square-root of the sum of the squares of all the elements) of a 1D or 2D array.

```
DIM M(2,2)
M() = 2,0,6,8,1,-4,0,5,7
PRINT FN_mod(M())
```

When executed this program will print:

13.96424

**FN_det (A())**

**FN_det** returns the determinant of a square array.

```
DIM M(2,2)
M() = 2,0,6,8,1,-4,0,5,7
PRINT FN_det(M())
```

When executed this program will print:

294
Toolbars and status bars

The WINLIB library contains a set of procedures and functions for creating and controlling toolbars and status bars. These allow you to make your BBC BASIC for Windows programs look more like genuine Windows™ programs, with a proper Graphical User Interface.

The library should be loaded from your program using the command:

```
INSTALL @lib$+"WINLIB"
```

The functions contained are:

- `FN_createstatusbar`
- `FN_createstatusbarex`
- `PROC_removestatusbar`
- `FN_createtoolbar`
- `FN_createtoolbarex`
- `PROC_removetoolbar`
- `FN_custombutton`
- `PROC_addtooltips`

The WINLIBU library contains an identical set of functions except that those receiving a text string assume it to be in Unicode (UTF-8) format, allowing non-ANSI (e.g. foreign language) characters to be incorporated.

**FN_createstatusbar (text$)**

This function creates a status bar, and returns its window handle. A single string parameter must be supplied, which contains the text (if any) to be displayed by default in the status bar. The returned value can be used for manipulating the status bar using Windows™ API functions (see below).

The following program segment creates a status bar containing the string "Press F1 for Help":

```
hstat% = FN_createstatusbar("Press F1 for help")
```
Note that BASIC's output window is **not** automatically made smaller to accommodate the status bar. You should use the VDU 24 and/or VDU 28 commands to reduce the size of the graphics and/or text windows to suit.

If you want the status bar to resize itself automatically when the size of the main window changes (e.g. as the result of the user dragging a corner) you should _forward_ the size-change message as follows:

```plaintext
ON MOVE SYS "PostMessage", hstat%, @msg%, @wparam%, @lparam% : RETURN
```

You should remove the status bar by calling PROC_removestatusbar before your program exits or returns to immediate mode. It is good practice to trap errors (using ON ERROR ) and to remove the status bar if an unexpected error occurs.

Once you have created a basic status bar, you can use Windows™ API functions to divide it into a number of _parts_ and write different text in each part. The following program segment illustrates how to split the status bar into three parts; `hstat%` is the value returned from the FN_createstatusbar function.

```plaintext
SB_SETTEXT = 1025
SB_SETPARTS = 1028
nparts% = 3
DIM edges%(nparts%-1)
FOR N% = 0 TO nparts%-1
    READ edges%(N%)
NEXT N%
SYS "SendMessage", hstat%, SB_SETPARTS, nparts%, ^edges%(0)
FOR N% = 0 TO nparts%-1
    READ text$
    SYS "SendMessage", hstat%, SB_SETTEXT, N%, CHR$9+text$
NEXT N%
DATA 440, 540, 640
DATA Part 1, Part 2, Part 3
```

The first DATA statement determines the width of each _part_, by specifying the position of its right-hand edge (in pixels). The second DATA statement determines the text which will appear in each part; the `CHR$9` character causes the text to be displayed centrally.

**FN_createstatusbarestex** (text$, style%)

This function is similar to FN_createstatusbar except that it takes an additional
style% parameter. This parameter may be set to SBARS_SIZEGRIP (256) to cause a 'size grip' to be displayed at the extreme right-hand end of the status bar. This is appropriate when the parent window can be resized by the user (by dragging a side or corner).

PROC_removeStatusBar

This procedure removes a status bar previously created with FN_createstatusbar or FN_createstatusbarex. You should always remove the status bar before your program exits or returns to immediate mode.

FN_create_toolbar (nbutts%, button%(), buttid%())

This function creates a toolbar, and returns its window handle. Three parameters must be supplied: the number of buttons in the toolbar, an integer array of button types and an integer array of button identifiers. The returned value can be used for manipulating the toolbar using Windows™ API functions (see below). To create a floating toolbar see the FN_createfloating_toolbar function in the WINLIB3 library.

The following program segment creates a toolbar containing three buttons: a cut button, a copy button and a paste button:

```pascal
  nbutts% = 3
  DIM button%(nbutts%-1), buttid%(nbutts%-1)
  button%() = 0, 1, 2
  buttid%() = 100, 101, 102
  htoolbar = FN_create_toolbar(nbutts%, button%(), buttid%())
```

The button% array contains the button types and the buttid% array contains the button identifiers (which are used to identify the buttons, and to determine which button has been clicked). If the button identifier is zero this signifies a separator rather than a button; the corresponding entry in the button% array contains a value (normally zero) by which the width of the separator is increased, in pixels.

In version 1.7 and later of the WINLIB library you can specify a negative identifier value. This creates an auto toggle button which alternates between the 'pressed' and 'unpressed' states when you click on it. In this case the actual identifier value allocated to the button is minus the value specified in the buttid% array.

The available button types are as follows:
Note that BASIC's output window is **not** automatically made smaller to accommodate the toolbar. You should use the VDU 24 and/or VDU 28 commands to reduce the size of the graphics and/or text windows to suit.

If you want the toolbar to resize itself automatically when the size of the main window changes (e.g. as the result of the user dragging a corner) you should *forward* the size-change message as follows:

```
ON MOVE SYS "PostMessage",htool%,@msg%,@wparam%,@lparam% : RETURN
```

You should remove the toolbar by calling PROC_removetoolbar before your program exits or returns to immediate mode. It is good practice to trap errors (using ON ERROR) and to remove the toolbar if an unexpected error occurs.

Once you have created a toolbar, clicking on the buttons will cause messages to be sent to your program which can be detected with ON SYS, in exactly the same way as a selection from a menu bar. The `@wparam%` value will be the button identifier.
you chose. Alternatively you can automate the process and avoid the need to use ON SYS in your code by allocating the identifier using FN_setproc in WINLIB5.

You can use Windows™ API functions to determine the appearance of the buttons. The following program segment illustrates how to cause a button to be disabled (greyed out); htool% is the value returned from the FN_createtoolbar function:

```plaintext
TB_SETSTATE = 1041
state% = 0
SYS "SendMessage", htool%, TB_SETSTATE, buttid%, state%
```

Some of the possible button states are:

- 0  Disabled
- 4  Enabled
- 6  Pressed
- 8  Hidden

You can discover the current state of a button as follows:

```plaintext
TB_GETSTATE = 1042
SYS "SendMessage", htool%, TB_GETSTATE, buttid%, 0 TO state%
```

In the case of an auto toggle button which has been toggled to the 'pressed' state, the returned value is one greater than those shown. So 5 signifies that it has been toggled to the 'pressed' state and 7 that it is also being pressed by the user.

**FN_createtoolbarex**

```plaintext
(nbutts%,button%(),buttid%(),resid%,style%)
```

This function is similar to FN_createtoolbar except that it takes additional resid% and style% parameters. resid% should be set to IDB_STD_SMALL_COLOR (0) for small toolbar buttons and IDB_STD_LARGE_COLOR (1) for large toolbar buttons. style% can be set to one or more of the toolbar styles, such as TBSTYLE_WRAPABLE (512) to cause the toolbar to wrap onto multiple lines if its width is insufficient.

**PROC_removetoolbar**
This procedure removes a toolbar previously created with FN_createtoolbar or FN_createtoolbarex. You should always remove the toolbar before your program exits or returns to immediate mode.

**FN_custombutton (htool%,bmpfile$,buttid%)**

This function allows you to customise a button image to suit your particular application. Three parameters must be supplied: the handle of the toolbar (as returned from FN_createtoolbar), the name of a Windows Bitmap file containing the required button image (it is automatically scaled to fit the button) and the identifier of the button you wish to change.

The function returns TRUE if the button image was successfully changed, and FALSE otherwise (most likely because the specified file could not be read or has an invalid format). You can customise as many buttons as you like by calling this function multiple times with different values of buttid%; alternatively you can customise multiple buttons from a single bitmap file using the FN_custombuttons function to be found in the WINLIB3 library.

The following program segment illustrates how to customise a button image:

```plaintext
ok% = FN_custombutton(htool%, "\pictures\bmp\owl.bmp", 102)
```

When initially creating the toolbar set the button 'type' of any custom buttons to 15 (no image). Also ensure that the background colour of your bitmap file is R=192, G=192, B=192 (&C0C0C0). These measures will ensure that your custom images are displayed correctly.

**PROC_add tooltips (htool%,nbutts%,buttip$(),buttid%)**

This procedure allows you to add tool tips to your toolbar. A tool tip is a text string which appears in a small yellow box when you 'hover' the mouse over the toolbar button for a short while. Adding tool tips to your toolbar gives your program an even more authentic Windows™ interface.

Four parameters must be supplied: the handle of the toolbar (as returned from FN_createtoolbar), the number of buttons for which you want to provide tips, an array of strings containing the tips to be displayed and an integer array of identifier
values for the appropriate buttons.

The following program segment illustrates how to add tooltips to the toolbar created in the earlier example:

```vbscript
nbutts% = 3
DIM buttip$(nbutts%-1), buttid%(nbutts%-1)
buttip$(()) = "Cut", "Copy", "Paste"
buttid%() = 100, 101, 102
PROC_addtooltips(htool%, nbutts%, buttip$(()), buttid%())
```

The `buttid%` array may often be the same as that passed to FN_createtoolbar.
Dialogue boxes

The **WINLIB2** library contains a set of procedures and functions for creating and controlling custom dialogue boxes. These allow you to adopt the preferred Windows™ method of requesting input from the user.

The library should be loaded from your program using the command:

```
INSTALL @lib$+%WINLIB2%
```

The functions contained are:

- FN_newdialog
- PROC_pushbutton
- PROC_checkbox
- PROC_radiobutton
- PROC_groupbox
- PROC_editbox
- PROC_static
- PROC_listbox
- PROC_combobox
- PROC_dlgitem
- PROC_dlgctrl
- PROC_showdialog
- PROC_closedialog

The **WINLIB2B** library contains an identical set of functions but additionally forwards to your program any **WM_HELP** and **WM_NOTIFY** messages generated by the dialogue box controls. These messages can be intercepted using *SYS 1 to allow your program to provide, for example, *context-sensitive help* for your dialogue box. WINLIB2B also suppresses the automatic closing of the dialogue box when you click on its close button, or press Escape.

The **WINLIB2U** library contains an identical set of functions to WINLIB2B except that those receiving a text string assume it to be in **Unicode** (UTF-8) format, allowing non-ANSI (e.g. foreign language) characters to be incorporated.

The supplied example program DLGEDIT.BBC can simplify the creation of...
dialogue boxes by providing a 'visual' editor and generating some of the required BBC BASIC code automatically.

**FN_newdialog**

Before you can use a custom dialogue box, you must first define its position, size and title. This should be done only once, typically in an initialisation routine, for each dialogue box your program contains (the dialogue box can subsequently be displayed as many times as you like):

```basic
dlg% = FN_newdialog(title$, x%, y%, cx%, cy%, font%, size%)
```

The string `title$` specifies the title of the dialogue box, which is displayed in its title bar. The values `x%` and `y%` specify the initial position of the dialogue box, with respect to the top left corner of your program's window. The values `cx%` and `cy%` specify the width and height, respectively, of the dialogue box.

The value `font%` specifies the size, in points, of all the text strings in the dialogue box, and it also determines the size of the units used to specify the position and size of the box. Therefore, if you change the size of the text all the other dimensions of the dialogue box are scaled to suit. A common value to use is 8.

The value `size%` specifies the number of bytes of memory to allocate for the dialogue box template. As a rule of thumb this can be set to approximately the number of individual items within the dialogue box multiplied by 50; if the value is too small you will receive the error message *No room for dialogue template* when the items are created. There is no harm, apart from wasted memory, in making it too big.

The returned value (`dlg%` in this case) identifies the dialogue box, and must be stored for use in the subsequent procedures.

Normally a dialogue box has a title bar. This can be eliminated by modifying the style after the call to **FN_newdialog** as follows:

```basic
dlg% = FN_newdialog("", x%, y%, cx%, cy%, font%, size%)
WS_BORDER = &800000
dlg%!16 AND= NOT WS_BORDER
```
Note, however, that if you do this the user won't be able to move the dialogue box, so you may need to set its position more carefully.

Make sure you only call **FN_newdialog once** (for each dialogue box); if you need to re-open a dialogue box that has already been opened, simply call **PROC_showdialog**. Preferably place all your calls to **FN_newdialog** in an initialisation routine that is only executed once.

**PROC_pushbutton**

Once you have created the basic dialogue box, you must define its contents. The common items (pushbuttons, edit boxes etc.) each have their own procedure. Again, the contents of the dialogue box must be defined just **once**, during your program's initialisation phase.

All dialogue boxes should have at least one *pushbutton*, labelled **OK**, which enables the user to confirm that his input is complete:

```
PROC_pushbutton(dlg%,text$,id%,x%,y%,cx%,cy%,style%)
```

The value **dlg%** identifies the dialogue box, and is the value returned from **FN_newdialog**. The string **text$** specifies the text to appear inside the button (e.g. "OK"), the values **x%** and **y%** specify the position of the button within the dialogue box and the values **cx%** and **cy%** specify the size of the button (in dialogue box units).

The value **id%** is a unique identifier of the pushbutton; all the items within a particular dialogue box must have different values. You can choose any value (within reason) but the values 1 and 2 are reserved for the **OK** and **Cancel** buttons respectively so should only be used for that purpose. Alternatively you can set this parameter to a value returned from **FN_setproc** (in WINLIB5 or WINLIB5A) which will cause the specified procedure to be called when the button is clicked.

The value **style%** can be zero, but other values allow you to modify the appearance or behaviour of the pushbutton. Setting it to 1 (**BS_DEFPUSHBUTTON**) causes the pushbutton to be the *default* button: it is displayed with a thicker border and pressing the Enter key has the same effect as pressing that button. There should only be one button in each dialogue box with this style; typically you would use it for the **OK** button. Setting style% to &20000 (**WS_GROUP**) causes the button to be the first
item in a new group; this affects navigation of the dialogue box using the cursor keys. Setting style% to &80 (BS_BITMAP) indicates that the button will display a bitmap image (see Initialising the contents of a dialogue box); in this case text$ should be empty. The values can be combined.

**PROC_checkbox**

A check box is a small square which can either be *checked* (contain a tick mark) or not. When the user clicks on the box, it (usually) toggles between being checked and being unchecked. A check box is used to select one of two states (e.g. off or on).

\[
\text{PROC_checkbox}(\text{dlg}, \text{text}, \text{id}, \text{x}, \text{y}, \text{cx}, \text{cy}, \text{style})
\]

The value **dlg** identifies the dialogue box, and is the value returned from FN_newdialog. The string **text** specifies the text to appear alongside the check box, the values **x** and **y** specify the position of the check box within the dialogue box and the values **cx** and **cy** specify the size of the check box and its associated text (in dialogue box units). The value **id** is a unique identifier for the check box.

The value **style** can be zero, but other values allow you to modify the appearance or behaviour of the check box. Setting it to &20 (BS_LEFTTEXT) causes the associated text to appear to the left of the check box rather than to the right. Setting it to &20000 (WS_GROUP) causes the check box to be the first item in a new group; this affects navigation of the dialogue box using the cursor keys. The values can be combined.

**PROC_radiobutton**

A radio button is a small circle which can either be *checked* (contain a central spot) or not. Radio buttons are used in groups of two or more, where only one of the buttons is checked at any one time. When the user clicks one of the buttons it becomes checked, and all the other buttons in the group become unchecked. A radio button is used to select one of two or more states.

\[
\text{PROC_radiobutton}(\text{dlg}, \text{text}, \text{id}, \text{x}, \text{y}, \text{cx}, \text{cy}, \text{style})
\]

The value **dlg** identifies the dialogue box, and is the value returned from
FN_newdialog. The string text$ specifies the text to appear alongside the radio button, the values x% and y% specify the position of the radio button within the dialogue box and the values cx% and cy% specify the size of the radio button and its associated text (in dialogue box units). The value id% is a unique identifier for the radio button.

The value style% can be zero, but other values allow you to modify the appearance or behaviour of the radio button. Setting it to &20 (BS_LEFTTEXT) causes the associated text to appear to the left of the radio button rather than to the right. Setting it to &20000 (WS_GROUP) causes the radio button to be the first item in a new group; this has special significance for radio buttons, since only one radio button in a group can be checked at any one time. The values can be combined.

**PROC_groupbox**

A group box is a rectangle which is used to enclose a number of items within the dialogue box, thus emphasising that they are grouped together.

PROC_groupbox(dlg%,text$,id%,x%,y%,cx%,cy%,style%)

The value dlg% identifies the dialogue box, and is the value returned from FN_newdialog. The string text$ specifies the text to appear in the top edge of the group box, the values x% and y% specify the position of the group box within the dialogue box and the values cx% and cy% specify the size of the group box (in dialogue box units). The value id% is a unique identifier for the group box.

The value style% may be zero, but will often have the value &20000 (WS_GROUP) signifying that the group box is the first item in a new group.

**PROC_editbox**

An edit box is a rectangular field into which the user can type textual or numeric input.

PROC_editbox(dlg%,text$,id%,x%,y%,cx%,cy%,style%)

The value dlg% identifies the dialogue box, and is the value returned from FN_newdialog. The string text$ specifies the initial text (if any) to appear in the
edit box, the values \texttt{x\%} and \texttt{y\%} specify the position of the edit box within the
dialogue box and the values \texttt{cx\%} and \texttt{cy\%} specify the size of the edit box (in
dialogue box units). The value \texttt{id\%} is a unique identifier for the edit box.

The value \texttt{style\%} can be zero, but other values allow you to modify the appearance
or behaviour of the edit box. Setting it to \&80 (ES.AUTOHSCROLL) causes the
contents of the edit box to scroll horizontally, if necessary. Setting it to \&2000
(ES.NUMBER) causes the edit box to accept only numeric input. Setting it to
\&20000 (WS.GROUP) causes the edit box to be the first item in a new \textit{group}. Setting it to \&1004 (ES.WANTRETURN + ES.MULTILINE), along with an
appropriate vertical size, creates a \textit{multi-line} edit box. The values can be combined.

\textbf{PROC\_static}

A static item is a rectangular area containing (usually) a text string or an image. This
may be used to label another item or be simply informative.

\begin{verbatim}
PROC_static(dlg%,text$,id%,x%,y%,cx%,cy%,style%)
\end{verbatim}

The value \texttt{dlg\%} identifies the dialogue box, and is the value returned from
\texttt{FN.newdialog}. The string \texttt{text$} specifies the required text (if any), the values \texttt{x\%}
and \texttt{y\%} specify the position of the rectangle within the dialogue box and the values
\texttt{cx\%} and \texttt{cy\%} specify the size of the rectangle (in dialogue box units). The value
\texttt{id\%} is a unique identifier for the static item.

The value \texttt{style\%} may be zero, but other values allow you to modify the appearance
of the static item. By default text is left-justified within the rectangle but setting
\texttt{style\%} to 1 (SS.CENTER) causes the text to be centred within the rectangle and
setting it to 2 (SS.RIGHT) causes the text to be right-justified within the rectangle.
Setting \texttt{style\%} to \&E (SS_BITMAP) indicates that the item will contain a bitmap
image to be loaded later (see Initialising the contents of a dialogue box); in this case
\texttt{text$} should be empty.

\textbf{PROC\_listbox}

A list box displays a list of two or more items from which the user can select one.

\begin{verbatim}
PROC_listbox(dlg%,"",id%,x%,y%,cx%,cy%,style%)
\end{verbatim}
The value `dlg%` identifies the dialogue box, and is the value returned from `FN_newdialog`. The text string is unused and should be set to an empty string, the values `x%` and `y%` specify the position of the list box within the dialogue box and the values `cx%` and `cy%` specify the size of the list box (in dialogue box units). The value `id%` is a unique identifier for the list box.

The value `style%` can be zero, but other values allow you to modify the appearance or behaviour of the list box. For example setting it to `&20000` (WS_GROUP) causes the list box to be the first item in a new `group`. To disable automatic sorting of the listbox contents subtract 2 (LBS_SORT) from the `style%` value you would otherwise have used.

The items to be displayed in the list box must be written as a separate exercise once the dialogue box has been displayed. See Initialising the contents of a dialogue box for details. If the height of the list box is insufficient for the number of items to be displayed, a vertical scroll bar is automatically generated. Adding `&100200` (WS_HSCROLL + LBS_MULTICOLUMN) to `style%` instead results in the list box having a horizontal scroll bar, and the items are presented in columns.

**PROC_combobox**

A combo box consists of a list and a selection field. The list presents the options a user can select and the selection field displays the current selection.

```
PROC_combobox(dlg%,"",id%,x%,y%,cx%,cy%,style%)
```

The value `dlg%` identifies the dialogue box, and is the value returned from `FN_newdialog`. The text string is unused and should be set to an empty string, the values `x%` and `y%` specify the position of the combo box within the dialogue box and the values `cx%` and `cy%` specify the size of the combo box (in dialogue box units). The value `id%` is a unique identifier for the combo box.

The value `style%` can be zero, but other values allow you to modify the appearance or behaviour of the combo box. Setting it to 3 (CBS_DROPDOWNLIST) creates a `drop down list`, where the list of items from which the selection can be made is only displayed when the user clicks on the arrow button (note particularly that in this case `cy%` is the `dropped down` height of the box). Setting it to `&100` (CBS_SORT) causes the items in the list to be sorted into alphabetical order. Setting it to `&20000`
(WS_GROUP) causes the combo box to be the first item in a new group. The values can be combined.

The items to be displayed in the combo box list must be written as a separate exercise once the dialogue box has been displayed. See Initialising the contents of a dialogue box for details. In the case of a drop-down list the height of the combo box should be made sufficient for the list when it is displayed.

**PROC_dlgitem**

PROC_dlgitem can be used to create items other than those for which dedicated procedures are provided.

```
PROC_dlgitem(dlg%,text$,id%,x%,y%,cx%,cy%,style%,class%)
```

The value `dlg%` identifies the dialogue box, and is the value returned from FN_newdialog. The values `x%` and `y%` specify the position of the item within the dialogue box and the values `cx%` and `cy%` specify the size of the item (in dialogue box units). The values of `text$`, `style%` and `class%` depend on the type of item. The value `id%` is a unique identifier for the item.

**PROC_dlgctrl**

PROC_dlgctrl is similar to PROC_dlgitem except that a class name is specified rather than a numeric value:

```
PROC_dlgctrl(dlg%,text$,id%,x%,y%,cx%,cy%,style%,class$)
```

This allows you to embed in your dialogue box standard Windows™ controls for which dedicated procedures are not provided. Examples of such controls are up-down controls, trackbars and progress bars:

**Adding an up-down control**

You can add an *up-down control* (a pair of arrows used for incrementing and decrementing a numeric value in an edit box) using the following program segment:

```
PROC_dlgctrl(dlg%,"",id%,0,0,0,0,&50000096,"msctls_updown32")
```
This item must immediately follow a numeric edit box in which the value controlled by the up and down arrows appears. The up-down control automatically positions itself at the right-hand end of the edit box (the position values are unused and are set to 0,0).

To set the range of the up-down control (which must be done after the PROC_showdialog):

```
UDM_SETRANGE = 1125
SYS "SendDlgItemMessage", !dlg%, id%, UDM_SETRANGE, 0, (min% << 16) + max%
```

**Adding a trackbar**

You can add a *trackbar* to a dialogue box using the following program segment:

```
PROC_dlgctrl(dlg%,"",id%,x%,y%,cx%,cy%,&50000000,"msctls_trackbar32")
```

The above example creates a horizontal trackbar with no tick-marks. To alter the style of the trackbar add one or more of the following values to the &50000000:

<table>
<thead>
<tr>
<th>Style</th>
<th>Name</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TBS_AUTOTICKS</td>
<td>Show tick marks</td>
</tr>
<tr>
<td>2</td>
<td>TBS VERT</td>
<td>Vertical trackbar</td>
</tr>
<tr>
<td>4</td>
<td>TBS_LEFT</td>
<td>Tick marks to top or left (default is bottom or right)</td>
</tr>
<tr>
<td>8</td>
<td>TBS BOTH</td>
<td>Tick marks on both sides</td>
</tr>
</tbody>
</table>

To set the range of the trackbar (which must be done after the PROC_showdialog):

```
TBM_SETRANGE = 1030
SYS "SendDlgItemMessage", !dlg%, id%, TBM_SETRANGE, 1, (max% << 16) + min%
```

To set the current position of the trackbar:

```
TBM_SETPOS = 1029
SYS "SendDlgItemMessage", !dlg%, id%, TBM_SETPOS, 1, position%
```
To read the current position of the trackbar:

```
TBM_GETPOS = 1024
SYS "SendDlgItemMessage", !dlg%, id%, TBM_GETPOS, 0, 0 TO position%
```

**Adding a progress bar**

You can add a *progress bar* to a dialogue box using the following program segment:

```
PROC_dlgctrl(dlg","",id%,x%,y%,cx%,cy%,&50000000,"msctls_progress32")
```

The above example creates a horizontal progress bar. To alter the style of the progress bar add one or more of the following values to the \&50000000:

<table>
<thead>
<tr>
<th>Style</th>
<th>Name</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PBS_SMOOTH</td>
<td>Smooth (rather than segmented) progress bar</td>
</tr>
<tr>
<td>4</td>
<td>PBS_VERTICAL</td>
<td>Vertical progress bar</td>
</tr>
</tbody>
</table>

To set the range of the progress bar (which must be done after the PROC_showdialog):

```
PBM_SETRANGE = 1025
SYS "SendDlgItemMessage", !dlg%, id%, PBM_SETRANGE, 0, (max% << 16) + min%
```

To set the current position of the progress bar:

```
PBM_SETPOS = 1026
SYS "SendDlgItemMessage", !dlg%, id%, PBM_SETPOS, position%, 0
```

To 'step' the progress bar:

```
PBM_STEPIT = 1029
SYS "SendDlgItemMessage", !dlg%, id%, PBM_STEPIT, 0, 0
```

**PROC_showdialog**
Once the position, size and contents of the dialogue box have been defined, it may be displayed on the screen. Whilst the creation of the dialogue box and the definition of its contents should be done just once, the box may subsequently be displayed as many times as you like:

PROC_showdialog(dlg%)

The value $dlg%$ identifies the dialogue box, and is the value returned from FN_newdialog. Once the dialogue box has been displayed, you can send messages to it to affect its contents, and request information about its current contents.

**Initialising the contents of a dialogue box**

In many cases the initial contents of the dialogue box are defined when the items within it are created. For example, the initial contents of an edit box can be specified in the PROC_editbox procedure call. However you may want to change the contents at other times, and in the case of list boxes and combo boxes, which contain multiple items, you must use other methods to initialise the contents. *Note that this initialisation must take place after the call to PROC_showdialog.*

To change the text associated with a dialogue box item (which may for example be the contents of an edit box or a label for a check box) you can use the SetDlgItemText API call:

```
SYS "SetDlgItemText", !dlg%, id%, text$
```

The value $!dlg%$ (note the exclamation mark) is the handle of the dialogue box, which is contained in memory at the address returned from FN_newdialog. The value $id%$ is the identifier for the item in question and $text\$ is the new text string to be associated with the item.

If the item is an edit box used for numeric entry, the value displayed in the box can be set with SetDlgItemInt:

```
SYS "SetDlgItemInt", !dlg%, id%, value%, signed%
```

The value $!dlg%$ is the handle of the dialogue box, $id%$ is the identifier for the item in question, $value%$ is the new value to display and $signed%$ determines whether
the value should be interpreted as signed (1) or unsigned (0).

If the value is controlled by an up-down control you can set the allowed range as follows:

```
UDM_SETRANGE = 1125
SYS "SendDlgItemMessage", !dlg%, id%, UDM_SETRANGE, 0, (min% << 16) + max%
```

where \text{id}\% is the identifier for the up-down control, \text{min}\% is the lowest value permitted and \text{max}\% is the highest value permitted.

To load a bitmap image into a static item or a pushbutton you should use the following program segment. In the case of a pushbutton change \text{STM_SETIMAGE} (370) to \text{BM_SETIMAGE} (247):

```
LR_LOADFROMFILE = 16
STM_SETIMAGE = 370
SYS "LoadImage", 0, bmpfile$, 0, cx%, cy%, LR_LOADFROMFILE TO hbitmap%
SYS "SendDlgItemMessage", !dlg%, id%, STM_SETIMAGE, 0, hbitmap%
```

The value \text{bmpfile}\$ is the name of a Windows™ bitmap file containing the image, \text{cx}\% and \text{cy}\% are the dimensions of the image in pixels, \text{!dlg}\% is the \textit{handle} of the dialogue box and \text{id}\% is the identifier for the static item or pushbutton in question. Once you have finished with the dialogue box (but not before) delete the bitmap handle as follows:

```
SYS "DeleteObject", hbitmap%
```

To enter the list of strings into a list box you should do the following:

```
LB_ADDSTRING = 384
SYS "SendDlgItemMessage", !dlg%, id%, LB_ADDSTRING, 0, "Listbox item 0"
SYS "SendDlgItemMessage", !dlg%, id%, LB_ADDSTRING, 0, "Listbox item 1"
SYS "SendDlgItemMessage", !dlg%, id%, LB_ADDSTRING, 0, "Listbox item 2"
etc.
```

where the value \text{id}\% is the identifier for the list box. The list box will (by default) sort the strings into alphabetical order, so the order in which they are sent is not important. To disable sorting subtract 2 (LBS_SORT) from the \textit{style}\% value you
would otherwise have used (see PROC_listbox).

To empty a list box of its contents do:

```
LB_RESETCONTENT = 388
SYS "SendDlgItemMessage", !dlg%, id%, LB_RESETCONTENT, 0, 0
```

To enter the list of strings into a combo box you should do the following:

```
CB_ADDSTRING = 323
SYS "SendDlgItemMessage", !dlg%, id%, CB_ADDSTRING, 0, "Combobox item 0"
SYS "SendDlgItemMessage", !dlg%, id%, CB_ADDSTRING, 0, "Combobox item 1"
SYS "SendDlgItemMessage", !dlg%, id%, CB_ADDSTRING, 0, "Combobox item 2"
```

where the value `id%` is the identifier for the combo box. In this case the items are not sorted (by default), so they must be sent in the order in which they should appear. The initial selection from the list should be made as follows:

```
CB_SETCURSEL = 334
SYS "SendDlgItemMessage", !dlg%, id%, CB_SETCURSEL, index%, 0
```

The value `index%` determines which of the items is initially selected (starting at 0).

To empty a combo box of its contents do:

```
CB_RESETCONTENT = 331
SYS "SendDlgItemMessage", !dlg%, id%, CB_RESETCONTENT, 0, 0
```

To initialise the state of a set of radio buttons, you can use the CheckRadioButton API call:

```
SYS "CheckRadioButton", !dlg%, first%, last%, id
```

Here `first%` is the identifier of the first radio button in the group, `last%` is the identifier of the last radio button in the group and `id%` is the identifier of the button you want to be checked.

To initialise the state of a check box, you can use the CheckDlgButton API call:
SYS "CheckDlgButton", !dlg%, id%, state%

Here \texttt{id\%} is the identifier of the button you want to affect and \texttt{state\%} is the state you wish it to have: 0 signifies unchecked and 1 signifies checked.

**Disabling and enabling dialogue box items**

You may wish to disable one or more items in a dialogue box, for example you may want to disable a button if it is not appropriate to click it in the current circumstances. An item which is disabled is shown in grey.

To disable an item you can use the EnableWindow API call:

```
SYS "GetDlgItem", !dlg%, id% TO h%
SYS "EnableWindow", h%, 0
```

Here \texttt{id\%} is the identifier of the button you want to affect. To re-enable the item change the zero to a one:

```
SYS "GetDlgItem", !dlg%, id% TO h%
SYS "EnableWindow", h%, 1
```

Note that, as with initialisation, these routines must be executed \texttt{after} the call to \texttt{PROC\_showdialog}.

**Reading the contents of a dialogue box**

Since the whole purpose of a dialogue box is to receive user input, it is vital that the current contents can be determined, particularly when the \texttt{OK} button is pressed.

To read the text associated with a dialogue box item (which may for example be the contents of an edit box or the current selection of a combo box) you can use the GetDlgItemText API call:

```
DEF FNgetdlgtext(dlg%, id%)
LOCAL text%
DIM text% LOCAL 255
SYS "GetDlgItemText", !dlg%, id%, text%, 255
= $$text%
```
The parameter `dlg%` is the value returned from `FN_newdialog` and the parameter `id%` is the identifier for the item in question. The maximum length of the string is 255 characters.

In the case of a *multi-line* edit box the maximum length should be set to an appropriate value. To save the returned data to a file you can do the following:

```vbs
DEF PROC savetofile(dlg%,id%,filename$)  
LOCAL text%, Len%  
DIM text% LOCAL 65535  
SYS "GetDlgItemText", !dlg%, id%, text%, 65535 TO Len%  
OSCLI "SAVE " + filename$ + " " + STR$~text% + " " + STR$~Len%  
ENDPROC
```

The data returned from a multi-line edit box consists of lines of text separated by CRLF (CHR$13+CHR$10) sequences. If you want to process the data you can split it into individual lines as follows:

```vbs
P% = text%  
REPEAT  
A$ = $P% : REM. get line of text from memory  
PRINT A$ : REM. print the text (for example)  
P% += LEN(A$)+2 : REM. advance pointer to next line  
UNTIL P% >= (text%+Len%)
```

If the item is an edit box used for numeric entry, the current value can be read with `GetDlgItemInt`:

```vbs
SYS "GetDlgItemInt", !dlg%, id%, 0, signed% TO value%
```

The value `!dlg%` is the *handle* of the dialogue box, `id%` is the identifier for the item in question and `signed%` determines whether a negative value should be accepted (1) or not (0).

To determine which item (if any) is selected in a list box you can do the following:

```vbs
LB_GETCURSEL = 392  
SYS "SendDlgItemMessage", !dlg%, id%, LB_GETCURSEL, 0, 0 TO sel%
```
where the value id% is the identifier for the list box. The returned value sel% gives
the index (starting at 0) of the currently selected item. If no item is selected, -1 is
returned. Note that if the listbox contents have been sorted, which is the default
behaviour, knowing the index alone is of little value. However you can use it to
discover the selected text as follows:

```
DEF FNgetlistboxtext(dlg%, id%, sel%)
LOCAL text%
DIM text% LOCAL 255
LB_GETTEXT = 393
SYS "SendDlgItemMessage", !dlg%, id%, LB_GETTEXT, sel%, text% = $$$text%
```

To determine which item is selected in a combo box do the following:

```
CB_GETCURSEL = 327
SYS "SendDlgItemMessage", !dlg%, id%, CB_GETCURSEL, 0, 0 TO sel%
```

To discover the current state of a check box or radio button, you can use the
IsDlgButtonChecked API call:

```
SYS "IsDlgButtonChecked", !dlg%, id% TO state%
```

where the value id% is the identifier for the check box or radio button. The value
state% is set to 0 if the button is not checked and to 1 if it is checked.

**Determining when a button is clicked**

You must be able to determine when the user has clicked on a button, so that (for
example) when the OK button is clicked the contents of the dialogue box can be
read and the box removed from the screen. The simplest method is probably to use
the FN_setproc function when the button is created; see PROC_pushbutton for more
details. In that case once you've shown your dialogue box all you need to do is enter
an 'idle loop':

```
REPEAT
  WAIT 1
UNTIL !dlg% = 0
PROC_closdialog(dlg%)
```
Alternatively you can use a polling technique to monitor button presses and other events handled by ON SYS. The following code segment can be used to wait for the **OK** or **Cancel** button to be clicked, and then take appropriate action:

```plaintext
Click% = 0
ON SYS Click% = @wparam% : RETURN
REPEAT WAIT 1
   click% = 0
   SWAP click%,Click%
UNTIL click% = 1 OR click% = 2 OR !dlg% = 0
ON SYS OFF
IF click% = 1 THEN
   PRINT "OK pressed"
   REM. process contents of dialogue box here
ELSE
   PRINT "Cancel pressed"
ENDIF
PROC_closedialog(dlg%)
```

Since clicking the close button of a dialogue box (or floating toolbar) always produces the same ID code (2) you cannot directly tell from ON SYS which dialogue box or toolbar was closed. If your program has more than one open at the same time this could be a problem. You can determine which are still open (and therefore, by a process of elimination, which was closed) by examining the window handles (!dlg% in the above example). If the handle is non-zero the window is still open, and if it is zero it has been closed.

**PROC_closedialog**

When the user has clicked on **OK**, and the contents of the dialogue box have been processed, the box should (generally) be removed from the screen:

```plaintext
PROC_closedialog(dlg%)
```

The dialogue box *template* remains in memory, so you can display it again at any time by calling PROC_showdialog.

The dialogue box should also be removed whenever your program returns to immediate mode (for example if an error occurs or the END statement is executed) or when your program's window is closed by the user. This can be achieved by executing the following statements immediately after the call to PROC_showdialog:
Because the dialogue box uses space on the heap, it is essential that you remove it before executing a CLEAR, CHAIN or RUN statement. Failure to do so is very likely to crash *BBC BASIC for Windows*. 
Trackbars and progress bars

The WINLIB3 library contains a set of procedures and functions for creating and controlling floating toolbars, trackbars and progress bars. The library should be loaded from your program using the command:

\[
\text{INSTALL @lib$+"WINLIB3"}
\]

The functions contained are:

- FN_createfloatingtoolbar
- PROC_showfloatingtoolbar
- PROC_removefloatingtoolbar
- FN_custombuttons
- FN_createtrackbar
- PROC_showtrackbar
- FN_trackbarpos
- PROC_removetrackbar
- FN_createprogressbar
- PROC_showprogressbar
- PROC_stepprogressbar
- PROC_removeprogressbar

**FN_createfloatingtoolbar**

\[\text{(nbutts\%,button\%(),buttid\%(),xpos\%,ypos\%,title$)}\]

This function creates a floating toolbar, and returns a pointer to its window handle. It works in a similar way to FN_createtoolbar except that it needs three extra parameters: the initial horizontal and vertical coordinates of the toolbar (measured, in dialogue box units, from the top-left corner of BASIC's output window) and a string to display in the toolbar's title bar. Unlike FN_createtoolbar, the toolbar is not actually displayed until you call PROC_showfloatingtoolbar; this allows you to remove and restore the toolbar without having to re-create it each time (avoiding the memory wastage that would involve).

Normally the floating toolbar has a title bar. This can be eliminated by modifying the style after the call to FN_createfloatingtoolbar as follows:
ftb% = FN_createfloatingtoolbar(nbutts%,button%(),buttid%(),50,50,"")
WS_BORDER = &800000
ftb%!16 AND= NOT WS_BORDER

Note, however, that if you do this the user won't be able to move the toolbar, so you may need to set its position more carefully.

**PROC_showfloatingtoolbar (ftb%)**

This procedure displays a previously created floating toolbar. It requires one parameter, being the value returned from FN_createfloatingtoolbar. The following program segment creates and displays a floating toolbar containing three buttons: a cut button, a copy button and a paste button:

```plaintext
nbutts% = 3
DIM button%(nbutts%-1), buttid%(nbutts%-1)
button%() = 0, 1, 2
buttid%() = 100, 101, 102
ftb% = FN_createfloatingtoolbar(nbutts%,button%(),buttid%(),50,50,"Floating")
PROC_showfloatingtoolbar(ftb%)
```

See FN_createtoolbar for more details. If you want to send messages to the toolbar (for example to control the appearance of the buttons) you can do so using the means described under FN_createtoolbar, but note that the value returned from FN_createfloatingtoolbar is a pointer to the toolbar handle not the handle itself. For example, to cause a button to be disabled (greyed out):

```plaintext
TB_SETSTATE = 1041
SYS "SendMessage", !ftb%, TB_SETSTATE, buttid%, 0
```

Note that you can only send messages to a floating toolbar after it has been displayed with PROC_showfloatingtoolbar.

**PROC_removefloatingtoolbar**

This procedure removes a floating toolbar previously created with FN_createfloatingtoolbar. You should always remove the toolbar before your program exits or returns to immediate mode:
Because FN_createfloatingtoolbar reserves space on the heap, it is essential that you remove the toolbar before executing a CLEAR, CHAIN or RUN statement. Failure to do so is very likely to crash BBC BASIC for Windows.

**FN_custombuttons (htool %, bmpfile $, nbutts %, buttid %())**

This function is very similar to FN_custombutton except that it allows you to customise the images of multiple buttons from a single bitmap file. Four parameters must be supplied: the handle of the toolbar (e.g. as returned from FN_createtoolbar), the name of a Windows Bitmap file containing the required button images (left to right), the number of button images in the file and an array of identifiers of the buttons you wish to change.

The function returns TRUE if the button images were successfully changed, and FALSE otherwise (most likely because the specified file could not be read or has an invalid format). You can customise as many buttons as you like by providing a bitmap with suitable dimensions.

Note that when used with a floating toolbar the value returned from FN_createfloatingtoolbar is a pointer to the toolbar handle not the handle itself. So to customise all the buttons in a floating toolbar:

```
ok % = FN_custombuttons(!ftb %, "birds.bmp", nbutts %, buttid %())
```

When initially creating the toolbar set the button 'type' of any custom buttons to 15 (no image). Also ensure that the background colour of your bitmaps is R=192, G=192, B=192 (&C0C0C0). These measures will ensure that your custom images are displayed correctly.

**FN_createtrackbar (hwnd %, xpos %, ypos %, width %, height %, style %)**

This function creates a trackbar (sometimes called a slider). A trackbar allows the user to control a parameter by dragging the slider to the wanted position. Six parameters must be supplied: the window handle of the trackbar's parent window (usually @hwnd % unless you need to place the trackbar in a child window), the
horizontal and vertical coordinates of the trackbar (in pixels, measured from the top-left corner of the bar's parent window), the width and height of the trackbar (in pixels) and the trackbar style. Possible values for the style are as follows:

<table>
<thead>
<tr>
<th>Style</th>
<th>Name</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TBS_AUTOTICKS</td>
<td>Show tick marks</td>
</tr>
<tr>
<td>2</td>
<td>TBS_VERT</td>
<td>Vertical trackbar</td>
</tr>
<tr>
<td>4</td>
<td>TBS_LEFT</td>
<td>Tick marks to top or left (default is bottom or right)</td>
</tr>
<tr>
<td>8</td>
<td>TBS_BOTH</td>
<td>Tick marks on both sides</td>
</tr>
</tbody>
</table>

The values can be combined.

**PROC_showtrackbar (tb%, max%)**

This function displays a trackbar created with FN_createtrackbar. It requires two parameters: the value returned by FN_createtrackbar and the maximum value to which the slider can be moved. If tick marks are shown, the total number (including both ends) will be max%+1.

To create and display a horizontal trackbar with tick marks, suitable for selecting a value between zero and ten:

```plaintext
  tb% = FN_createtrackbar(@hwnd%, 100, 200, 20, 300, 1)
  PROC_showtrackbar(tb%, 10)
```

**FN_trackbarpos (tb%)**

This function returns the current position of the trackbar (between zero and the value of max% supplied to PROC_showtrackbar):

```plaintext
  trackpos% = FN_trackbarpos(tb%)
```

Normally the trackbar is moved by the user, but if you want your program to move it to a specific position you can do that as follows:

```plaintext
  TBM_SETPOS = 1029
  SYS "SendMessage", !tb%, TBM_SETPOS, 1, trackpos%
```
PROC_removetrackbar (tb%)  

This procedure removes a trackbar previously created with FN_createtrackbar. You should always remove the trackbar before your program exits or returns to immediate mode:

PROC_removetrackbar(tb%)

Because FN_createtrackbar reserves space on the heap, it is essential that you remove the trackbar before executing a CLEAR, CHAIN or RUN statement. Failure to do so is very likely to crash BBC BASIC for Windows.

FN_createprogressbar (hwnd%, xpos%, ypos%, width%, height%, style%)  

This function creates a progress bar which is typically used to inform the user of the progress of a time-consuming operation. Six parameters must be supplied: the window handle of the progress bar's parent window (usually @hwnd% unless you need to place the progress bar in a child window or status bar), the horizontal and vertical coordinates of the progress bar (in pixels, measured from the top-left corner of the bar's parent window), the width and height of the progress bar (in pixels) and the progress bar style. A style value of zero selects a horizontal progress bar with separate 'blocks'. Other values are as follows:

<table>
<thead>
<tr>
<th>Style</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Progress 'blocks' are contiguous</td>
</tr>
<tr>
<td>4</td>
<td>The progress bar is vertical</td>
</tr>
</tbody>
</table>

The values may be combined.

PROC_showprogressbar (pb%, max%)  

This function displays a progress bar created with FN_createprogressbar. It requires two parameters: the value returned by FN_createprogressbar and the maximum value which can be indicated. For example if the progress is shown in percent,  


max% should be 100.

To create and display a progress bar, suitable for showing a value between zero and ten:

```
pb% = FN_createprogressbar(@hwnd%, 100, 200, 20, 300, 0)
PROC_showprogressbar(pb%, 10)
```

**PROC_stepprogressbar (pb %, step %)**

This procedure advances the progress bar by the number of steps indicated by step% (which can be negative). If the end result exceeds the value of max% (supplied to PROC_showprogressbar), or is negative, it wraps around. To advance a progress bar by one step:

```
PROC_stepprogressbar(pb%, 1)
```

You can also set the progress bar to an absolute value as follows:

```
PBM_SETPOS = 1026
SYS "SendMessage", !pb%, PBM_SETPOS, progress%, 0
```

**PROC_removeprogressbar (pb %)**

This procedure removes a progress bar previously created with FN_createprogressbar. You should always remove the progress bar before your program exits or returns to immediate mode:

```
PROC_removeprogressbar(pb%)
```

Because FN_createprogressbar reserves space on the heap, it is essential that you remove the progress bar before executing a CLEAR, CHAIN or RUN statement. Failure to do so is very likely to crash **BBC BASIC for Windows**.
Property sheets and wizards

The WINLIB4 library contains a set of procedures and functions for creating and controlling property sheets and wizards. The library should be loaded from your program using the command:

```
INSTALL @lib$+%"WINLIB4"
```

The functions contained are:

- FN_newpropsheet
- PROC_showpropsheet
- PROC_closepropsheet

The WINLIB4U library contains an identical set of functions except that those receiving a text string assume it to be in Unicode (UTF-8) format, allowing non-ANSI (e.g. foreign language) characters to be incorporated.

Property sheets and wizards are very similar, both effectively consisting of multiple dialogue box pages within a single window (only one page being displayed at a given time). They differ principally in the way the individual pages are selected: in a property sheet the pages can be selected in any order (each has a tab which can be clicked by the user), whereas in a wizard the pages must be selected sequentially by means of Next and Back buttons.

Because property sheets and wizards are like multiple dialogue boxes, this is exactly how they are created in BBC BASIC for Windows. Each page must be created using the FN_newdialog function in the WINLIB2 library. However, unlike conventional dialogue boxes, you should not include OK, Apply or Cancel buttons in the individual pages.

Instead of displaying the individual dialogue boxes with PROC_showdialog they are grouped together as a property sheet or wizard using FN_newpropsheet and displayed using PROC_showpropsheet.

**FN_newpropsheet (caption$, npages%, initial%, style%, dlg%())**

This function groups together multiple dialogue boxes to form a single property
sheet or wizard. Five parameters must be supplied: a caption for the title bar, the number of dialogue box pages, the page which should be initially displayed (where the first page is numbered zero), a window style (usually 0 for a property sheet or &20 for a wizard) and an array of dialogue box pointers (each being the value returned from FN_newdialog).

The following program segment creates a property sheet containing three dialogue box pages:

```
pages% = 3
DIM page%(pages%-1)
page%(0) = FN_newdialog("First page", 32, 32, 288, 128, 8, 650)
REM create the contents of the first page here
page%(1) = FN_newdialog("Second page", 32, 32, 288, 128, 8, 1100)
REM create the contents of the second page here
page%(2) = FN_newdialog("Third page", 32, 32, 288, 128, 8, 1100)
REM create the contents of the third page here
psh% = FN_newpropsheet("Property sheet",pages%,0,0,page%())
```

Following each FN_newdialog call the contents of the relevant page are created using the procedures provided in WINLIB2, e.g. PROC_static, PROC_editbox etc. If a wizard rather than a property sheet is wanted, the last line would be:

```
psh% = FN_newpropsheet("Property sheet",pages%,0,&20,page%())
```

**PROC_showpropsheet (psh%,hdlg%())**

This procedure displays the property sheet or wizard. Two parameters must be supplied: the pointer to the property sheet returned by FN_newpropsheet and an empty array to receive the dialogue box handles. This array should be DIMensioned to have at least as many elements as there are pages in the property sheet or wizard, and on return from PROC_showpropsheet will contain the handles of each dialogue box page. These will be needed in order to initialise and interrogate the contents of the individual pages.

The following program segment displays the property sheet or wizard created above:

```
DIM hdlg%(pages%-1)
PROC_showpropsheet(psh%,hdlg%())
```
Initialising the contents of a property sheet

As with a dialogue box, the initial contents of the property sheet pages may be defined when the items within them are created. For example, the initial contents of an edit box can be specified in the PROC_editbox procedure call. However, you may want to change the contents at other times, and in the case of list boxes and combo boxes, which contain multiple items, you must use other methods to initialise the contents.

The methods for doing this are identical to those listed in Initialising the contents of a dialogue box except that whenever the dialogue box handle is needed you must supply an element from the hdlg%() array returned from PROC_showpropsheet rather than the value pointed to by FN_newdialog. For example, to change the text associated with an item in the first page of the property sheet or wizard:

```
SYS "SetDlgItemText", hdlg%(0), id%, text$
```

You should normally ensure that the item identifier id% is unique rather than being used in two or more different pages. If the specified item is not in the page corresponding to the specified dialogue box handle, the call will fail.

Determining when a property sheet button is pressed

You must be able to determine when the user has clicked on the OK or Finish button, so that the contents of the property sheet or wizard can be read and the window removed from the screen. When any item in a property sheet is clicked, a message is sent (similar to those from a menu or a toolbar) which can be detected with ON SYS.

The following code segment can be used to wait for the OK, Finish or Cancel button to be clicked, and then take appropriate action:

```
Click% = 0
ON SYS Click% = @wparam%  : RETURN
REPEAT WAIT 1
  click% = 0
  SWAP click%,Click%
UNTIL click% = 1 OR click% = 2
ON SYS OFF
IF click% = 1 THEN
  PRINT "OK or Finish pressed"
```
REM process contents of property sheet here
ELSE
    PRINT "Cancel pressed"
ENDIF
PROC_closepropsheet(psh%)

Monitoring property sheet page changes

Under some circumstances you may need to know when the user has changed the property sheet or wizard page. This is particularly true of a wizard, since it is the responsibility of the user's program to replace the Next button by a Finish button when the last page of the wizard is displayed (and also to inhibit the Back button when the first page is displayed).

You can do this by monitoring the current page handle, which can be determined by means of an API call. For example, the loop for monitoring which buttons are pressed can be modified to check also for page changes as follows:

```plaintext
PSM_SETWIZBUTTONS = 1136
PSM_GETCURRENTPAGEHWND = 1142
Click% = 0
ON SYS Click% = @wparam% : RETURN
oldhdlg% = 0
REPEAT
    WAIT 1
    click% = 0
    SWAP click%,Click%
    SYS "SendMessage", !psh%, PSM_GETCURRENTPAGEHWND, 0, 0 TO hdlg%
    IF hdlg%<>oldhdlg% THEN
        oldhdlg% = hdlg%
        CASE hdlg% OF
            WHEN hdlg%(0):
                SYS "SendMessage", !psh%, PSM_SETWIZBUTTONS, 0, 2 : REM Next only
            WHEN hdlg%(1):
                SYS "SendMessage", !psh%, PSM_SETWIZBUTTONS, 0, 3 : REM Back and Next
            WHEN hdlg%(2):
                SYS "SendMessage", !psh%, PSM_SETWIZBUTTONS, 0, 5 : REM Back and Finish
        ENDCASE
    ENDIF
    CASE click% OF
        WHEN 1: PRINT "Finish pressed"
        REM Process contents of wizard here
        WHEN 2: PRINT "Cancel pressed"
    ENDCASE
UNTIL hdlg% = 0
PROC_closepropsheet(psh%)
```
The above example is for a wizard with three pages. According to which page is currently displayed the **Back**, **Next** and **Finish** buttons are displayed appropriately.

**Reading the contents of a property sheet**

Since the whole purpose of a property sheet or wizard is to receive user input, it is vital that the current contents can be determined, particularly when the **OK** or **Finish** button is pressed.

The methods for doing this are identical to those listed in Reading the contents of a dialogue box except that whenever the dialogue box handle is needed you must supply an element from the `hdlg%()` array returned from `PROC_showpropsheet` rather than the value pointed to by `FN_newdialog`. For example, to read the text associated with an item in a property sheet or wizard:

```plaintext
DEF FNgetproptext(hdlg%, page%, id%)  
LOCAL text%  
DIM text% LOCAL 255  
SYS "GetDlgItemText", hdlg%(page%), id%, text%, 255  
= $$text%
```

**PROC_closepropsheet (psh%)**

When the contents of the property sheet or wizard have been processed, it should be removed from the screen:

```plaintext
PROC_closepropsheet(psh%)
```

The property sheet *templates* remain in memory, so you can display it again at any time by calling `PROC_showpropsheet`.

The property sheet or wizard should also be removed whenever your program returns to immediate mode (for example if an error occurs or the END statement is executed) or when your program’s window is closed by the user. This can be achieved by executing the following statements immediately after the call to `PROC_showpropsheet`:

```plaintext
ON CLOSE PROC_closepropsheet(psh%):QUIT
ON ERROR PROC_closepropsheet(psh%):PRINT'REPORT$':END
```
Because the property sheet uses space on the *heap*, it is essential that you remove it before executing a CLEAR, CHAIN or RUN statement. Failure to do so is very likely to crash *BBC BASIC for Windows*. 
Boxes and buttons

Windows™ programs containing push buttons, edit boxes and so on generally incorporate them within dialogue boxes. BBC BASIC for Windows supports this use by means of the WINLIB2 library. However it is perfectly possible to incorporate such items directly within your program's output window, or indeed as completely independent windows.

The WINLIB5 and WINLIB5A libraries contain a set of procedures and functions for incorporating push buttons, edit boxes etc. in your program without the need to create a dialogue box to contain them. The library should be loaded from your program using the command:

    INSTALL @lib$+"WINLIB5"

or

    INSTALL @lib$+"WINLIB5A"

The functions contained are:

- FN_button
- FN_combobox
- FN_editbox
- FN_listbox
- FN_staticbox
- FN_createwindow
- PROC_closewindow
- PROC_setfocus
- FN_setproc

The WINLIB5A library contains an identical set of functions but each takes an additional first parameter: the parent or owner window handle. Use this library instead of WINLIB5 when you need to place the buttons, boxes or controls on (or for them to be owned by) a window other than your main output window. For example you might wish to place a combo box on a toolbar or rebar.

The WINLIB5U library contains an identical set of functions to WINLIB5A, except that those receiving a text string assume it to be in Unicode (UTF-8) format,
allowing non-ANSI (e.g. foreign language) characters to be incorporated.

**FN_button (text$, x%, y%, cx%, cy%, id%, style%)**

This function creates a rectangular pushbutton. The string `text$` specifies the text to appear inside the button, the values `x%` and `y%` specify the position of the button and the values `cx%` and `cy%` specify the size of the button (in pixels, where 0,0 is the top-left corner of the window).

The value `id%` is a unique identifier of the pushbutton, and can be any constant you choose (within reason) or a value returned from FN_setproc. The value `style%` can be zero, but other values allow you to modify the appearance or behaviour of the button, for example the value &100 (BS_LEFT) causes the text to be left-justified rather than centered, &80 (BS_BITMAP) creates a **bitmap button**, 3 (BS_AUTOCHECKBOX) creates a **checkbox** and 9 (BS_AUTORADIOBUTTON) creates a **radiobutton**.

Clicking on the button causes an ON SYS event in the same way as a menu selection, with `@wparam%` equal to the value of `id%`.

The function returns the window handle of the button, which is needed when removing the button with PROC_closewindow.

To load an image into a bitmap button do:

```plaintext
LR_LOADFROMFILE = 16
BM_SETIMAGE = 247
SYS "LoadImage", 0, bmpfile$, 0, cx%, cy%, LR_LOADFROMFILE TO hbitmap%
SYS "SendMessage", hbutton%, BM_SETIMAGE, 0, hbitmap%
```

The value `bmpfile$` is the name of a Windows™ bitmap file containing the image, `cx%` and `cy%` are the dimensions of the image in pixels and `hbutton%` is the value returned from FN_button. Once you have finished with the button (but not before) delete the bitmap handle:

```plaintext
SYS "DeleteObject", hbitmap%
```

**FN_combobox ("", x%, y%, cx%, cy%, id%, style%)**
This function creates a combo box. The text string is unused and should be set to an empty string, the values \texttt{x\%} and \texttt{y\%} specify the position of the box and the values \texttt{cx\%} and \texttt{cy\%} specify the size of the box (in pixels, where \texttt{0,0} is the top-left corner of the window).

The value \texttt{id\%} is a unique identifier of the combo box, and can be any constant you choose (within reason) or a value returned from \texttt{FN_setproc}. The value \texttt{style\%} can be zero, but other values allow you to modify the appearance or behaviour of the box, for example the value 3 (CBS_DROPDOWNLIST) creates a \textit{drop down list} combo box. The function returns the window handle of the box.

To enter a list of strings into a combo box do:

\begin{verbatim}
CB_ADDSTRING = 323
SYS "SendMessage", hbox\%, CB_ADDSTRING, 0, "Combobox item 0"
SYS "SendMessage", hbox\%, CB_ADDSTRING, 0, "Combobox item 1"
etc.
\end{verbatim}

where \texttt{hbox\%} is the value returned from \texttt{FN_combobox}. An initial (default) selection can be made as follows:

\begin{verbatim}
CB_SETCURSEL = 334
SYS "SendMessage", hbox\%, CB_SETCURSEL, index\%, 0
\end{verbatim}

To determine which item is selected in a combo box do:

\begin{verbatim}
CB_GETCURSEL = 327
SYS "SendMessage", hbox\%, CB_GETCURSEL, 0, 0 TO sel\%
\end{verbatim}

\textbf{FN_editbox (text\$,x\%,y\%,cx\%,cy\%,id\%,style\%)}

This function creates an edit box. The string \texttt{text\$} specifies the initial contents of the box, the values \texttt{x\%} and \texttt{y\%} specify the position of the box and the values \texttt{cx\%} and \texttt{cy\%} specify the size of the box (in pixels, where \texttt{0,0} is the top-left corner of the window).

The value \texttt{id\%} is a unique identifier of the edit box, and can be any constant you choose (within reason) or a value returned from \texttt{FN_setproc}. The value \texttt{style\%} can be zero, but other values allow you to modify the appearance or behaviour of the box.
box, for example the value &80 (ES_AUTOHSCROLL) allows the contents of the box to scroll horizontally. The function returns the window handle of the box.

To read the contents of an edit box do:

```
DEF FN gettext(hbox%)
LOCAL text%
DIM text% LOCAL 65535
SYS "GetWindowText", hbox%, text%, 65535
= $$text%
```

where the parameter `hbox%` is the value returned from `FN_editbox`.

To create a scrollable, multi-line edit box you can specify the style value as `WS_VSCROLL OR ES_AUTOHSCROLL OR ES_MULTILINE`. To create an edit control which behaves as an independent window, with its own title bar, system menu and close box you can additionally specify the style value as `WS_POPUP OR WS_CAPTION OR WS_SYSMENU`. Note that in this latter case you **must** set the `id%` value to zero (it has the meaning of menu handle in the case of a popup window).

```
FN_listbox ("",x%,y%,cx%,cy%,id%,style%)
```

This function creates a list box. The text string is unused and should be set to an empty string, the values `x%` and `y%` specify the position of the box and the values `cx%` and `cy%` specify the size of the box (in pixels, where `0,0` is the top-left corner of the window).

The value `id%` is a unique identifier of the list box, and can be any constant you choose (within reason) or a value returned from `FN_setproc`. The value `style%` can be zero, but other values allow you to modify the appearance or behaviour of the box, for example the value 2 (LBS_SORT) causes the contents of the box to be sorted. The function returns the window handle of the box.

To enter a list of strings into a list box do:

```
LB_ADDSTRING = 384
SYS "SendMessage", hbox%, LB_ADDSTRING, 0, "Listbox item 0"
SYS "SendMessage", hbox%, LB_ADDSTRING, 0, "Listbox item 1"
etc.
```

where `hbox%` is the value returned by `FN_listbox`. To determine which item is
selected do:

```plaintext
LB_GETCURSEL = 392
SYS "SendMessage", hbox%, LB_GETCURSEL, 0, 0 TO sel%
```

The value `sel%` gives the index (starting at 0) of the currently selected item or -1 if no item is selected.

**FN_staticbox (text$, x%, y%, cx%, cy%, id%, style%)**

This function creates a static box. The string `text$` specifies the contents (if a text box), the values `x%` and `y%` specify the position of the box and the values `cx%` and `cy%` specify the size of the box (in pixels, where 0,0 is the top-left corner of the window).

The value `id%` is a unique identifier of the static box, and can be any constant you choose (within reason) or a value returned from FN_setproc. The value `style%` can be zero, but other values allow you to modify the appearance or behaviour of the box, for example the value &E (SS_BITMAP) indicates that the box will contain a bitmap image. The function returns the window handle of the box.

To load a bitmap image into a static box do:

```plaintext
LR_LOADFROMFILE = 16
STM_SETIMAGE = 370
SYS "LoadImage", 0, bmpfile$, 0, cx%, cy%, LR_LOADFROMFILE TO hbitmap%
SYS "SendMessage", hbox%, STM_SETIMAGE, 0, hbitmap%
```

The value `bmpfile$` is the name of a Windows™ bitmap file containing the image, `cx%` and `cy%` are the dimensions of the image in pixels and `hbox%` is the value returned from FN_staticbox. Once you have finished with the box (but not before) delete the bitmap handle:

```plaintext
SYS "DeleteObject", hbitmap%
```

**FN_createwindow (class$, text$, x%, y%, cx%, cy%, id%, style%, exst%)**
This function can be used to create child window types other than those described above. The parameters have the same meaning as for the other types except that `class$` is the window class name and `exst%` is an extended window style. The function returns the window handle.

Note that the `style%` parameter is exclusive-ORed with `&50000000`, which is the numeric equivalent of `WS_CHILD + WS_VISIBLE`. If you prefer the window not to be constrained within the bounds of its parent, add `&C0000000` (`WS_CHILD + WS_POPUP`) to the supplied parameter.

**PROC_closewindow (hwnd%)**

This procedure can be used to close (i.e. remove) any window created with the above functions. The value `hwnd%` is the window handle returned from the appropriate function.

**PROC_setfocus (hwnd%)**

This procedure can be used to control which window has the input focus (i.e. receives keyboard input). The value `hwnd%` is the handle of the window to receive the focus, which can be the value returned from one of the above functions or `@hwnd%` in order for your BASIC program to receive keyboard input again.

**FN_setproc (PROCname)**

This function (available in version 1.4 or later of WINLIB5) takes as a parameter the name of a procedure and returns an ID number which you can use when creating a control or a menu item. Using this function automates the process of executing a procedure when (for example) a button or menu item is clicked, without needing to use `ON SYS`. Here are some examples of its use:

To create a menu, in which selecting **Open** causes **PROCopen** to be executed and selecting **Exit** causes **PROCexit** to be executed:

```plaintext
SYS "CreatePopupMenu" TO hfile%
SYS "AppendMenu", hfile%, 0, FN_setproc(PROCopen), "&Open"
SYS "AppendMenu", hfile%, 0, FN_setproc(PROCexit), "E&xit"
```

To create buttons in a dialogue box, where clicking on **Button 1** causes
PROCbutton1 to be executed and clicking on Button 2 causes PROCbutton2 to be executed:

```plaintext
INSTALL @lib$:"WINLIB2"
dlg% = FN_newdialog("Button test", 200, 100, 100, 100, 8, 1000)
PROC_pushbutton(dlg%, "Button 1", FN_setproc(PROCbutton1), 20, 10, 64, 16, 0)
PROC_pushbutton(dlg%, "Button 2", FN_setproc(PROCbutton2), 20, 32, 64, 16, 0)
```

To create buttons on your main window, where clicking on Button 3 causes PROCbutton3 to be executed and clicking on Button 4 causes PROCbutton4 to be executed:

```plaintext
hbutt3% = FN_button("Button 3", 300, 20, 100, 24, FN_setproc(PROCbutton3), 0)
hbutt4% = FN_button("Button 4", 300, 90, 100, 24, FN_setproc(PROCbutton4), 0)
```

If you need to know the values of @wparam% and/or @lparam% (unlikely in the case of menu selections or button presses, but possible with other controls) you can arrange for them to be passed to your procedure by adding a pair of parentheses when you call FN_setproc, as follows:

```plaintext
hedit% = FN_editbox("", 300, 20, 100, 24, FN_setproc(PROCeditbox()), 0)
```

You must then define your procedure to receive two parameters:

```plaintext
DEF PROCeditbox(W%, L%)
```

where W% and L% receive the values of @wparam% and @lparam% respectively.
Sprites

The **SPRITELIB** library contains a set of procedures and functions for creating and controlling sprites. A sprite is a graphical object which is displayed *in front of*, and independent of, whatever else is plotted in the BBC BASIC window. A sprite can have any shape and pattern: where it is opaque it will completely obscure what is behind and where it is transparent the normal window contents will show through. You can move a sprite around without having to worry about redrawing the background which is *revealed*.

The library should be loaded from your program using the command:

```
INSTALL @lib$$"SPRITELIB"
```

The functions contained are:

- **FN_initsprites**
- **FN_createsprite**
- **FN_createspritefrombmp**
- **PROC_movesprite**
- **PROC_updatesprite**
- **PROC_exitsprites**

**FN_initsprites (nsprites%)**

This function initialises the sprite system. It requires one parameter: the maximum number of sprites which will need to be displayed. This parameter should be set as small as possible, since the larger the number of sprites the more the memory used, and the greater the slowing effect on the display.

```
IF FN_initsprites(2) = 0 STOP
```

The returned value is TRUE if the sprite system was initialised successfully and FALSE if not. The function will fail if the supplied parameter is zero, or if there is insufficient memory.

**FN_createsprite (sprite%,file$,width%,height%)**
This function creates a sprite of a given size and with a specified shape and appearance. It requires four parameters: the number of the sprite (which must be between zero and one less than the parameter supplied to \texttt{FN\_initsprites} ), the name of the file containing the sprite image (and transparency mask), the width of the sprite and the height of the sprite (both in pixels).

The file must be an Icon-format file (usually having the extension .ICO) generated using a suitable icon editor. \textit{BBC BASIC for Windows} is supplied with a simple icon editor (ICONEDIT.BBC) which will suffice if no other editor is available. Under normal circumstances you should specify the same dimensions in the \texttt{FN\_createsprite} call as were used when the icon was created, however if you do not do so the sprite will be \textit{scaled} to the specified size (with some attendant loss of quality).

\begin{verbatim}
  ok% = FN\_createsprite(0, "bbcmicro.ico", 64, 64)
\end{verbatim}

The returned value is non-zero if the sprite was created successfully, and zero otherwise. The most likely reason for the function to fail is if the file does not exist or is not a suitable icon-format file.

The sprite is not displayed until \texttt{PROC\_movesprite} or \texttt{PROC\_updatesprite} is executed.

\textbf{FN\_createspritefrombmp} (\texttt{sprite\%,bmpfile$})

This function is similar to \texttt{FN\_createsprite} except that instead of an icon (.ICO) file being required a bitmap (.BMP) file must be specified, and there are no size parameters (the sprite may be scaled by passing appropriate width and height values to \texttt{PROC\_updatesprite} ). The function makes it straightforward to create a sprite at run-time, since a suitable BMP file can be generated using the \texttt{*GSAVE} command. The sprite may be any size (within reason) and is not limited to 256 x 256 pixels as would be the case for an icon file.

The BMP file must contain a single bitmap which is \textbf{twice the height} of the sprite, the upper half of which contains the sprite's transparency mask and the lower half of which contains the sprite's image. Here is an example of a suitable bitmap:
The mask is white where the sprite is fully transparent and black where the sprite is fully opaque (intermediate transparency values are not supported). In areas where the sprite is transparent its image should be black.

**PROC_movesprite (sprite%, xpos%, ypos%, show%)**

This function displays (or hides) a sprite, and moves it to a specified position on the screen. It requires four parameters: the number of the sprite, the horizontal and vertical coordinates of the sprite (in BBC BASIC graphics units) and a value which determines whether the sprite should be displayed (1) or not (0). The coordinates refer to the *top left-hand corner* of the sprite (which is consistent with text characters plotted in the VDU 5 mode).

```
PROC_movesprite(0, 200, 200, 1)
```

In the case of an *animated sprite* (.ANI file) the show% parameter specifies the required frame number (+1), so 1 causes the first frame to be displayed, 2 causes the second frame to be displayed and so on.

Sprites are unaffected by the current graphics window (if any) and always display *in front of* any other graphics or text. Sprites have a predefined priority order such that, if they overlap, a higher-numbered sprite always appears in front of a lower-numbered sprite.

**PROC_updatesprite (sprite%, xpos%, ypos%, show%, wide%, high%)**

This procedure is similar to PROC_movesprite except that it takes additional *wide%* and *high%* parameters. These parameters may be set to the desired width
and height of the sprite in pixels, respectively. So for example a sprite could be made to get smaller or larger as it moves.

In addition, by specifying **negative** values for `wide%` and/or `high%` it is possible to reflect the sprite about its horizontal or vertical axis (or both). Note that this feature works only for sprites having a 1-bit mask (not a linear alpha mask) and relies on an undocumented feature of Windows.

**PROC_exitsprites**

When you have finished with the sprites, you should disable them:

```
PROC_exitsprites
```

Because the sprite routines use space on the heap, it is essential that you call **PROC_exitsprites** before executing a CLEAR, CHAIN or RUN statement. Failure to do so is very likely to crash *BBC BASIC for Windows*. 
Formatting and conversion

BBC BASIC does not include the PRINT USING statement which, in many dialects of BASIC, provides a convenient means of formatting numeric output. Although equivalent effects may be obtained by using the format-control variable @%, if necessary in conjunction with string-manipulation functions (e.g. MID$), they are less straightforward to achieve.

BBC BASIC also lacks the UPPER$ (or UCASE$) and LOWER$ (or LCASE$) functions provided in some dialects of BASIC to convert strings to uppercase (capitals) or lowercase characters respectively.

The FNUSING library provides replacements for these operations. It should be loaded from your program using the command:

    INSTALL @lib$+"FNUSING"

Alternatively, since the functions are quite short, you might prefer to incorporate them in your own program (use the Insert command from the File menu).

The functions contained are:

    - FNusing
    - FNlower
    - FNupper

**FNusing(format$,value)**

The FNusing function takes two parameters, a format string and a numeric value. It will normally be used within a PRINT statement in the following context:

    PRINT FNusing(fmt1$,val1) FNusing(fmt2$,val2) ..... 

A significant difference from the conventional PRINT USING statement is that each format string can only refer to one numeric value, so you must call FNusing for each value you want to output.

The format string is a string literal or variable containing special formatting
The hash character is used to represent a digit position. Digit positions are always filled: if the number has fewer digits than positions specified it is right-justified (preceded by spaces) in the field. A decimal point may be inserted at any position in the field and numbers are rounded as necessary. For example:

```
PRINT FNusing("##.##", .78)
  0.78
PRINT FNusing("###.##", 987.654)
  987.65
```

A plus sign at the beginning or end of the format field causes the sign of the number (plus or minus) to be printed before or after the number. For example:

```
PRINT FNusing("++++.##", 2.4)
  +2.40
PRINT FNusing("###.##+", 55.678)
  55.68+
PRINT FNusing("###.##", -3)
   3.00-
```

A minus sign at the end of the format field causes negative numbers to be printed with a trailing minus sign. For example:

```
PRINT FNusing("###-", -68.95)
  68.95-
PRINT FNusing("###-", -7)
   7.00-
```

A double asterisk at the beginning of the format field causes leading spaces in the field to be filled with asterisks. The ** also specifies two more digit positions. For example:

```
PRINT FNusing("**.#", 12.39)
  *12.4
PRINT FNusing("***.##", -0.9)
 **-0.90
```

A double dollar (or pound) sign at the beginning of the format field causes a dollar (or pound) sign to be printed to the immediate left of the formatted number. The $$ also specifies two more digit positions, one of which is the currency symbol. For example:
**$** A **$** (or **£**) at the beginning of the format field combines the effects of the previous two formats. Leading spaces are filled with asterisks, and a dollar (or pound) sign is printed before the number. **$** specifies three more digit positions, one of which is the currency symbol. For example:

```
PRINT FNusing("**$##.##", 2.34)
***$2.34
```

, A comma to the left of the decimal point in the format string causes a comma to be printed between every third digit before the decimal point. For example:

```
PRINT FNusing("#,###.##", 1234.5)
1,234.50
PRINT FNusing("##,###,###", 1E6)
1,000,000
```

^^^^ Four carets may be placed after the digit characters to specify exponential format. The four carets allow space for "E-xx" to be printed. For example:

```
PRINT FNusing("##.##^^^^", 234.56)
2.35E2
PRINT FNusing("##.##^^^^", 1E-30)
1.00E-30
```

If the format string includes any characters other than those listed above, they are incorporated *verbatim* in the output string. For example:

```
PRINT FNusing("Price ££#.## including VAT", 29.99)
Price £29.99 including VAT
```

If the number cannot be represented in the format supplied, question marks are printed:

```
PRINT FNusing("##.##", 123)
??????
```

**FNlower(string$)**
The **FNlower** function takes a string parameter and returns a string in which capital letters (A to Z inclusive), if any, have been converted to lowercase (a to z).

```plaintext
PRINT FNlower("The Quick Brown Fox")
the quick brown fox
```

**FNupper(string$)**

The **FNupper** function takes a string parameter and returns a string in which lowercase letters (a to z inclusive), if any, have been converted to capitals (A to Z).

```plaintext
PRINT FNupper("The Quick Brown Fox")
THE QUICK BROWN FOX
```
Multiple Document Interface

The **MDILIB** library contains a set of procedures and functions for using the Windows™ *Multiple Document Interface* (MDI). This allows a program to open multiple output windows, typically so the user can work with more than one document at a time or with more than one type of view. For example, a spreadsheet program might use one window for a tabular view of the data and another window for a graphical view.

The library should be loaded from your program using the command:

```
INSTALL @lib$+"MDILIB"
```

The functions contained are:

- PROC_initmdi
- PROC_exitmdi
- FN_createmdichild
- FN_hdc
- PROC_closemdichild
- PROC_tile
- PROC_cascade

**PROC_initmdi (menu%)**

This procedure initialises the Multiple Document Interface. It requires one parameter: the handle of the **Window** top-level menu. Every MDI program must have a Window menu, because the contents of that menu are automatically modified by the operating system as **child windows** are added and removed. Therefore as a minimum your program should include code similar to the following:

```pascal
SYS "CreatePopupMenu" TO hwindow%
SYS "CreateMenu" TO hmenu%
SYS "AppendMenu", hmenu%, 16, hwindow%, "&Window"
SYS "SetMenu", @hwnd%, hmenu%
SYS "DrawMenuBar", @hwnd%
PROC_initmdi(hwindow%)
```
See the section on adding popup and sub-menus for more details. **PROC_initmdi** must be called just once at the start of your program; once it has been called no output to the screen is possible until you have created one or more child windows (if necessary you can still display information using a message box).

**PROC_exitmdi**

This procedure shuts down the Multiple Document Interface. It must be called before your program exits and before executing a CLEAR, CHAIN or RUN statement; failure to do so is very likely to crash *BBC BASIC for Windows*. You are advised to include ON CLOSE and ON ERROR statements in your program so that you can ensure **PROC_exitmdi** is called as necessary.

**FN_createmdichild** (name$)

This function creates a *child window* and gives it the specified name; the name is displayed in the title bar of the child window and in the **Window** menu. If the child window is *maximised* it no longer has a title bar, but the name is then automatically appended to the main program's title.

The child window is created with a default size and position, but your program can change that subsequently if necessary. It can be minimised, maximised, re-sized and moved by the user in the usual way, except that the child window is constrained to remain within the confines of your program's main window. You can create as many child windows as you like (within reason!).

**FN_createmdichild** returns the *window handle* of the child window it has created:

```plaintext
hwnd1% = FN_createmdichild("Hello world")
hwnd2% = FN_createmdichild("Rectangles")
hwnd3% = FN_createmdichild("Circles")
```

Once you have created a child window you can send output to it in the same way as you would to BASIC's normal output window, however you must first select the appropriate window by changing the values of `@hwnd%` and `@memhdc%`. The easiest way of doing that (and of ensuring they are restored to their original values afterwards) is to pass them as parameters of a procedure. So for example a procedure for writing text to a child window might be:
To write a text string to a particular window you would call this procedure as follows:

PROCwritetext("Some text", hwnd1%, FN_hdc(hwnd1%))

(see below for a description of \textbf{FN\_hdc} )

As it stands this will work fine when writing text to just one window, but if you write text to two or more windows concurrently you will find that all the windows share the same text output position, so sending a 'newline' to one window will affect the subsequent position of text written to another window. One way to provide each child window with its own 'private' text position is to pass the X and Y coordinates to the procedure:

DEF PROCwritetext(A$, @hwnd%, @memhdc%, RETURN @vdu%!48, RETURN @vdu%!52)
PRINT A$
ENDPROC

which you would call as follows:

PROCwritetext("Text for window 1", hwnd1%, FN_hdc(hwnd1%), xpos1%, ypos1%)
PROCwritetext("Text for window 2", hwnd2%, FN_hdc(hwnd2%), xpos2%, ypos2%)

Since they are passed by reference the variables containing the text coordinates for each window are automatically updated. All the other text and graphics parameters (colour, plotting mode, window positions etc.) are similarly shared between the child windows, so you may need to set the parameters appropriate to that window before performing any output.

Note that (in \textit{BBC BASIC for Windows} ) MDI child windows do not display the text cursor (caret). User input, when required, should normally be done via a dialogue box .

\textbf{FN\_hdc (hwnd\%)}
This function returns the memory hdc belonging to the specified window handle. This is needed when selecting a particular child window for output (it must be copied into @memhdc% ).

**PROC_closemdichild (hwnd%)**

This procedure closes the specified child window. It has the same effect as the user clicking on the window's close button or using the keyboard shortcut Ctrl+f4.

You cannot prevent the user closing a child window (there is no direct equivalent to ON CLOSE) but you can detect that he has closed it using the IsWindow API call:

```plaintext
SYS "IsWindow", hwnd1% TO res%
IF res% = 0 THEN
  REM child window has been closed
  REM take appropriate action if necessary
ENDIF
```

**PROC_tile**

This procedure causes Windows™ to tile all the currently-open child windows within your program's output window. It would typically be called in response to the user selecting the Tile item in the Window menu.

**PROC_cascade**

This procedure causes Windows™ to cascade all the currently-open child windows within your program's output window. It would typically be called in response to the user selecting the Cascade item in the Window menu.
Calendar functions

The **DATELIB** library contains a set of procedures and functions for performing operations on dates. The library should be loaded from your program using the command:

```
INSTALL @lib$+"DATELIB"
```

The functions contained are:

- FN_mjd
- FN_day
- FN_month
- FN_year
- FN_dow
- FN_dim
- FN_today
- FN_date$
- FN_readdate

**FN_mjd (day%, month%, year%)**

This function takes a date (consisting of a day-of-month, a month and a year) and converts it to the corresponding Modified Julian Day number. The Modified Julian Day is a count of days starting from Wednesday 17th November 1858 (which is MJD 0). Days prior to that date have negative MJD numbers. You can easily calculate the number of days between two different dates by subtracting their Modified Julian Day numbers.

The parameters supplied are the day of the month (1-31), the month number (1-12) and the year number (1-9999). Note that the functions in the DATELIB library will behave consistently for any date in that range (for example, converting from DMY to MJD and back will return the original values) but should not normally be used for dates prior to the introduction of the Gregorian calendar (in the UK on Thursday 14th September 1752, MJD -38779). For earlier dates the day, month and year values may not be correct, and since use of the old Julian calendar persisted in some countries until as late as 1927 care should be taken when using this function.
FN_day (mjd\%)  
This function takes a Modified Julian Day number and returns the day-of-month (1-31) to which it corresponds.

FN_month (mjd\%)  
This function takes a Modified Julian Day number and returns the month (1-12) to which it corresponds.

FN_year (mjd\%)  
This function takes a Modified Julian Day number and returns the year (1-9999) to which it corresponds.

FN_dow (mjd\%)  
This function takes a Modified Julian Day number and returns the day-of-week (0-6, where 0 is Sunday) to which it corresponds.

FN_dim (month\%, year\%)  
This function takes a month (1-12) and a year (1-9999) and returns the number of days in the month (in the range 28 to 31). By setting month\% to 2 (February) you can use this function to determine whether the year is a Leap Year.

FN_today  
This function returns the Modified Julian Day number corresponding to today's date (assuming the PC's clock is correctly set).

FN_date$ (mjd\%, format$)  
This function takes a Modified Julian Day number and a format string, and returns a formatted string containing the date. The format string can contain any of the following codes:
d Day of month as digits with no leading zero.
dd Day of month as digits with leading zero for single-digit days.
ddd Day of week as a three-letter abbreviation.
dddd Day of week as its full name.
M Month as digits with no leading zero.
MM Month as digits with leading zero for single-digit months.
MMM Month as a three-letter abbreviation.
MMMM Month as its full name.
y Year as last two digits, but with no leading zero.
yy Year as last two digits, but with leading zero for years less than 10.
yyyy Year represented by full four digits.

For example:

```vba
date$ = FN_date$(mjd%, "ddd dd MMM yyyy")
```

will return a string of the form "Sun 22 Feb 2004".

**FN_readdate (date$, code$, minyear%)**

This function parses a string containing a date, and returns the corresponding Modified Julian Day number. In addition to the date you must supply a string containing one of the following codes: "dmy", "mdy", "ymd", "ydm", "dym" or "myd"; this informs the function of the order in which the various elements of the date (day, month, year) are present in the string. If the year is specified as a four digit number the third parameter is not used; if only the last two digits of the year are specified the third parameter is the *minimum* year number to return. For example if the third parameter is 1950, two digit year numbers correspond to the years 1950 to 2049 inclusive.

The **FN_readdate** function attempts to make sense of the date string however it is formatted, so long as the elements are in the specified order. For example it will accept "22/2/2004", "22 Feb 04", "22-02-04" etc. If it cannot make sense of the string it will return the value &80000000.
Direct3D graphics

The **D3DLIB** (etc.) libraries contain a set of procedures and functions for displaying and animating three-dimensional graphics. They provide an interface to Microsoft's **Direct3D™** and require **DirectX** version 8.0 or later to be installed (version 9.0 or later for D3D9LIB and D3D9LIBA). The required library should be loaded from your program using a command similar to this:

```
INSTALL @lib$+"D3DLIB"
```

The functions contained are:

- `FN_initd3d`
- `FN_load3d`
- `FN_loadtexture`
- `PROC_release`
- `FN_f4`
- `PROC_render`

The libraries with an **A** suffix (D3DLIBA.BBC, D3D9LIBA.BBC) provide an additional parameter to the `PROC_render` function containing the camera's roll-angle.

**FN_initd3d (hw%,cull%,light%)**

This function initialises the Direct3D system and returns a pointer to a Direct3D device. If the returned value is zero it probably indicates that DirectX 8.0 or later is not installed.

The value **hw%** is the *handle* of the window which is to contain the 3D graphics. It can be set to `@hwnd%` if the graphics are to be displayed in BBC BASIC's main output window or to the handle of a *child window* (for example as returned from `FN_staticbox` ) if you want them to appear in a separate frame. Note that you cannot mix Direct3D graphics and normal BBC BASIC output (text or graphics) in the same window.

The value **cull%** specifies the *culling mode*, which determines whether surfaces behave as *single sided* or *double sided*. Possible values are 1 (none), 2 (clockwise)
The value \texttt{light\%} determines whether Direct3D's lighting engine is enabled. Set to 1 to enable lighting or to 0 to disable lighting. When lighting is disabled all objects appear normally as if uniformly illuminated. When lighting is enabled it is necessary for all objects to include \textit{surface normals} in the vertex description.

\textbf{FN\_load3d (pdev\%, file\%, num\%, fmt\%, size\%)}

This function loads an object or scene (comprising a set of triangles, each consisting of three vertices) from a file. It returns a pointer to a vertex buffer; if zero is returned the file could not be opened.

The value \texttt{pdev\%} is the pointer returned from \texttt{FN\_initd3d}. The value \texttt{file\%} is the name of a file containing vertex data.

The values \texttt{num\%}, \texttt{fmt\%} and \texttt{size\%} are \textit{outputs} from \texttt{FN\_load3d} and are set to the number of vertices, the vertex format and the size in bytes of each vertex respectively.

The file format is as follows:

- Number of vertices (4 bytes, LSB first)
- Vertex format (2 bytes, LSB first)
- Vertex size in bytes (2 bytes, LSB first)
- Data for each vertex (see below)

\textbf{Vertex description}

The vertex data must conform to the requirements of Direct3D's \textit{flexible vertex format}. Each vertex consists of one or more of the following items, in the specified order:
<table>
<thead>
<tr>
<th>Code</th>
<th>Size</th>
<th>Data</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;002</td>
<td>12</td>
<td>XYZ position</td>
<td>Always required</td>
</tr>
<tr>
<td>&amp;010</td>
<td>12</td>
<td>Surface normal</td>
<td>When lighting used</td>
</tr>
<tr>
<td>&amp;040</td>
<td>4</td>
<td>Diffuse colour</td>
<td>When neither texture nor material specified</td>
</tr>
<tr>
<td>&amp;080</td>
<td>4</td>
<td>Specular colour</td>
<td>For shiny objects</td>
</tr>
<tr>
<td>&amp;100</td>
<td>8</td>
<td>UV texture coordinates</td>
<td>When texture specified</td>
</tr>
</tbody>
</table>

To obtain the vertex format code add together the codes for the items included. To obtain the vertex size add together the sizes of the items included. The **XYZ position** and **surface normal** items each consist of three 4-byte floating point numbers (see FN_f4 ). The **diffuse colour** and **specular colour** items each consist of 4-byte colour values (&FFrrggbb). The **texture coordinates** consist of a pair of 4-byte floating point numbers. The simplest vertex description consists of an **XYZ position** and a **diffuse colour** (format &042; size 16 bytes). See FN_f4 for an example of creating a file in this format. Refer to Microsoft documentation for more details.

**FN_loadtexture (pdev%,file$)**

This function loads a texture map from an image file (BMP, ICO, GIF, JPEG, EMF or WMF format) and returns a pointer to the texture; if zero is returned the file could not be opened or recognised. The filename must begin with a drive letter to distinguish it from a web URL.

The value `pdev%` is the pointer returned from FN_initd3d . The value `file$` is the name of the image file.

The image will be padded to a size of $2^n$ pixels in both horizontal and vertical directions.

**PROC_release (pobj%)**

This function releases an object (Direct3D device, vertex buffer or texture) when it is no longer required. It should be used to free the resources used by the object(s), otherwise you may eventually run out of memory.
The value `pobj%` is the pointer returned from FN_initd3d, FN_load3d or FN_loadtexture.

**FN_f4 (num)**

This function converts a number to a 4-byte (single precision) floating point value. All floating point values used by Direct3D (e.g. within vertex descriptions) are in this format. As an illustration of its use the following code segment creates a file containing a single triangle consisting of three vertices, suitable for being loaded by FN_load3d:

```plaintext
F% = OPENOUT"TRIANGLE.B3D"
PROC4(3):REM 3 vertices
PROC4(&100042):REM vertex size &10 and format &42
PROC4(FN_f4(-1.0)):PROC4(FN_f4(-1.0)):PROC4(FN_f4(1.0)):PROC4(&FF0000FF)
PROC4(FN_f4(1.0)):PROC4(FN_f4(-1.0)):PROC4(FN_f4(1.0)):PROC4(&FF00FF00)
PROC4(FN_f4(0.0)):PROC4(FN_f4(1.0)):PROC4(FN_f4(0.0)):PROC4(&FFFF0000)
CLOSE #F%
DEF PROC4(A%):BPUT#F%,A%:BPUT#F%,A%>>8:BPUT#F%,A%>>16:BPUT#F%,A%>>24:ENDPROC
```

**PROC_render (pdev%,...)**

This procedure draws a 2D view of the 3D world to the screen. It takes 23 parameters as follows (24 parameters in the case of D3DLIBA and D3D9LIBA):

- `pdev%` The value returned from FN_initd3d.
- `bcol%` The background colour (&FFrrggbb).
- `nlight%` The number of lights. Set to zero if lighting is not used.
- `light%()` An array of pointers to D3DLIGHT8 structures (see note 1).
- `nobj%` The number of objects (i.e. vertex buffers).
- `mat%()` An array of pointers to D3DMATERIAL8 structures (see note 2).
- `tex%()` An array of texture pointers (e.g. returned from FN_loadtexture).
- `vbuf%()` An array of vertex buffer pointers (e.g. returned from FN_load3d).
- `vnum%()` An array of vertex counts (e.g. returned from FN_load3d).
- `vfmt%()` An array of vertex format codes (e.g. returned from FN_load3d).
- `vsize%()` An array of vertex sizes (e.g. returned from FN_load3d).
- `yaw()` An array of yaw angles (rotations about the Y-axis).
pitch() An array of pitch angles (rotations about the X-axis).
roll() An array of roll angles (rotations about the Z-axis).
X() An array of translations along the X-axis.
Y() An array of translations along the Y-axis.
Z() An array of translations along the Z-axis.
eye() An array eye(0), eye(1), eye(2) holding the XYZ coordinates of the eye or camera.
look() An array look(0), look(1), look(2) holding the XYZ coordinates of a point on the eyeline.
fov The vertical field-of-view in radians (equivalent to the camera's zoom).
ar The aspect ratio of the 3D graphics window (width/height).
zn The distance from the camera to the near plane (objects nearer than this are invisible).
zf The distance from the camera to the far plane (objects further away than this are invisible).
roll (D3DLIBA and D3D9LIBA only) The camera's roll angle (in radians).

Notes:
1. A Direct3D directional light can be created as follows. Refer to Microsoft documentation for other light types.

```
DIM light%(0) 103
light%(0)!0 = 3 : REM directional light
light%(0)!4 = FN_f4(1) : REM red component
light%(0)!8 = FN_f4(1) : REM green component
light%(0)!12 = FN_f4(0) : REM blue component
light%(0)!64 = FN_f4(0) : REM. X component of direction
light%(0)!68 = FN_f4(0) : REM. Y component of direction
light%(0)!72 = FN_f4(1) : REM. Z component of direction
```

2. A matt Direct3D material can be created as follows. Refer to Microsoft documentation for other material types.

```
DIM mat%(0) 67
mat%(0)!0 = FN_f4(1) : REM red component of colour
mat%(0)!4 = FN_f4(1) : REM green component of colour
```
3. The arrays of object parameters should contain at least as many elements as the value of \texttt{nobj}%. Unused arrays, for example \texttt{mat()} or \texttt{tex()}, should contain zeros (this is the initial state following DIM).

4. Rotations take place around world axes in the order roll then pitch then yaw.
Plotting angled ellipses

The *BBC BASIC for Windows* ELLIPSE statement plots only axis-aligned ellipses. The ELLIPSE library contains the procedures **PROCellipse** and **PROCellipsefill** which provide the facility to plot an outline or filled ellipse rotated by a specified angle. The library should be loaded from your program using the command:

```
INSTALL @lib$+"ELLIPSE"
```

Alternatively, since the procedures are quite short, you might prefer to incorporate them in your own program (use the Insert command from the File menu).

**PROCellipse(x,y,a,b,angle)**

This procedure plots an outline ellipse centred at graphics coordinates `x,y` and with radii of length `a` and `b`. The fifth parameter specifies that the ellipse should be rotated anticlockwise by `angle` radians (if the angle is zero the ellipse is plotted with the `a` axis horizontal, as with the normal ELLIPSE statement).

The ellipse is drawn in the current graphics foreground colour and mode, as specified by `GCOL`.

**PROCellipsefill(x,y,a,b,angle)**

This procedure works in an identical fashion to PROCellipse, except that a filled (solid) ellipse is plotted. Again, the ellipse is drawn in the current graphics foreground colour and mode, as specified by `GCOL`. 
Sorting data arrays

The **SORTLIB** library provides a fast, highly optimised and flexible means of sorting data contained in arrays. SORTLIB will sort byte arrays, integer arrays, floating point arrays and string arrays. The library should be loaded from your program using the command:

```
INSTALL @lib$+"SORTLIB"
```

It contains the single function **FN_sortinit**.

**FN_sortinit(dir%, smode%)**

Before any sorting operations can be carried out the library must be initialised as follows:

```
sort% = FN_sortinit(dir%, smode%)
```

where **dir%** determines the sorting direction (0 = ascending, 1 = descending) and **smode%** determines how strings are sorted:

<table>
<thead>
<tr>
<th>smode%</th>
<th>string sort action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>word sort, case sensitive</td>
</tr>
<tr>
<td>1</td>
<td>word sort, ignore case</td>
</tr>
<tr>
<td>2</td>
<td>ASCII (character code) sort</td>
</tr>
<tr>
<td>&amp;1000</td>
<td>string sort, case sensitive</td>
</tr>
<tr>
<td>&amp;1001</td>
<td>string sort, ignore case</td>
</tr>
</tbody>
</table>

In a **word sort** the hyphen and apostrophe are treated differently from the other nonalphanumeric symbols, in order to ensure that words such as "coop" and "co-op" stay together within a sorted list. In a **string sort**, the hyphen and apostrophe are treated like the other symbols. In an **ASCII sort** the order corresponds to the regular BASIC less-than (<) and greater-than (>) operators.

If you prefer, you can initialise it multiple times with the different options and then
CALL the appropriate variable when needed:

\[
\text{sortascendingnormal} = \text{FN\_sortinit}(0, 0) \\
\text{sortdescendingnormal} = \text{FN\_sortinit}(1, 0) \\
\text{sortascendingignorecase} = \text{FN\_sortinit}(0, 1) \\
\text{sortdescendingignorecase} = \text{FN\_sortinit}(1, 1)
\]

To sort the contents of an entire array do the following:

\[
\text{C} = \text{DIM(array(),DIM(array()))} + 1 \\
\text{CALL sort\%, array(0)}
\]

To sort the contents of a '1-based' array (where the first element has the subscript 1) do the following:

\[
\text{C} = \text{DIM(array(),DIM(array()))} \\
\text{CALL sort\%, array(1)}
\]

To sort only part of an array, set \text{C\%} to the number of elements you want to sort and specify the first element to be sorted:

\[
\text{C\%} = \text{howmany\%} \\
\text{CALL sort\%, array(first\%)}
\]

To sort multiple arrays according to the contents of a key array do the following:

\[
\text{C\%} = \text{size\%} \\
\text{CALL sort\%, keyarray(0), array2$(0), array3%(0),...}
\]

There can be any number of dependent arrays of any type. If the primary key array contains two or more identical elements, the remaining array(s) will be used as secondary keys, in the order specified.

To sort a two-dimensional array, where the contents of the first row are used as a key for the other rows, do the following:

\[
\text{DIM array(2,999)} \\
\text{C\%} = \text{DIM(array(),DIM(array()))} + 1 \\
\text{CALL sort\%, array(0,0), array(1,0), array(2,0)}
\]
Socket (network) connections

The **SOCKLIB** library contains a set of procedures and functions for making \textit{socket} connections to a remote computer via a local area network or the internet, and transferring data in both directions.

The library should be loaded from your program using the command:

\begin{verbatim}
INSTALL @lib$+"SOCKLIB"
\end{verbatim}

The functions contained are:

- PROC_initsockets
- PROC_exitsockets
- FN_gethostname
- FN_tcplisten
- FN_tcpconnect
- FN_check_connection
- FN_writesocket
- FN_writelinesocket
- FN_readsocket
- FN_readlinesocket
- FN_getpeername
- PROC_closesocket
- FN_socketerror

**PROC_initsockets**

**PROC_initsockets** must be called (once) before any other functions in SOCKLIB are accessed.

**PROC_exitsockets**

**PROC_exitsockets** must be called after all socket operations have been completed. You should normally incorporate ON ERROR and ON CLOSE statements in your program to ensure that **PROC_exitsockets** is called even if the program is terminated unexpectedly. If you don't do so your program might not work correctly.
if executed a second time.

**FN_gethostname**

**FN_gethostname** returns, as a string, the network name of your local computer (the one on which *BBC BASIC for Windows* is running).

**FN_tcplisten (host$,port$)**

**FN_tcplisten** creates a *listening socket* to listen for incoming TCP/IP connections. If your program needs to respond to connections made from a remote computer (a web server would be an example) it must call **FN_tcplisten**. The function takes two string parameters: the name of the local computer (typically as returned from **FN_gethostname**) and the port name or number on which you want to listen. For example if you are listening for incoming HTTP (Hyper Text Transport Protocol) connections you would normally set **port$** to "80".

If the listening socket is created successfully the socket number is returned. If the call fails a negative number is returned (see the code of SOCKLIB.BBC for details). You can call **FN_socketerror** to discover more about the error.

**FN_tcpconnect (host$,port$)**

**FN_tcpconnect** makes a TCP/IP connection to a remote machine. You would call this function if your program needs to initiate a connection (a program to send email would be an example). The function takes two string parameters: the name or IP address of the remote computer and the port name or number on which to make the connection. For example if you were connecting to an SMTP mail server would would normally set **port$** to "25" or "mail". If an IP address rather than a machine name is specified, it must be supplied as a string of the form "xx.xx.xx.xx".

If the connection is made successfully the socket number is returned. If the call fails a negative number is returned (see the code of SOCKLIB.BBC for details). You can call **FN_socketerror** to discover more about the error.

**FN_check_connection (socket%)**

**FN_check_connection** checks to see if an incoming connection has been received by a *listening socket*; it takes as a parameter the socket number (as returned from
FN_tceplisten ). If no connection has been received zero is returned, and the listening socket continues to listen for a connection. If an incoming connection has been received, FN_check_connection returns the socket number on which the connection has been opened. Note that if you want to listen for further connections, another listening socket must be opened.

**FN_writesocket (socket%, buffer%, size%)**

**FN_writesocket** writes data to a connected socket. It takes three parameters: the socket number (as returned from FN_tcpconnect or FN_check_connection), the address of a buffer containing the data, and the length of the data in bytes. If the function succeeds it returns the number of bytes sent.

Note that it is possible for **FN_writesocket** to return a value less than the total length of the data. This indicates that only some of the data has been sent, and you should make further calls (adjusting the values of buffer% and size% accordingly) until all the data has been sent.

**FN_writelinesocket (socket%, string$)**

**FN_writelinesocket** writes a string, followed by the characters carriage return (CHR$13) and line feed (CHR$10), to a connected socket. It takes two parameters: the socket number (as returned from FN_tcpconnect or FN_check_connection) and the string to be sent. If the function succeeds it returns zero.

**FN_readsocket (socket%, buffer%, size%)**

**FN_readsocket** reads data from a connected socket. It takes three parameters: the socket number (as returned from FN_tcpconnect or FN_check_connection), the address of a buffer to receive the data, and the maximum length of the data in bytes (the size of the buffer). If the function succeeds it returns the number of data bytes received. If the socket has been disconnected (e.g. by the remote computer) -1 is returned.

**FN_readsocket** does not wait for data to be received. If no data has been received since the last call, it returns zero.

**FN_readlinesocket (socket%, maxtime%, string$)**
**FN_readlinesocket** reads a line from a connected socket (a line is defined as a string of characters terminated by CRLF, LFCR or LF, where CR signifies carriage return and LF signifies line feed); the terminator is not returned as part of the string. It takes three parameters: the socket number (as returned from **FN_tcpconnect** or **FN_check_connection**), a maximum time to wait in centiseconds (e.g. 100 signifies one second) and the name of the string variable in which the line will be returned. If the function succeeds it returns the length of the string in characters. If the socket has been disconnected (e.g. by the remote computer) -1 is returned.

**FN_getpeername (socket%)**

**FN_getpeername** returns the IP address of the remote machine to which a socket is connected, as a string of the form "xx.xx.xx.xx". It takes as a parameter the socket number (as returned from **FN_tcpconnect** or **FN_check_connection**). If the function fails an empty string is returned.

**PROC_closesocket (socket%)**

**PROC_closesocket** closes a socket connection. It takes as a parameter the socket number (as returned from **FN_tcpconnect** or **FN_check_connection**).

**FN_socketerror**

**FN_socketerror** returns the number of the most recent socket error to have occurred (if any). If **FN_tcpconnect** or **FN_teplisten** returns a negative number, you can call **FN_socketerror** to discover more details of the error.

The error code numbers can be found in the Microsoft™ file **WINERROR.H**.
Antialiased graphics

The built-in BBC BASIC graphics statements, and graphics drawn using the Windows™ Graphics Device Interface (GDI), are not antialiased. As a result things like curves and gently-sloping lines can have aliases (or jaggies ) causing the appearance to be impaired.

The GDIPLIB library contains a set of procedures and functions for drawing antialiased graphics. It relies upon the presence of the Microsoft GDI+ graphics subsystem, which is installed as standard only with Windows XP (or later). However it is available for earlier versions of Windows (98 onwards) in the form of a redistributable file which can be downloaded from Microsoft's site (search for GDIPLUS.DLL).

The library should be loaded from your program using the command:

```
INSTALL @lib$+"GDIPLIB"
```

The functions contained are:

- PROC_gdipinit
- PROC_gdipexit
- FN_gdipcreatepen
- FN_gdipcreatebrush
- PROC_gdipdeletepen
- PROC_gdipdeletebrush
- PROC_gdipline
- PROC_gdippolyline
- PROC_gdipbezier
- PROC_gdippolybezier
- PROC_gdiparc
- PROC_gdipsector
- PROC_gdippolygon

**PROC_gdipinit**

**PROC_gdipinit** must be called before any of the other functions in GDIPLIB are used.
PROC_gdipexit

PROC_gdipexit must be called after the GDIPLIB library has been finished with.

FN_gdipcreatepen (colour%, style%, width)

FN_gdipcreatepen is used to create a pen for drawing lines. It takes three parameters: the colour of the pen, the style of the pen and the width of the pen. It returns a value corresponding to the created pen.

The colour should be supplied as an integer equivalent to the hexadecimal value &AARRGGBB where AA is the alpha channel or opacity (00 = transparent, FF = opaque), RR is the red component (00 = no red, FF = maximum red), GG is the green component and BB is the blue component.

The style can be a combination of one or more of the following values:

- LineEndFlat = 0
- LineEndSquare = 1
- LineEndRound = 2
- LineEndTriangle = 3
- LineEndSquareAnchor = &11
- LineEndRoundAnchor = &12
- LineEndDiamond = &13
- LineEndArrow = &14
- LineStartSquare = &100
- LineStartRound = &200
- LineStartTriangle = &300
- LineStartSquareAnchor = &1100
- LineStartRoundAnchor = &1200
- LineStartDiamond = &1300
- LineStartArrow = &1400
- LineDash = &10000
- LineDot = &20000
- LineDashDot = &30000
- LineDashDotDot = &40000

The width is the width of the pen in pixels. Note that this need not be an integer.

FN_gdipcreatebrush (colour%)

FN_gdipcreatebrush is used to create a brush for filling solid shapes. It takes one parameter, the colour of the brush (in the same format as for FN_gdipcreatepen.
above), and returns a value corresponding to the created brush.

**PROC_gdipdeletepen (pen%)**

**PROC_gdipdeletepen** deletes a previously created pen. It takes one parameter: the value returned by FN_gdipcreatepen. Pens must be deleted after use.

**PROC_gdipdeletebrush (brush%)**

**PROC_gdipdeletebrush** deletes a previously created brush. It takes one parameter: the value returned by FN_gdipcreatebrush. Brushes must be deleted after use.

**PROC_gdipline (pen%, x1, y1, x2, y2)**

**PROC_gdipline** draws a straight line between two points. It takes five parameters: the value returned by FN_gdipcreatepen, the x and y coordinates of the start of the line and the x and y coordinates of the end of the line (in BBC BASIC graphics units). Note that the coordinates do **not** need to be integers.

**PROC_gdippolyline (pen%, points%, x(), y())**

**PROC_gdippolyline** draws a set of straight line segments between specified points. It takes four parameters: the value returned by FN_gdipcreatepen, the number of points, an array containing the x-coordinates of the points and an array containing the y-coordinates of the points (in BBC BASIC graphics units). Note that the arrays must **not** be integer arrays, and that the first point corresponds to subscript zero. If you are drawing a closed polygon the number of points should be one more than the number of vertices, with the first and last points coinciding.

**PROC_gdipbezier (pen%, x1, y1, x2, y2, x3, y3, x4, y4)**

**PROC_gdipbezier** draws a Bezier curve. It takes nine parameters: the value returned by FN_gdipcreatepen and the x and y coordinates of four control points (in BBC BASIC graphics units). Note that the coordinates do **not** need to be integers.

**PROC_gdippolybezier (pen%, points%, x(), y())**

**PROC_gdippolybezier** draws a set of Bezier curves. It takes four parameters: the
value returned by \texttt{FN\_gdipcreatepen} , the number of control points ($\geq 4$), an array containing the x-coordinates of the points and an array containing the y-coordinates of the points (in BBC BASIC graphics units). Note that the arrays must \textit{not} be integer arrays, and that the first point corresponds to subscript zero.

You can draw an \textit{angled} ellipse (or at least a very close approximation to it) by using 13 control points, as below. The coordinates $x,y$ are the centre of the ellipse, $a,b$ are the major and minor radii respectively and $t$ is the angle of rotation (radians):

\begin{verbatim}
DEF PROCgdipellipse(pen%, x, y, a, b, t)
LOCAL c, d, o, p, q, r, x(), y()
DIM x(12), y(12)
c=a*SIN(t) : a=a*COS(t)
d=b*SIN(t) : b=b*COS(t)
o=a*0.552285 : q=c*0.552285
p=b*0.552285 : r=d*0.552285
x() = x-a,x-a-r,x-o-d,x-d,x-o-d,x+a-r,x+a,x+a+r,x+o+d,x+d,x-o+d,x-a+r,x-a
y() = y+c,y-p+c,y-b+q,y-b,y-b-q,y-p-c,y-c,y+p-c,y+b-q,y+b,y+b+q,y+p+c,y+c
PROC\_gdippolybezier(pen%, 13, x(), y())
ENDPROC
\end{verbatim}

\textbf{PROC\_gdiparc (pen\%, xc, yc, xr, yr, as, ad)}

\textbf{PROC\_gdiparc} draws an outline (axis-aligned) elliptical arc. It takes seven parameters: the value returned by \texttt{FN\_gdipcreatepen} , the x and y coordinates of the centre of the ellipse (in BBC BASIC graphics units), the x and y radii, the start angle (degrees, clockwise from the x-axis) and the length of the arc (degrees, clockwise).

\textbf{PROC\_gdipsector (brush\%, xc, yc, xr, yr, as, ad)}

\textbf{PROC\_gdipsector} draws a filled (axis-aligned) elliptical sector. It takes seven parameters: the value returned by \texttt{FN\_gdipcreatebrush} , the x and y coordinates of the centre of the ellipse (in BBC BASIC graphics units), the x and y radii, the start angle (degrees, clockwise from the x-axis) and the size of the sector (degrees, clockwise).

\textbf{PROC\_gdippolygon (brush\%, vertices\%, x(), y(), fmode\%)}

\textbf{PROC\_gdippolygon} draws a filled polygon. It takes five parameters: the value returned by \texttt{FN\_gdipcreatebrush} , the number of vertices, an array containing the x-coordinates of the vertices, an array containing the y-coordinates of the vertices (in
BBC BASIC graphics units) and the fill mode. Note that the arrays must not be integer arrays, and that the first vertex corresponds to subscript zero.

The fill mode should be 0 for alternate and 1 for winding. The images below illustrate the effect of the different modes on a five-pointed star:

![Alternate and Winding Modes](image-url)
**COM automation**

The **COMLIB** library contains a set of procedures and functions for controlling applications via the COM (Component Object Model) interface, otherwise known as **COM Automation** or **ActiveX**. This is a method for co-operation and data sharing between applications, standardised by Microsoft, which allows the manipulation of other programs through a defined interface.

The library should be loaded from your program using the command:

```
INSTALL @lib$+"COMLIB"
```

The functions contained are:

- PROC_cominit
- PROC_cominitlclid
- PROC_comexit
- FN_createobject
- FN_getobject
- PROC_releaseobject
- PROC_putvalue
- PROC_callmethod
- FN_getvalueint
- FN_getvaluefloat
- FN_getvaluestr
- FN_getvariant
- FN_coerce

**PROC_cominit**

This procedure initializes the library. It must be called **just once**, typically in the initialisation phase of your program, before you use any of the other functions in COMLIB.

**PROC_cominitlclid (lcid%)**

This procedure works the same as **PROC_cominit** except that it takes as a
parameter a *Locale Identifier*. By specifying an appropriate value you may be able to overcome problems caused by regional variations. For example commands to **Microsoft Office** programs must normally be in the user’s local language, but by specifying the correct Locale Identifier you can force them to accept commands in English.

Some common Locale Identifier values are as follows:

<table>
<thead>
<tr>
<th>LCID</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>1031</td>
<td>German</td>
</tr>
<tr>
<td>1032</td>
<td>Greek</td>
</tr>
<tr>
<td>1033</td>
<td>English</td>
</tr>
<tr>
<td>1034</td>
<td>Spanish</td>
</tr>
<tr>
<td>1036</td>
<td>French</td>
</tr>
<tr>
<td>1037</td>
<td>Hebrew</td>
</tr>
<tr>
<td>1040</td>
<td>Italian</td>
</tr>
<tr>
<td>1043</td>
<td>Dutch</td>
</tr>
</tbody>
</table>

**PROC_comexit**

You should call this procedure **just once** when you have finished with the COM library, typically on exit from your program. It does not free the memory used by objects created with FN_createobject; these must be explicitly released using PROC_releaseobject before calling PROC_comexit.

**FN_createobject (ProgID$)**

This function is the fundamental starting point of a COM operation and returns a reference to a created object. It takes as a parameter either the name of the object or a class ID; for most programs that support COM this will be the "Application" object. These programs contain standalone COM servers. Here are some common ones:
<table>
<thead>
<tr>
<th>Program Name</th>
<th>ProgID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft Access</td>
<td>Access.Application</td>
</tr>
<tr>
<td>Microsoft Excel</td>
<td>Excel.Application</td>
</tr>
<tr>
<td>Microsoft Outlook</td>
<td>Outlook.Application</td>
</tr>
<tr>
<td>Microsoft PowerPoint</td>
<td>PowerPoint.Application</td>
</tr>
<tr>
<td>Microsoft Word</td>
<td>Word.Application</td>
</tr>
<tr>
<td>Internet Explorer</td>
<td>InternetExplorer.Application</td>
</tr>
<tr>
<td>Text to Speech</td>
<td>Sapi.SpVoice</td>
</tr>
</tbody>
</table>

For example you would create an *instance* of the Excel application (assuming it is installed on your PC) as follows:

```plaintext
xlapp% = FN_createobject("Excel.Application")
```

You use the value returned in `xlapp%` for all subsequent references to that object. When you have finished with the object you should call `PROC_releaseobject`.

Note that you may not immediately be able to see Excel (etc.) running, but if you look at the "Processes" tab in Windows Task Manager you will see that it is there.

**FN_getobject (parentref%,"child.grandchild")**

Objects can, and usually do, contain other objects in a hierarchical style; that is the essence of the COM model. If you are familiar with Visual Basic for Applications, the macro language for Microsoft Office, then you will know that Excel contains Workbook objects, Word contains Document objects etc. The hierarchy is denoted by a dot operator. You can obtain an explicit reference to these subordinate objects with `FN_getobject`:

```plaintext
mychild% = FN_getobject(parent%,"name_of_child(1)")
mysheet% = FN_getobject(xlapp%,"Workbooks(1).Worksheet(1)")
myrange% = FN_getobject(mysheet%,"Range(""D1"")")
```

The value returned is a pointer or handle to the object and must be stored for future use. It may refer to a single object or a *collection* of objects. To specify a particular
item in a collection you need to call it by name or its item number in the collection.

The first parameter of **FN_getobject** may be the Application object or some other child object that you already have a handle to; this is the starting point to look up the named object that you give as the second parameter. The problem that you will most likely have is knowing what objects exist! For this you will need the application's documentation of its COM hierarchy, or an Object Viewer. See Object Browser for more details.

**PROC_releaseobject (objref%)**

This procedure is used to release an object's resources (memory etc.) back to Windows:

```vba
PROC_releaseobject(xlapp%)
```

This must be called after the object is no longer required and before your program exits. The parameter is the object reference or handle returned by **FN_createobject**. Objects obtained using **FN_getobject** do not need to be explicitly released as they will be released when their parent object is released. However, you may prefer to release all objects to be on the safe side.

It is important that objects be released whenever your program exits, even unexpectedly as the result of an error or the user clicking Close. You should therefore ensure that appropriate calls to **PROC_releaseobject** are made from your program's ON CLOSE and ON ERROR routines.

**PROC_putvalue (objref%,"Property(value)")**

This procedure writes a value to one of the object's properties. **PROC_putvalue** is one of the prime procedures for manipulating the remote object. You need to define exactly which object you want to access and what property you want to change. See Objects, Methods and Properties for more detail. If the property name does not exist or the value is not of the correct data type, or not in the correct range, the call will fail.

**PROC_putvalue** takes two parameters. Firstly a handle to the object whose property you wish to change, or a handle to a parent object. Secondly a reference to the property and value; this may inclue the path down the object tree:
The value being 'put', such as the Boolean `BTRUE` or the string "Some text" must be enclosed in parentheses. The type or format of the data has to be correct for the call to succeed. Although many applications will accept several data types it is up to the user to ensure that only sensibly typed data reach the application. Packaging the data to a format and syntax that the application will accept can present a real challenge! There are two main aspects:

**Setting the correct data type**

The 'put' data can be in many forms; some are native to *BBC BASIC for Windows* and some are not. The data type is forced by putting a prefix character in front of the value:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Prefix</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long Integer (4 bytes)</td>
<td>None</td>
<td>cell(1,1).value(3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>value(mydata%)</td>
</tr>
<tr>
<td>Short Integer (2 bytes)</td>
<td>S</td>
<td>value(S 525)</td>
</tr>
<tr>
<td>Unsigned Integer (4 bytes)</td>
<td>U</td>
<td>value(U 123456)</td>
</tr>
<tr>
<td>Double Floating Point (8 bytes)</td>
<td>F</td>
<td>value(F 3.66)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cell(1,1).value(F myfloat#)</td>
</tr>
<tr>
<td>Single Floating Point (4 bytes)</td>
<td>G</td>
<td>value(G myfloat)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>value(G 2.67777)</td>
</tr>
<tr>
<td>Boolean (2 bytes)</td>
<td>B</td>
<td>Visible(B TRUE)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Saved(B FALSE)</td>
</tr>
<tr>
<td>Object reference (4 bytes)</td>
<td>O</td>
<td>ChartWizard(O oResizeRange%, Nul, , x1Columns%)</td>
</tr>
<tr>
<td>Null (no data content)</td>
<td>N</td>
<td>BorderAround(Nul, 4, 7)</td>
</tr>
<tr>
<td>String</td>
<td>&quot;</td>
<td>value(&quot;&quot;&quot;&quot;+a$+&quot;&quot;&quot;&quot;)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>worksheet(&quot;&quot;&quot;&quot;Sheet 1&quot;&quot;&quot;&quot;)</td>
</tr>
</tbody>
</table>

If the prefix is omitted COMLIB assumes a floating-point numeric value if a decimal point is present and an integer numeric value if not. This works well with
constants but if a variable name is passed omitting the prefix will cause it to be sent as an integer, potentially resulting in truncation (conversely a structure member will be sent as a floating point value, because of the presence of the dots!).

It should be pointed out that all the prefixes present a restriction in that there could be confusion with variable names that start with capitals B, F, G, N, O, S and U. If for instance you had used variables **North%** and **South%** as parameters then this would fail:

```
PROC_callmethod(objref%,"child.some_method(North%,South%)")
```

The first variable **North%** would be ignored (a Null was assumed) and the second would look for a non-existent variable named **outh%**! This requires the user to choose variable names that do not conflict, in much the same way as you have to avoid variable names starting with BASIC keywords.

Note that in any case the passing of named variables is not compatible with compiling the program with the **Abbreviate names** crunch option, unless you use the **REM!Keep** compiler directive. It may therefore be better to convert the variables' values to strings using `STR$`:

```
PROC_callmethod(objref%,"child.some_method("+STR$North%+","+STR$South%+)")
```

**Formatting string data**

The writing of strings can be confusing. Strings must be enclosed within quotation marks, which you can represent in your program either as `CHR$34` or using the *escape* convention `""`. So to name an Excel worksheet as **My Sheet** you could do either of the following:

```
PROC_putvalue(Xlbook%,"WorkSheets(1).Name(""My Sheet"")")
PROC_putvalue(Xlbook%,"WorkSheets(1).Name(""+CHR$34+"My Sheet""+CHR$34+)"")
```

If the property value itself includes quotation marks they must each be represented in your program as `CHR$34+CHR$34` or `""""`. So for example to set an Excel formula to the value `=A2 &" "& B2` you could do:

```
PROC_putvalue(oRng%,"Formula(""=A2 &"""" ""& B2""")")
```
PROC_callmethod (objref%,"methodname(param1,param2,...)"")

This procedure actions an object *method*. Whenever you need to have the remote application do something you call **PROC_callmethod** with the required parameter list. Methods are things like Add, Open, Quit, Select and Sort; the names of methods are often verbs. Many methods have required parameters and some have optional parameters. To let the application know which parameters you are supplying you must provide the full list in the correct sequence, placing dummy parameters, nulls or commas for the missing optional items up to the point where you have your last parameter (any optional parameters after your last set value can be ignored).

For example **Addshape** is a method which takes one integer and four floating point values:

```
PROC_callmethod(oWB%,"ActiveSheet.Shapes.AddShape(17, 466.5, 40.5, 95.5, 120.1)"
PROC_callmethod(oChart%,"ChartWizard(O oResizeRange%, xl3DColumn%, , xlColumns%)"
```

In this last example we have one parameter that was not defined and was optional. In such cases the application will use its default value. This can also be filled with anything beginning with the capital N such as Null or simply left empty. In each case a null variant is sent to the application. The commas are counted to determine that it was a parameter:

```
PROC_callmethod(myrange%,"BorderAround(Null,4,7)"
```

You may have noticed that the earlier example used 'O oResizeRange%' as a parameter. The prefix O defines the value as an Object or Collections reference. In this case you have passed an object reference as a parameter just as you might pass an array in BBC BASIC.

**FN_getvalueint (objref%,"property.value")**

This function retrieves an integer *property* value from the object. It will return an integer, if at all possible, no matter what type was being held in the application.
Thus if the property is actually a date or a string you will receive no useful information and no warning.

R% = FN_getvalueint(oWB%, "ActiveSheet.cells(8,2).value")
R% = FN_getvalueint(xlapp%, "Visible")

In the first instance the application's internal format may well have been floating point, in which case **FN_getvalueint** will round it to an integer. In the second instance the data type is Boolean but it is translated into integer with the standard values of 0 for **FALSE** and -1 for **TRUE**.

**FN_getvaluefloat (objref%, "property.value")**

This function retrieves a floating point property value:

n = FN_getvaluefloat(oWB%, "ActiveSheet.cells(8,2).value")

If the application's internal data format is not floating point it will be converted to a floating point value if possible.

**FN_getvaluestr (objref%, "property.value")**

This function retrieves a property value as a string, and since nearly all data types can be expressed that way it is very reliable:

a$ = FN_getvaluestr(oWB%, "ActiveSheet.cells(8,2).value")

The string a$ will contain the textual version of the data, be it a number, date, string or a formula.

**FN_getvariant (objref%, "property.value", _ret{})**

This function gets a *variant* type and returns an integer 0 if successful or an error code if not. The function is intended to retrieve nearly any data type, and thus allow you to get to the raw bytes returned in the structure passed by reference. The structure _ret{} is defined as:
The member _ret.vartype% will contain the variant type. The low and high data words are returned in _ret.ldata% and _ret.hdata% respectively. A string describing the variant type is returned in _ret.text$ and the data expressed as a string is returned in _ret.data$. This function is useful if the data type to be returned is unknown. Examination of _ret.vartype% will allow correct interpretation of the resulting data, or allow your program to choose which FN_getvaluexxx to use.

When a string data type is returned you should use the 8-bit ANSI version at _ret.data$. The 16-bit Unicode string, which you might expect to find at the address returned in _ret.ldata% cannot be accessed because COMLIB frees the memory it occupied.

**FN_coerce (var{},vartype%)**

This function takes variant var{} and converts it to the type specified by vartype%. The structure is defined in variant format. Whilst it is mainly for internal use within COMLIB it is of such versatility for type conversion it is documented here. To use outside the COM automation setting you still need to call PROC_cominit before use and PROC_comexit afterwards to allow access to the required Windows library files.

If the conversion succeeds FN_coerce returns zero, otherwise it returns an error code.

**COM hierarchy and the Object Browser**

All Microsoft programs that support automation have a VBA Editor window. This is activated by pressing Alt+F11. With this window open press function key F2 to open the Object Browser. From here you can look up the available COM objects exposed, along with methods, properties and constants. The Help files will take you to the VBA help and to the methods and properties in the COM hierarchy for the application. You will need to understand what the icons mean in the browser window.

**Objects, Methods and Properties**
What is an **Object**? In software terms an object is a piece of portable reusable code that has a defined interface.

An object has two types of thing you can do to it. First it has **Methods** which are actions that do things with it, and secondly it has **Properties** which are the way it looks or what information it contains. Objects can encompass other objects. The overall object, as far as Office automation is concerned, is the Application object. This is a COM server, and your BBC BASIC program acts as the client. This Application object will contain other smaller objects such as workbooks or documents or task lists. These objects have **Collections** of other objects such as words or paragraphs or cells. Finally you get down to the smallest parts that only have methods and properties. Before you can manipulate these methods or change the properties you have to access them.

To access an individual property or method you have to spell out exactly which one you want. Starting from the application object we refer to the property by listing all the steps to get there, for example:

```
Application.workbook(1).worksheet(2).cells(1,3).value(23.6)
```

The Object that you create is called an *instance* of the class of objects. You can run three or five instances of Word.Application. Each one is a separate copy of the software stored in memory with its own workspace. These instances stay until you destroy them.

What is a property and what is a method? **Methods** do things, they take actions and the order in which you activate methods is important. Common methods include Open, Add, Sort, Quit, Save. Methods can create objects, as in the case of Add: Add items, Add documents, Add worksheets. **Properties** change informational states, and it is not important what order you do this in. Properties include things like paragraph settings, font size, font colour, cell values. There are a lot more properties than methods and you will need good documentation or an Object Browser to find them all.

When we have many similar objects we refer to them as **Collections**. There is often a plural in the name. Shapes is a collection of Shape objects. Cells is a collection of range objects. You can refer to the individual Shape by number Shape(1) or it may have a name as in Shape("Triangle"). So how do you know how many there are and what names they have? Well collections always have a property called Count which
is the number in the collection. Then you can use a FOR loop to go through all the Shapes asking for their name:

```vbscript
n% = FN_getvalueint(myobj%,"ActiveDocument.Shapes.Count")
FOR i% = 1 TO n%
    a$ = FN_getvaluestr(myobj%,"ActiveDocument.Shape("+STR$i%+").name")
    PRINT a$
NEXT i%
```

Collections are actually arrays of objects, and like any array you have to make sure you keep track of what is in what element and make sure that you don't go out of range. If you call a method and specify a collection as the object don't be surprised when every member of the collection gets acted upon.

**Variants and variant format**

**Variants** are used in COM processes to allow flexibility of data types. To do this the variant has to contain both the data and some indications of the data type. The structure has a length of 16 bytes. The official definition is 2 bytes to define the data type, 6 bytes undefined and 8 bytes of data, but we can use the following for simplicity:

```vbscript
DIM var{%t%,r%,l%,h%}
```

Here `var.t% AND &FFFF` is the vartype, `var.l%` and `var.h%` are the data bytes. The member `var.r%` is reserved and not used. Integer type data and object references are placed in `var.l%`. Floating point (8 byte) format, currency and date use both `var.l%` and `var.h%`. When a string is contained then a pointer to the string is held in `var.l%`. The string is in **Unicode** (16-bit character) format and cannot be used directly by BBC BASIC.
String manipulation

The STRINGLIB library contains several functions for manipulating or creating character strings. It should be loaded from your program as follows:

```
INSTALL @lib$+"STRINGLIB"
```

The functions contained are:

- FN_lower
- FN_upper
- FN_title
- FN_binary
- FN_tobase
- FN_findreplace
- FN_findreplacei
- FN_instrrr
- FN_instrri
- FN_trim
- FN_split
- FN_join

**FN_lower (A$)**

**FN_lower** converts a string to lower case. Unlike the similar function in the FNUSING library, account is taken of the current language setting.

```
PRINT FN_lower("The Quick Brown Fox")
the quick brown fox
```

**FN_upper (A$)**

**FN_upper** converts a string to upper case (capitals). Unlike the similar function in the FNUSING library, account is taken of the current language setting.
**FN_title (A$)**

**FN_title** converts a string to title case (the first letter of each word is converted to a capital). Account is taken of the current language setting.

```
PRINT FN_title("the quick brown fox")
The Quick Brown Fox
```

**FN_binary (N%)**

**FN_binary** converts the supplied integer parameter to a binary string.

```
PRINT FN_binary(22)
10110
```

**FN_tobase (num%, base%, min%)**

**FN_tobase** converts the integer value num% to a string in number base base% with leading zeroes added (if necessary) so that the total number of characters is at least min%.

```
PRINT FN_tobase(22, 8, 4)
0026
```

**FN_findreplace (text$, old$, new$, start%)**

**FN_findreplace** searches the string text$ for occurrences of the substring old$, and if found replaces them with the string new$, optionally specifying where in the original string the search should start. The parameter text$ is modified, so it must be a variable. The returned value is the number of replacements made.

```
text$ = "The quick brown fox"
num% = FN_findreplace(text$, "brown", "silver", 0)
PRINT text$
```
The quick silver fox

**FN_findreplacei** (text$, old$, new$, start%)  

**FN_findreplacei** works the same as FN_findreplace except that a case-insensitive comparison is made.

```vbnet
text$ = "The quick brown fox"
num% = FN_findreplacei(text$, "BROWN", "silver", 0)
PRINT text$
The quick silver fox
```

**FN_instrr** (A$, B$, S%)  

**FN_instrr** works like the INSTR function except that the position of the last rather than the first match is returned (effectively the search takes place in reverse, starting at the end of the string). The third parameter specifies where to start the search (set it to zero to search the entire string).

```vbnet
PRINT FN_instrr("the quick brown fox", "o", 0)
18
```

**FN_instri** (A$, B$, S%)  

**FN_instri** works like the INSTR function except that a case-insensitive comparison is made.

```vbnet
PRINT FN_instri("the quick brown fox", "O", 0)
13
```

**FN_instrri** (A$, B$, S%)  

**FN_instrri** works like FN_instrr except that a case-insensitive comparison is made.

```vbnet
PRINT FN_instrri("the quick brown fox", "O", 0)
18
```
**FN_trim (A$)**

**FN_trim** removes leading and trailing spaces from the supplied string.

```plaintext
PRINT """" FN_trim(" The quick brown fox ") """
"The quick brown fox"
```

**FN_split (text$, delim$, array$())**

**FN_split** splits a string into parts at the specified delimiter, and stores the parts in the specified array. The value returned is the total number of parts stored in the array (if the string does not contain the specified delimiter, the returned value is 1 and the entire string is copied into element zero of the array). If the array doesn't already exist it is created. If the array exists, but it is not large enough, the original array is discarded (the memory it occupied is leaked) and a new array created.

```plaintext
parts% = FN_split("The quick brown fox", " ", a$())
PRINT a$(1)
quick
```

**FN_join (array$(), delim$, num%)**

**FN_join** reverses the effect of FN_split. It returns a string consisting of the first num% elements of the supplied array (starting at element zero), separated by the supplied delimiter.

```plaintext
parts% = FN_split("The quick brown fox", " ", a$())
PRINT FN_join(a$(), "+", parts%)
The+quick+brown+fox
```
Multiple output windows

The MULTIWIN library provides the capability of having two or more largely independent output windows, each capable of receiving normal BBC BASIC text and graphics output. The library should be loaded from your program as follows (the library calls FN_createwindow in WINLIB5A so that must be installed as well):

```javascript
INSTALL @lib$+"WINLIB5A"
INSTALL @lib$+"MULTIWIN"
```

The functions contained are:

- PROC_multiwin
- FN_createwin
- PROC_selectwin
- PROC_closewin

There are a few minor restrictions in its use:

1. The REFRESH command is disabled.
2. The text cursor (caret) can only be shown in the main window.
3. The output windows share the same colour palette.
4. The @ox% and @oy% system variables cannot be used.

**PROC_multiwin (extra%)**

PROC_multiwin sets up the multiple-window capability and should be called just once during the initialisation phase of your program. It takes one parameter, which is the maximum number of extra output windows required (i.e. in addition to the normal main window).

**FN_createwin (n%, text$, x%, y%, w%, h%, i%, s%, e%)**

FN_createwin creates a new output window. The parameters are as follows: n% is the window number (between 1 and the value specified in the call to PROC_multiwin), text$ is the window title, x% is the horizontal position of the window, y% is the vertical position of the window, w% is the width of the window, h% is the height of the window, i% is an ID number, s% is the window style and
e% is the extended window style.

Normally i% and e% can be set to zero, and s% can be set to &96C00000 which is the numeric equivalent of WS_POPUP + WS_VISIBLE + WS_CLIPSIBLINGS + WS_CLIPCHILDREN + WS_CAPTION. If you prefer the window to be constrained within the bounds of its parent, use WS_CHILD instead of WS_POPUP (&56C00000).

The function returns the handle of the created window.

**PROC_selectwin (win%)**

PROC_selectwin selects a window to be the active output window; it takes one parameter, the window number (either 0, to select the main window, or between 1 and the value specified in the call to PROC_multiwin).

After calling PROC_selectwin all text and graphics output goes to the selected window.

**PROC_closewin (win%)**

PROC_closewin closes one of the extra windows. It takes one parameter, the window number (between 1 and the value specified in the call to PROC_multiwin). Each call to FN_createwin should be paired with a call to PROC_closewin and you should ensure that your program contains appropriate ON ERROR and ON CLOSE statements to guarantee that the additional windows are closed before exit.

You cannot close the currently-selected output window; if necessary precede the statement with a PROC_selectwin(0).
No-wait function replacements

The NOWAIT library provides replacements for the GET/GET$ and INKEY/INKEY$ functions, and for the WAIT, SOUND and INPUT statements which, unlike the native versions, don't block or stall the processing of asynchronous events such as ON SYS and ON TIME interrupts. The library should be loaded from your program as follows:

```
INSTALL @lib$+"NOWAIT"
```

The functions contained are:

- FNget
- FNget$
- FNinkey
- FNinkey$
- PROCwait
- PROCsound
- FNinput

These replacements are particularly useful if your program uses the CALLBACK, SUBCLASS or TIMERLIB libraries, or for other reasons needs to guarantee a timely response to interrupts.

**FNget and FNget$**

FNget and FNget$ directly replace the GET and GET$ functions.

**FNinkey(t%) and FNinkey$(t%)**

FNinkey and FNinkey$ directly replace the INKEY and INKEY$ functions when used with a positive parameter (the maximum time to wait in centiseconds).

**PROCwait (t%)**

PROCwait directly replaces the WAIT statement. It takes a single parameter, being the approximate time to wait in centiseconds.
**PROCsound (c%, a%, p%, d%)**

**PROCsound** directly replaces the SOUND statement. It takes four parameters: channel/effects, amplitude/envelope, pitch and duration.

**FNinput**

**FNinput** provides a partial replacement for the INPUT statement; it returns a string containing the user's response. If you need to display a prompt string, do so using PRINT before calling **FNinput**. If you need to input a numeric value rather than a string, use VAL to perform the conversion.
Callback routines

Some API functions, in Windows® or third-party DLLs, require you to specify a **callback** routine: a subroutine in your program which is called by the external code. The **CALLBACK** library makes it possible to write such routines in BASIC rather than in assembler code. The library should be loaded from your program as follows:

```
INSTALL @lib$+"CALLBACK"
```

The functions contained are:

- FN_syscalls
- FN_syscalln
- FN_callback
- FN_systo
- FN_sysresult

The callback address which you need to pass to the API should be obtained as follows:

```
address% = FN_callback(FNcbroutine(), npar%)
```

Here **FNcbroutine** is the name of the callback function in your program and **npar%** is the number of parameters it takes (this will be specified in the description of the API function).

The callback routine itself needs to be of the following form:

```
DEF FNcbroutine(par1%, par2%, ....)
REM Do whatever is necessary here
= return_value%
```

When using callbacks you must be careful to avoid the use of **blocking** operations in your program, that is functions which are time-consuming or stall interrupt (event) processing, such as INPUT, GET, SOUND and WAIT (with a non-zero parameter). If required the NOWAIT library provides non-blocking replacements.
for these.

Not uncommonly the callback will be made from within an API function itself and in this case it is necessary to replace the conventional form of the SYS statement:

```
SYS "FunctionName", parameters TO result%
```

with the following non-blocking code:

```
SYS FN_syscalls("FunctionName"), parameters TO !FN_systo(result%)
```

Note the exclamation mark! You must call `FN_systo` even if you don't need the value returned by the API function. In that case simply assign the result to a dummy variable. An alternative syntax, compatible with earlier versions of the CALLBACK library, is as follows:

```
SYS FN_syscalls("FunctionName"), parameters
result% = FN_sysresult
```

In the event that you need to call the API function by *address* rather than by *name* use the following code:

```
SYS FN_syscalln(FunctionAddr%), parameters TO !FN_systo(result%)
```

In the event that one of the parameters needs to be a `string`, add an explicit NUL-termination:

```
temp$ = parm$ + CHR$(0)
SYS FN_syscalls("FunctionName"), temp$ TO !FN_systo(result%)
```

For some examples of the use of the CALLBACK library see this Wiki article.
Window subclassing

**Subclassing** is an advanced technique and a detailed explanation is outside the scope of this documentation; for more information see this Microsoft article. The **SUBCLASS** library makes it possible to carry out subclassing in BASIC rather than by writing assembler code. The library should be loaded from your program as follows:

```
INSTALL @lib$+"SUBCLASS"
```

The functions contained are:

- PROC_subclass
- PROC_subclassdlg
- PROC_subclasswin
- PROC_unsubclass

When using the **SUBCLASS** library your program must avoid statements that are time-consuming or which stall interrupt (event) processing, in particular the GET/GET$ and INKEY/INKEY$ functions, and the WAIT, SOUND and INPUT statements. You should use the replacement functions in the NOWAIT library to achieve this. Some Windows messages, for example **WM_NOTIFY**, cannot be subclassed using this technique.

**PROC_subclass (WM_xxxx, FNhandler())**

**PROC_subclass** initiates subclassing of a Windows message. It takes two parameters: **WM_xxxx** is the message number and **FNhandler()** is the name of the function in your program which will handle that message. The handler must be defined as follows:

```
DEF FNhandler(msg%, wparam%, lparam%)
REM Do the necessary message processing here
= return_value%
```

You can subclass multiple messages by calling **PROC_subclass** for each one.
PROC_subclassdlg (dlg%, WM_xxxx, FNhandler())

PROC_subclassdlg is similar to PROC_subclass except that it allows you to subclass messages sent to a dialogue box rather than to the main window. It takes an additional first parameter dlg% which is the value returned from FN_newdialog. You must call PROC_subclassdlg before you call PROC_showdialog.

PROC_subclasswin (hwnd%, WM_xxxx, FNhandler())

PROC_subclasswin is similar to PROC_subclass except that it allows you to subclass messages sent to a window other than the main output window (for example a window created using FN_createwin). It takes an additional first parameter hwnd% which is the handle of the window you want to subclass.

PROC_unsubclass

PROC_unsubclass cancels subclassing of the main window (if multiple messages are being subclassed, they are all cancelled). Optionally you may specify the handle of a window being subclassed, i.e. PROC_unsubclass(hwnd%) . There is no equivalent for dialogue boxes.
High speed timers

The standard ON TIME timers have two disadvantages: firstly their minimum period is 10 milliseconds (a maximum rate of 100 Hz) and secondly they must share the same interrupt service routine. These limitations may be overcome by using the TIMERLIB library. The library should be loaded from your program as follows:

```
INSTALL @lib$+"TIMERLIB"
```

The functions contained are:

- FN_ontimer
- PROC_killtimer

Since the timer interrupts available using this library may occur at rates up to 1000 Hz, the problem of stalling them is particularly severe. To ensure a timely response to the interrupts, and reduce the likelihood of flooding the event queue, you should avoid using any time-consuming statements and functions such as INKEY, INPUT, GET, SOUND or WAIT. If necessary you can make use of the replacement routines in the NOWAIT library.

Also, pay particular attention to the precautions described under Notes on the use of interrupts.

**FN_ontimer (period%, PROCcallback, flags%)**

*FN_ontimer* sets up a timer. It takes three parameters: *period%* is the required period (or delay) in milliseconds, *PROCcallback* is the name of the procedure to be called with the specified periodicity (or after the specified delay) and *flags%* is 0 for a single-shot (delay) timer and 1 for a periodic timer.

The function returns an identifier for use in a subsequent *PROC_killtimer*, or zero if the timer could not be created.

**PROC_killtimer (timerid%)**

*PROC_killtimer* cancels the timer whose identifier is passed as a parameter. It is important to cancel all periodic timers before quitting your program. You should
normally incorporate ON CLOSE and ON ERROR statements to ensure this happens however the program is terminated.
**Parsing XML files**

XML (Extensible Markup Language) files are increasingly commonly used, for example they are the format in which RSS feeds are delivered. The `XMLLIB` library provides a number of functions to simplify the parsing of XML files. The library should be loaded from your program as follows:

```
INSTALL @lib$+"XMLLIB"
```

The functions contained are:

- `PROC_initXML`
- `FN_isTag`
- `FN_nextToken`
- `FN_getLevel`
- `FN_skipTo`
- `FN_skipToRet`
- `PROC_exitXML`
- `FN_repEnt`

**PROC_initXML (xmlobj{}, xmlfile$)**

`PROC_initXML` initialise the parser and opens the file `xmlfile$`, which contains the XML data. It returns the structure `xmlobj{}`, which must be passed to the other library functions as required; `xmlobj{}` need not be (and should not be) initialised prior to calling `PROC_initXML`.

You can concurrently parse multiple XML files by using separate object structures for each file.

**FN_isTag (xmlobj{})**

`FN_isTag` returns TRUE if the next token to be returned by `FN_nextToken` is a tag and FALSE if not. It takes one parameter: the object structure returned by `PROC_initXML`.

**FN_nextToken (xmlobj{})**
FN_nextToken sequentially finds each token in the XML file, and returns a string containing that token. It takes one parameter: the object structure returned by PROC_initXML.

A token can be either a tag or the text contained between two tags; if it is a tag the first character of the returned string will be "< " and the last character will be " > ".

FN_getLevel (xmlobj{})

FN_getLevel returns the current level in the hierarchy (where the base level is 1). It takes one parameter: the object structure returned by PROC_initXML.

At each level of nesting the value returned increases by one. If the returned value is zero it indicates an end-of-file condition (or that PROC_exitXML has been called).

FN_skipTo (xmlobj{}, tagname$, level%)

FN_skipTo scans forwards through the XML file until the specified tag is found, or until the level in the hierarchy is lower than a specified value. It takes three parameters: the object structure returned by PROC_initXML, the name of the tag for which to search and the minimum level. The value returned is the level at which the tag was found, or 0 if the tag was not found within the specified scope.

If the third parameter is set to zero the function will scan the remainder of the file; if the third parameter is set equal to the current level (as returned by FN_getLevel) only the rest of the current 'block' will be searched.

FN_skipToRet (xmlobj{}, tagname$, level%, token$)

FN_skipToRet is similar to FN_skipTo except that it returns, in the string variable specified as the fourth parameter, the token corresponding to the specified tag.

PROC_exitXML (xmlobj{})

PROC_exitXML closes the XML file and cancels the parser. It takes one parameter: the object structure returned by PROC_initXML.

FN_repEnt (text$)
FN_repEnt replaces entity references in the supplied text string; the string is modified (so must be specified as a variable). Entity references are special representations of characters which otherwise have a special meaning in XML (for example <, >, ', " and &).

Note that the returned string may contain Unicode characters encoded as UTF-8; see User-defined modes for how to display such characters.
Extending the assembler

The ASMLIB library provides support for some additional assembler instructions, in particular the CMOV (conditional move) instructions available on the Pentium Pro and later processors, and the SSE (Streaming SIMD Extension) instructions available on the Pentium III and later processors. The library should be loaded from your program as follows:

\[
\text{INSTALL @lib$+"ASMLIB"}
\]

The functions contained are:

- FN_asmext
- FN_cpuid

There are a number of limitations in the use of the ASMLIB library:

1. The additional instructions are accepted only in lower case.
2. There must be only one instruction per line.
3. Labels must be placed on separate lines.
4. Assembler code using the extra instructions cannot be compiled unless all the Crunch options are disabled. As this usually isn't acceptable, you should place the assembler source code in a separate file which you execute using CALL, and give the file an extension other than .BBC to prevent crunching, even if the Crunch embedded program files option is enabled.

**FN_asmext**

**FN_asmext** enables the assembler extensions. It should be used in the line immediately preceding the opening bracket ([ ) of your assembler code, in the following context:

\[
\text{ON ERROR LOCAL [OPT FN_asmext : ] [OPT pass% }} \\
\text{Assembler code starts here...}
\]

You should add a RESTORE ERROR statement after the closing bracket (]) of your assembler code, unless that is shortly followed by an ENDPROC or end-of-function
FN_cpuid (level%, cpuid{})

FN_cpuid allows you to test whether the processor supports the additional instructions; you should always perform such a test before using them. FN_cpuid takes two parameters: a level value specifying what information is required and a structure cpuid{} in which the requested information is returned (the structure should not be initialised prior to calling the function). The function returns FALSE if the CPU does not support the CPUID instruction and TRUE otherwise.

The returned structure contains the members A%, B%, C% and D% corresponding to the values returned in the eax, ebx, ecx and edx registers by the CPUID instruction. To test for the availability of the CMOV (conditional move) instructions use the following code:

```
IF FN_cpuid(1, cpuid{}) IF cpuid.D% AND &8000 THEN
    REM CMOV instructions available
ELSE
    REM CMOV instructions not available
ENDIF
```

To test for the availability of the SSE (Streaming SIMD Extension) instructions use the following code:

```
IF FN_cpuid(1, cpuid{}) IF cpuid.D% AND &2000000 THEN
    REM SSE instructions available
ELSE
    REM SSE instructions not available
ENDIF
```

To discover the Vendor ID string use code similar to the following:

```
IF FN_cpuid(0, cpuid{}) THEN
    PRINT "Vendor ID string: " LEFT$(cpuid{},12)
ELSE
    PRINT "Vendor ID string not available"
ENDIF
```
More assembler extensions

The ASMLIB2 library further extends the assembler, supporting in particular the SSE2 (Streaming SIMD Extension 2) instructions available on the Pentium 4 and later processors. The library should be loaded from your program as follows:

```
INSTALL @lib$+"ASMLIB2"
```

ASMLIB2 is a superset of ASMLIB. It supports all the assembler extensions provided by ASMLIB and contains compatible functions. It is never necessary to INSTALL both ASMLIB and ASMLIB2.

To test for the availability of the SSE2 instructions use the following code:

```
IF FN_cpuid(1, cpuid()) IF cpuid.D% AND &4000000 THEN
    REM SSE2 instructions available
ELSE
    REM SSE2 instructions not available
ENDIF
```
High quality sound patch

The HQSOUND library patches the BBC BASIC for Windows run-time engine, in memory, to cause the SOUND and ENVELOPE statements to generate 'CD quality' stereo sound output (16-bit samples, 44.1 kHz sampling). A processor which supports the SSE2 instruction set is required. The library should be loaded from your program as follows:

```plaintext
INSTALL @lib$+"HQSOUND"
```

If run from the BBC BASIC for Windows IDE the patch persists for the entire session and can only be removed by quitting and restarting. The functions contained are:

- PROC_hqinit
- PROC_stereo
- PROC_voice

**PROC_hqinit (pstereo%, pvoices%, pwaves%)**

PROC_hqinit patches the run-time engine and returns three pointers as follows:

- **pstereo%** is a pointer to eight consecutive 16-bit words in memory which control the stereo mix of the four sound channels. By default channels 0 and 1 are sent to the left output and channels 2 and 3 to the right output, but by storing values (in the range 0 to &8000) in these memory locations the stereo mix can be changed. To set all four sound channels to the centre of the stereo stage you would do:

  ```plaintext
  pstereo%!0  = &40004000 : REM Channels 0 and 1 left
  pstereo%!4  = &40004000 : REM Channels 2 and 3 left
  pstereo%!8  = &40004000 : REM Channels 0 and 1 right
  pstereo%!12 = &40004000 : REM Channels 2 and 3 right
  ```

- **pvoices%** is a pointer to four consecutive bytes in memory which control which of eight 'voices' (waveforms) each of the four sound channels uses. Voice 0 is an approximation to a square wave, voice 1 is a triangular wave, voice 4 is a sine wave and the other five voices are similar to other organ stops. By default all four channels use voice 0 (which is most similar to the 'unpatched' waveform). To set all
four sound channels to use voice 1 you would do:

```plaintext
pvoices%?0 = 1 : REM Channel 0
pvoices%?1 = 1 : REM Channel 1
pvoices%?2 = 1 : REM Channel 2
pvoices%?3 = 1 : REM Channel 3
```

`pwaves%` is a pointer to eight consecutive waveforms in memory, each of which consists of 8,192 signed 16-bit samples (so the allocated memory is &20000 bytes in total). Any of the eight voices may be replaced with a custom waveform by modifying the associated block.

If the default waveforms are acceptable, and PROC_stereo and/or PROC_voice are used to make the selection, PROC_hqinit may be called without any parameters. If no control over the waveforms, stereo position or voice is required the library may be executed using the code below, in which case there is no need to call PROC_hqinit at all and the library will not remain resident in memory after it has patched the executable:

```plaintext
CALL @lib$+'HQ SOUND'
```

### PROC_stereo (channel%, pan%)

PROC_stereo positions one of the four sound channels (0, 1, 2, 3) on the stereo 'stage'. The pan% parameter takes a value from -127 (fully left) to +127 (fully right); zero indicates the centre of the stage. The change happens immediately, even if a note is currently being played on that channel.

### PROC_voice (channel%, voice%)

PROC_voice configures one of the four sound channels (0, 1, 2, 3) to a specific 'voice' or 'waveform' (analogous to an organ stop). Eight voices are available, numbered 0 to 7 (see the description of PROC_hqinit above). The change happens immediately, even if a note is currently being played on that channel.
Introduction to MODE 7

MODE 7 is unique amongst the modes provided by BBC BASIC for Windows, as it was on the original BBC Microcomputer. MODE 7 is a text-only mode (although low-resolution block graphics are available) and has 25 rows each of 40 characters. Each of these characters can be a conventional alphanumeric symbol or a 'control code'; control codes (usually) display as spaces but affect the attributes (e.g. colour) of subsequent characters on the row.

MODE 7 is compatible with the Videotex (Viewdata) and Teletext formats, and uses the same control codes. The only difference is that the 'start box' and 'end box' control codes, which in the case of a Teletext display allow text to be superimposed on a television picture, are not implemented.
Implementation

On the BBC Micro, MODE 7 was implemented in hardware using the SAA5050 chip. PC compatibles have no comparable hardware, so BBC BASIC for Windows emulates the SAA5050 chip in software. With the exception of the 'start box' and 'end box' control codes, all the Videotex display features are implemented, as specified in the Broadcast Teletext Specification (September 1976). These include colour, graphics (contiguous and separated), background colour, flashing, double-height (text and graphics) and held-graphics.
## Character set (MODE 7)

The full Videotex (UK) character set is supported. Those characters not having an equivalent in the ASCII set are obtained from the keys listed below. Note particularly that the code normally corresponding to backslash (and used by Windows™ as a directory delimiter) displays as the fraction $\frac{1}{2}$:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>←</td>
<td>[</td>
</tr>
<tr>
<td>→</td>
<td>]</td>
</tr>
<tr>
<td>↑</td>
<td>^</td>
</tr>
<tr>
<td>$\frac{1}{2}$</td>
<td>\</td>
</tr>
<tr>
<td>$\frac{1}{4}$</td>
<td>{</td>
</tr>
<tr>
<td>$\frac{3}{4}$</td>
<td>}</td>
</tr>
<tr>
<td>-</td>
<td>_ (underscore)</td>
</tr>
<tr>
<td>$\div$</td>
<td>~ (tilde)</td>
</tr>
<tr>
<td>£</td>
<td>` or £</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

As with the BBC Micro, the codes for # (hash) £ (pound) and _ (underscore) are 'shuffled' so as to display as expected, rather than using the Videotex codes. To force use of the standard Videotex codes (such as would be required for displaying a Teletext page) set bit 7 of the stored characters, i.e. add 128 to their normal ASCII values.
Control codes

Control codes can be written to the screen using PRINT CHR$ or VDU in the usual way, but an alternative is to incorporate them in a constant (quoted) string. To enter a control code into a constant string hold down the Alt key and type a four digit decimal value on the numeric keypad. For example, to enter the code for yellow text hold down Alt, type 0131 on the numeric keypad, then release Alt.

So to write some yellow text you could do any of the following:

```
MODE 7
PRINT CHR$131;"Yellow text"
VDU 131 : PRINT "Yellow text"
PRINT "•Yellow text"
```

where the symbol • corresponds to that obtained by entering Alt-0-1-3-1.

Normally control codes display as a space (in the current background colour) but see the Held graphics section for exceptions. Control codes affect subsequent characters in the row; all rows start off as steady, single-height alphanumeric white characters on a black background.

Control codes summary

<table>
<thead>
<tr>
<th>Code</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>129</td>
<td>alphanumeric red</td>
</tr>
<tr>
<td>130</td>
<td>alphanumeric green</td>
</tr>
<tr>
<td>131</td>
<td>alphanumeric yellow</td>
</tr>
<tr>
<td>132</td>
<td>alphanumeric blue</td>
</tr>
<tr>
<td>133</td>
<td>alphanumeric magenta</td>
</tr>
<tr>
<td>134</td>
<td>alphanumeric cyan</td>
</tr>
<tr>
<td>135</td>
<td>alphanumeric white</td>
</tr>
<tr>
<td>136</td>
<td>flash</td>
</tr>
<tr>
<td>137</td>
<td>steady</td>
</tr>
<tr>
<td>140</td>
<td>normal height</td>
</tr>
<tr>
<td>141</td>
<td>double height</td>
</tr>
</tbody>
</table>
graphics red
graphics green
graphics yellow
graphics blue
graphics magenta
graphics cyan
graphics white
conceal
contiguous graphics
separated graphics
black background
new background
hold graphics
release graphics
Coloured text (MODE 7)

By default text displays as white characters on a black background. You can change the text colour by using one of the following control codes:

- 129 red text
- 130 green text
- 131 yellow text
- 132 blue text
- 133 magenta text
- 134 cyan text
- 135 white text

MODE 7
PRINT CHR$130;"Green text"
Background colour (MODE 7)

The background colour is normally black. To change the background colour you must first select the required colour as the current text (or graphics) colour, then insert the new background control code 157.

For example to select a blue background you can insert code 132 followed by code 157 (which will together occupy two consecutive character positions on the screen). Remember to change the text colour back to something different, since blue text on a blue background isn't very useful!

```
MODE 7
PRINT CHR$132;CHR$157;CHR$131"Yellow text on blue background"
```

To switch back to a black background insert control code 156.
Block graphics

The graphics capability in MODE 7 is limited to low-resolution block graphics, where each character cell is occupied by six small blocks in an arrangement of two columns by three rows. For obvious reasons these blocks are sometimes referred to as *sixels*. To select a graphics mode insert one of the following control codes:

- 145 red graphics
- 146 green graphics
- 147 yellow graphics
- 148 blue graphics
- 149 magenta graphics
- 150 cyan graphics
- 151 white graphics

Once a graphics mode is selected each of the 64 possible patterns of *sixels* corresponds to an alphanumeric character code. The following table shows the correspondence between each code and the displayed graphic:

- 160
- 161
- 162
- 163
- 164
- 165
- 166
- 167
- 168
- 169
- 170
- 171
- 172
- 173
- 174
- 175
- 176
- 177
- 178
- 179
- 180
- 181
- 182
- 183
- 184
- 185
- 186
- 187
- 188
- 189
- 190
- 191
- 224
- 225
- 226
- 227
- 228
- 229
- 230
- 231
- 232
- 233
- 234
- 235
- 236
- 237
- 238
- 239
- 240
- 241
- 242
- 243
- 244
- 245
- 246
- 247
- 248
- 249
- 250
- 251
- 252
- 253
- 254
- 255

As with control codes, these character codes can be entered with PRINT CHR$, VDU or by holding down Alt and typing a four digit decimal value on the numeric keypad.

MODE 7
PRINT "A red box:"CHR$145;CHR$247;CHR$251
**Blast-through text**

Capital letters display as text *even when in a graphics mode*. This is handy as it allows you to incorporate text without the overhead (and wasted space) of switching from graphics mode to text mode and back again. This is called blast-through text.

**Separated graphics**

Normally graphics characters display as contiguous blocks as shown in the table above (in other words, there are no gaps between the sixels). By inserting control code 154 you can switch to the separated graphics mode; in this mode the graphics blocks are smaller and are separated by small gaps. This can give a shaded appearance and improve the subjective quality of certain types of graphics.

To switch back to contiguous graphics on the same row, insert the control code 153.

**Held graphics**

Generally all control codes are displayed as spaces, in the current background colour. In the held graphics mode, which can be selected by inserting control code 158, control codes are displayed as a copy of the most recently displayed graphics symbol. This permits a limited range of abrupt display colour changes.

The held graphics character is displayed in the same contiguous/separated mode as when it was first displayed. If there has been a change in the text/graphics mode or the normal/double-height mode since the last graphics character was displayed, the held graphics character is cleared and control codes once again display as spaces.

To switch held graphics mode off, on the same row, insert control code 159.
Flashing

Text or graphics may be made to flash by inserting control code 136; the characters alternate between the selected colour and the current background colour. To switch back to steady characters, on the same row, insert control code 137.

All rows begin as steady, single-height, white alphanumeric characters on a black background.

MODE 7
PRINT CHR$136"Some flashing text"
Double height

Text or graphics may be displayed in double-height, where one row contains the top halves of the characters and the next row contains the bottom halves. To select double-height, insert control code 141; to switch back to single-height, on the same row, insert control code 140. Wherever you have single-height characters the second row of the pair will contain spaces (in the current background colour).

The characters in the first row of a double-height pair are not automatically duplicated in the second row (as would be the case for a conventional hardware implementation) so you must arrange for two consecutive display rows to contain identical data. The BBC Micro had the same requirement and it is what BBC BASIC programs expect. It has the advantage of allowing special effects such as the top and bottom halves of double-height characters being in different colours. Software drivers for Viewdata or Teletext emulations must take account of this feature and carry out the duplication themselves.

There is no need to, nor can you, specify which row should contain the top halves of the characters and which the bottom; this is worked out automatically by the MODE 7 display driver. The consequence of this is that if you delete all the double-height control codes (141) in the first row of the pair, the characters in the 'second' row will instantly change from bottom halves to top halves.

```
MODE 7
PRINT CHR$141"Double-height text"
PRINT CHR$141"Double-height text"
```
Reveal/conceal

The **conceal** control code (152) causes subsequent text in the row to be hidden (displayed as spaces in the current background colour) until the next text colour or graphics colour control code is encountered. To reveal the hidden text, an applications program must change all the **conceal** characters to something innocuous (e.g. the **escape** character, control code 155, which does nothing).

The two procedures below are taken from the demonstration program 'MODE7DEM.BBC'. The first procedure converts all **conceal** characters (&98) to **escape** characters (&9B). The second converts all **escape** characters back to **conceal** characters.

```basic
DEF PROC reveal
LOCAL N%,X%,Y%
X%=POS:Y%=VPOS:VDU 30
FOR N%=1 TO 24*40
  IF (&FF00 AND USR&FFF4)DIV256=&98 VDU &9B ELSE VDU 9
NEXT
PRINTTAB(X%,Y%);
ENDPROC

DEF PROC conceal
LOCAL N%,X%,Y%
X%=POS:Y%=VPOS:VDU 30
FOR N%=1 TO 24*40
  IF (&FF00 AND USR&FFF4)DIV256=&9B VDU &98 ELSE VDU 9
NEXT
PRINTTAB(X%,Y%);
ENDPROC
```
Trapped OPENIN, OPENOUT or OPENUP

If you pass the OPENIN, OPENOUT or OPENUP functions a null filename (""”) or a wildcard filename ("*.dat") BBC BASIC for Windows displays a file selector window. This is a handy way of allowing the user to choose a file for reading or writing. However there is a small snag: if the user clicks on the Cancel button BASIC behaves as if the Escape key was pressed (even if the *ESC OFF command was issued). This can be undesirable, as Escape results in an error being generated. A simple solution to this problem is to use the following functions, which instead return -1 in the event of Cancel being pressed:

DEF FNopenin(file$)
ON ERROR LOCAL = -1
= OPENIN(file$)
DEF FNopenout(file$)
ON ERROR LOCAL = -1
= OPENOUT(file$)
DEF FNopenup(file$)
ON ERROR LOCAL = -1
= OPENUP(file$)
Using windows larger than 1920 x 1440 pixels

The default output bitmap used by *BBC BASIC for Windows* is 1920 x 1440 pixels. Whilst this should be large enough for the majority of display settings and applications you may occasionally want to increase its size. For example you might want to extend the technique used in the example program SCROLL.BBC to scroll over an even larger 'canvas'. Alternatively you might want to create an output bitmap much larger than your screen to improve the quality of graphics printed using *HARDCOPY*. The procedure below allows you to do that:

```basic
DEF PROChugewindow(x%, y%) : LOCAL bmih{}, bits%, hbm%, oldbm%
DIM bmih(Size%, Width%, Height%, Planes{l&,h&}, BitCount{l&,h&}, \
    Compression%, SizeImage%, XPelsPerMeter%, YPelsPerMeter%, \
    ClrUsed%, ClrImportant%)
bmih.Size% = DIM(bmih{})
bmih.Width% = x%
bmih.Height% = y%
bmih.Planes.l& = 1
bmih.BitCount.l& = 24
SYS "CreateDIBSection", @memhdc%, bmih{}, 0, ^bits%, 0, 0 TO hbm%
IF hbm% = 0 ERROR 100, "Couldn't create DIBSection"
SYS "SelectObject", @memhdc%, hbm% TO oldbm%
SYS "DeleteObject", oldbm%
SYS "SetWindowPos", @hwnd%, 0, 0, x%, y%, 6
@vdu%!208 = x%
@vdu%!212 = y%
VDU 24,0;0;x%*2-2;y%*2-2;
CLG
ENDPROC
```

Note that increasing the size of the text viewport beyond 1920 x 1440 pixels (e.g. using VDU 26 or VDU 28) is liable to crash BASIC. The routine above deliberately leaves the text viewport unchanged.

Note also that the above routine does not support 'palette animation'.
TrackPopupMenu

If you've tried using SYS "TrackPopupMenu" to display a 'floating' (context) menu you will have found it doesn't work! This is because it is one of the few Windows™ API calls which must be made from a thread with a message loop (mind you, nowhere does Microsoft bother to tell you that!). To solve this problem use the following function instead:

```
DEF FNtrackpopupmenu(hmenu%,flags%,x%,y%)
LOCAL M%, O%, P%, T%
DIM P% LOCAL 60
[OPT 2
 .T% push 0
 push @hwnd%
 push 0
 push y%
 push x%
 push flags%
 push hmenu%
 call "TrackPopupMenu"
 ret 16
 .M% cmp dword [esp+8],&500 : jz T%
 pop eax : push [^O%] : push eax
 jmp "CallWindowProc"
]
SYS "GetWindowLong", @hwnd%, -4 TO O%
SYS "SetWindowLong", @hwnd%, -4, M%
SYS "SendMessage", @hwnd%, &500, 0, 0 TO T%
SYS "SetWindowLong", @hwnd%, -4, O%
SYS "SendMessage", @hwnd%, 0, 0, 0
= T%
```
Passing floats to assembler code or DLLs

If you need to speed up floating point (real number) calculations you may want to do some of the work in assembly language. The BBC BASIC for Windows assembler makes this relatively easy by incorporating all the floating point instructions but you have to be careful when passing floating point numbers from BASIC to the assembler code. Firstly you must have selected the *FLOAT 64 mode (only then are BASIC’s numbers in a format which the processor can understand directly) and secondly you need to be aware that BASIC stores integers in a special way. To solve this latter problem you should multiply all values by 1.0 (the decimal point is important) before passing them to the assembler code:

```basic
fpvalue *= 1.0
CALL code, fpvalue
```

If you don't do that your assembler code will do completely the wrong thing in the event that `fpvalue` contains an integer. Exactly the same applies if you are passing a floating point value to a function in a DLL (which will typically require a pointer to the value):

```basic
fpvalue *= 1.0
SYS dllfunction%, ^fpvalue
```

In the event that you need to pass an array of floating point values to a DLL function each element of the array must be multiplied by 1.0 for the same reason (and the address passed must be that of the first element of the array):

```basic
fparray() *= 1.0
SYS fft%, ^fparray(0)
```

Note that the assembler code or DLL must be expecting double precision (64-bit) floating-point numbers. BBC BASIC for Windows has no built-in support for ordinary 32-bit floats (but see the FN_f4 library routine for a conversion function).
Precautions when using LOCAL arrays

LOCAL arrays are stored on the stack and special precautions are required as a result. Firstly, avoid swapping a local array with a global array (i.e. one stored on the heap). Because SWAP exchanges the array pointers (rather than the array data) the 'global' array may end up pointing to data on the stack, which will become invalid on exit from the function or procedure in which the array was defined. Any subsequent attempt to access the array data will fail, and may crash BASIC. You can safely copy the array, because the data is copied rather than the pointer:

```basic
DEF PROChint
LOCAL localarray()
DIM localarray(10)
SWAP globalarray(),localarray() : REM Don't do this!
globalarray() = localarray() : REM This is OK
ENDPROC
```

Secondly, be careful if your program uses LOCAL arrays and ON ERROR. If an error occurs in a procedure or function in which a local array is defined, BASIC will jump immediately to the ON ERROR routine without passing through the ENDPROC or end-of-function statement. The 'local' array will still exist, but will point to an invalid area of memory (errors cause the stack to be discarded), so again any subsequent attempt to access the contents of the array will fail and may cause a crash. To protect against this either ensure errors (even Escape) cannot occur in those functions or procedures, use ON ERROR LOCAL and RESTORE LOCAL to clear the local array(s), or make sure all your local arrays have different names from any global arrays:

```basic
DIM temp(10) : REM Global array
DEF PROChint
LOCAL temp() : REM Don't do this if ON ERROR is active
ENDPROC
```

Simply changing the name of the global array from temp to (for example) Temp would avoid any problems in the event of an error occurring inside the procedure.
Using status bars and toolbars with the Multiple Document Interface

The WINLIB library allows you to incorporate status bars and toolbars in your program, and the MDILIB library allows you to use the Multiple Document Interface. However if you try to do both at the same time there are problems, because the MDI window covers up the toolbar and the status bar. The solution is to change the size and position of the MDI window so that the toolbar and/or status bar are not covered; this can be done by adding the following code immediately after the call to PROC_initmdi() :

For a toolbar and status bar :

```plaintext
DIM rc{l%,t%,r%,b%}
SYS "GetWindowRect", hstat%, rc{}
sbh% = rc.b%-rc.t%
SYS "GetWindowRect", htool%, rc{}
tbh% = rc.b%-rc.t%
SYS "GetClientRect", @hwnd%, rc{}
SYS "MoveWindow", @hmdi%, 0, sbh%, rc.r%, rc.b%-sbh%-tbh%, 1
```

For just a toolbar :

```plaintext
DIM rc{l%,t%,r%,b%}
SYS "GetWindowRect", htool%, rc{}
tbh% = rc.b%-rc.t%
SYS "GetClientRect", @hwnd%, rc{}
SYS "MoveWindow", @hmdi%, 0, tbh%, rc.r%, rc.b%-tbh%, 1
```

For just a status bar :

```plaintext
DIM rc{l%,t%,r%,b%}
SYS "GetWindowRect", hstat%, rc{}
sbh% = rc.b%-rc.t%
SYS "GetClientRect", @hwnd%, rc{}
SYS "MoveWindow", @hmdi%, 0, 0, sbh%, rc.r%, rc.b%-sbh%, 1
```

Here hstat% is the handle of the status bar and htool% is the handle of the toolbar.

In order that the toolbar and status bar remain visible after the user resizes the window, the same code should be duplicated in your ON MOVE routine, for example:

```plaintext
hwndorig% = @hwnd%
```
DIM Move%(2)
ON MOVE Move%()=@msg%,@wparam%,@lparam%:PROCmove:RETURN
...
DEF PROCmove
LOCAL rc{},sbh%,tbh%
DIM rc{l%,t%,r%,b%}
SYS "GetWindowRect", hstat% , rc{}
sbh% = rc.b%-rc.t%
SYS "GetWindowRect", htool% , rc{}
tbh% = rc.b%-rc.t%
SYS "PostMessage", hstat%, Move%(0), Move%(1), Move%(2)
SYS "PostMessage", htool%, Move%(0), Move%(1), Move%(2)
SYS "GetClientRect", hwndorig%, rc{}
SYS "MoveWindow", @hmdi%, 0, tbh%, rc.r%, rc.b%-tbh%-sbh%, 1
ENDPROC

Here hstat% is the handle of the status bar and htool% is the handle of the toolbar. Note that PROCmove uses a copy of the original @hwnd% because programs using the Multiple Document Interface often modify @hwnd% during the course of their execution.
Foreign language equivalent of TIME$

The built-in TIME$ function always returns the day and month names in English. The function below returns a string identical in format to TIME$ but with the day and month names in the language for which the PC has been configured:

```
DEF FNtime$ : LOCAL D%, L%
DIM D% LOCAL 255
SYS "GetDateFormat", 0, 0, 0, "ddd.dd MMM yyy", D%, 255 TO L%
SYS "GetTimeFormat", 0, 0, 0, "HH:mm:ss", D%+L%-1, 255-L%
= $$D%
```
Loading a sprite (or icon) from memory

The FN_createsprite routine in the SPRITELIB library creates a sprite from an icon (.ICO) file. There may be situations where you would prefer to load the file into memory first (perhaps containing many different icons or other data) and create the sprite from the memory contents. You can do that using the routine below, which takes a memory pointer rather than a filename but is otherwise used in exactly the same way as FN_createsprite:

```basic
DEF FN_createspritefrommemory(N%, P%, W%, H%) : LOCAL S%
   IF N% >= `sprites%!0 THEN = FALSE
   S% = `sprites% + N%*24 + 32
   SYS "CreateIconFromResourceEx", P%+P%!18, P%!14, 1, &30000, W%, H%, 0 TO !S%
   S%!4 = W%
   S%!8 = H%
   = !S%
```

The second parameter must be the address in memory at which an icon file is loaded.
Setting the File Open dialogue's initial directory

The File Open and File Save dialogue boxes remember the directory (folder) which was last viewed, and by default select it as the initial directory on the next occasion. This is generally a useful feature, although it only works properly when your BASIC program has been compiled to a stand-alone executable (the 'remembered' directory can be changed to something different when running under the interactive environment). However there may be occasions when you want to override this behaviour and determine the initial directory yourself. You can do that by using the following code instead of that listed in the manual:

```basic
DIM fs{1StructSize%, hwndOwner%, hInstance%, lpstrFilter%, \lpstrCustomFilter%, nMaxCustFilter%, nFilterIndex%, \lpstrFile%, nMaxFile%, lpstrFileName%, \nMaxFileTitle%, lpstrInitialDir%, lpstrTitle%, \flags%, nFileOffset{l&,h&}, nFileExtension{l&,h&}, \lpstrDefExt%, lpfnHook%, lpTemplateName%}
DIM fp% 255
InitialDir$ = "C:\bbcbasic"+CHR$0
FileFilter$ = "BMP files"+CHR$0+"*.BMP"+CHR$0+CHR$0
fs.lStructSize% = DIM(fs{})
fs hwndOwner% = @hwnd%
fs.lpstrFilter% = !^FileFilter$
fs.lpstrFile% = fp%
fs.nMaxFile% = 256
fs.lpstrInitialDir% = !^InitialDir$
fs.flags% = 6
```

Note that the string containing the initial directory must be terminated with a NUL (CHR$0). Once you have executed this code you can call up the File Open or File Save dialogue with SYS "GetOpenFileName" or SYS "GetSaveFileName" in the normal way. The usual precautions for the use of DIM apply (make sure you do it only once, or use DIM LOCAL).
Centering a hardcopy printout on the page

The manual tells you how to centre printed text on a particular column, but not how to centre it on the page. The following routine prints (on the printer) a string centred between the left and right margins:

```
DIM size(cx%,cy%)
*PRINTERFONT Arial,24
*MARGINS 10,10,10,10
VDU 2,1,32
string$ = "The quick brown fox"
SYS "GetTextExtentPoint32", @prthdc%, string$, LEN(string$), size()
@vdu%!-12 = (@vdu%!232 + @vdu%!236 - size.cx%)/2
PRINT string$
VDU 1,12,3
```

The VDU 1,32 is only needed if nothing else has previously been printed on the page. For multiple lines of text simply repeat the four statements commencing `string$ =` and ending `PRINT string$`. 
Using DATA in installed modules

Because CALL ed and INSTALL ed modules cannot use line numbers, the traditional form of the RESTORE statement cannot be used to move the data pointer into such modules. However the relative form RESTORE +n can. A convenient way of arranging this is to associate a restore procedure with each independent block of data you might want to use:

```
DEF PROC restore1 : RESTORE +1 : ENDPROC
DATA 1, 2, 3, 4, 5, 6, 7, etc
DATA ...
DEF PROC restore2 : RESTORE +1 : ENDPROC
DATA 8, 9, 10, 11, 12, 13, etc
DATA ...
```

Then, when you want to READ the data (which can be done from the main program, or another installed module) you just call the appropriate restore procedure first:

```
PROC restore1
READ A, B, C, D, E, F, G, etc
PROC restore2
READ a, b, c, d, e, f, g, etc
```

Using this technique you can hide your data away in a separate module.
Re-entrant ON MOVE interrupts

The manual gives the following example of the use of ON MOVE:

```
ON MOVE PROCmove(@msg%,@lparam%) : RETURN
```

This will work, and guarantees that the PROCmove procedure will be called once for each ON MOVE event, but there is a snag: because the procedure can itself be interrupted by a subsequent ON MOVE, the events may not be processed in the order in which they arrive! It may well be more important to ensure that the last ON MOVE is processed last, even if it means that some earlier ones are discarded. You can achieve this behaviour by passing the parameters in a global array. For example, to *forward* the ON MOVE event to a toolbar and status bar so they resize themselves correctly:

```
DIM Move%(2)
ON MOVE Move%()=@msg%,@wparam%,@lparam% : PROCmove : RETURN
......
DEF PROCmove
SYS "SendMessage", hToolbar%, Move%(0), Move%(1), Move%(2)
SYS "SendMessage", hStatbar%, Move%(0), Move%(1), Move%(2)
ENDPROC
```

It is important that the three elements of the `Move%(0)` array are used in the same statement, so another interrupt cannot occur between them.
Cunning use of the CASE statement

Testing for several mutually-exclusive possibilities using nested IF...ENDIF statements can be messy:

```。
IF abc%<10 THEN
  state% = 1
ELSE IF abc%=10 THEN
  state% = 2
ELSE IF abc%>20 THEN
  state% = 3
ELSE
  state% = 99
ENDIF
ENDIF
ENDIF
```

The unhelpful indentation and the multiple ENDIFs make it unclear what is happening. The same thing can be achieved more elegantly by using the CASE statement in a cunning way:

```。
CASE TRUE OF
  WHEN abc%<10: state%=1
  WHEN abc%=10: state%=2
  WHEN abc%>20: state%=3
  OTHERWISE: state%=99
ENDCASE
```

Notice the use of CASE TRUE to force the interpreter to test the truth of each of the conditional expressions.
Docking' a dialogue box

On occasions you may want your program's user interface to consist solely of a dialogue box, for example if all input/output is by means of buttons, list boxes etc. The easiest way to achieve that is to hide your program's main output window (using a SYS "ShowWindow" command or by selecting the initial window state as hidden when you compile it).

However there is a disadvantage in this approach: the user cannot 'minimise' your program's window because dialogue boxes don't have a minimise button. Even if you add a minimise button, the window won't minimise to an icon on the task bar as a conventional application would.

A solution is to 'dock' the dialogue box to the main window so that it appears and behaves as a dialogue box but can be minimised just like an ordinary window. This requires three steps:

- Resize the main output window so it is exactly the same size as the dialogue box.
- Remove the dialogue box's title bar.
- Make the dialogue box a child window so it moves with the main window.

The program segment below achieves all three effects:

```plaintext
dlg% = FN_newdialog("", 0, 0, width%, height%, font%, size%)
dlg%!16 = (dlg%!16 OR &40000000) AND NOT &80400000
REM Add dialogue box contents here in the usual way
PROC_showdialog(dlg%)
DIM rc{l%, t%, r%, b%}
SYS "GetWindowRect", !dlg%, rc{}
SYS "GetWindowLong", @hwnd%, -16 TO style%
SYS "SetWindowLong", @hwnd%, -16, style% AND NOT &50000
SYS "AdjustWindowRect", rc{}, style% AND NOT &50000, 0
SYS "SetWindowPos", @hwnd%, 0, 0, 0, rc.r%-rc.l%, rc.b%-rc.t%, 102
```

This code should be executed just once at the beginning of your program. Note that the second and third parameters of FN_newdialog must be zero; if you create your dialogue box template using DLGEDIT remember to change them.
You can use a similar technique to 'dock' a **Property Sheet** or a **Wizard**. In those cases add the following code *after* the PROC_showpropsheet statement:

```plaintext
DIM rc{l%,t%,r%,b%}
SYS "GetClientRect", !psh%, rc{}
SYS "SetParent", !psh%, @hwnd%
SYS "GetWindowLong", !psh%, -16 TO style%
SYS "SetWindowLong", !psh%, -16, (style% OR &40000000) AND NOT &80400000
SYS "SetWindowPos", !psh%, 0, 0, 0, 0, 0, 37
SYS "GetWindowLong", @hwnd%, -16 TO style%
SYS "SetWindowLong", @hwnd%, -16, style% AND NOT &50000
SYS "AdjustWindowRect", rc{}, style%, 0
SYS "SetWindowPos", @hwnd%, 0, 0, 0, rc.r%-rc.l%, rc.b%-rc.t%, 102
```

Here **psh%** is the value returned from FN_newpropsheet.
Disabling Windows XP Visual Styles

*BBC BASIC for Windows* by default uses Windows XP™ Visual Styles. If for some reason you don't want to use the new-style controls in your program, or you simply don't like their appearance, you have three options:

- You can disable Windows XP Visual Styles altogether by selecting 'Display Properties... Appearance... Windows and buttons... Windows Classic style' (but this will affect all programs, not just BBC BASIC).

- You can disable Windows XP Visual Styles for just one application by right-clicking on its shortcut, selecting Properties... Compatibility and checking the 'Disable visual themes' box.

- You can compile your program to an executable and **not** select the 'Use Windows XP visual styles' option when you do so. The executable will then run without Windows XP visual styles (however you should **not** do this if you want the program to run in *Windows Vista™* or later, because enabling XP visual styles also activates the important UAC and compatibility **manifest**).

- You can disable Windows XP visual styles for a specific control using the following procedure, where the parameter is the control's window handle:

  ```basic
  DEF PROC_nostyle(hw%)
  LOCAL uxt%, swt%
  SYS "LoadLibrary", "uxtheme.dll" TO uxt%
  SYS "GetProcAddress", uxt%, "SetWindowTheme" TO swt%
  IF swt% SYS swt%, hw%, CHR$0, CHR$0
  SYS "FreeLibrary", uxt%
  ENDPROC
  ```
## INDEX

<table>
<thead>
<tr>
<th>'Docking' a dialogue box</th>
<th>978</th>
</tr>
</thead>
<tbody>
<tr>
<td>! operator, The</td>
<td>116</td>
</tr>
<tr>
<td>$ operator, The</td>
<td>116</td>
</tr>
<tr>
<td>$$ operator, The</td>
<td>116</td>
</tr>
<tr>
<td>*</td>
<td>541</td>
</tr>
<tr>
<td>*BYE</td>
<td>488</td>
</tr>
<tr>
<td>*CHDIR (*CD)</td>
<td>489</td>
</tr>
<tr>
<td>*COPY</td>
<td>490</td>
</tr>
<tr>
<td>*DELETE (*DEL)</td>
<td>492</td>
</tr>
<tr>
<td>*DIR</td>
<td>493</td>
</tr>
<tr>
<td>*DISPLAY</td>
<td>494</td>
</tr>
<tr>
<td>*EGA</td>
<td>496</td>
</tr>
<tr>
<td>*ERASE (*ERA)</td>
<td>497</td>
</tr>
<tr>
<td>*ESC</td>
<td>498</td>
</tr>
<tr>
<td>*EXEC</td>
<td>499</td>
</tr>
<tr>
<td>*FLOAT</td>
<td>500</td>
</tr>
<tr>
<td>*FONT</td>
<td>502</td>
</tr>
<tr>
<td>*FX</td>
<td>504</td>
</tr>
<tr>
<td>*GSAVE</td>
<td>505</td>
</tr>
<tr>
<td>*HARDCOPY</td>
<td>506</td>
</tr>
<tr>
<td>*HELP</td>
<td>507</td>
</tr>
<tr>
<td>*HEX</td>
<td>508</td>
</tr>
<tr>
<td>*INPUT</td>
<td>509</td>
</tr>
<tr>
<td>*KEY</td>
<td>510</td>
</tr>
</tbody>
</table>
*LIST 512
*LOAD 513
*LOCK 514
*LOWERCASE 515
*MARGINS 516
*MDISPLAY 518
*MKDIR (*MD) 517
*NOEGA 520
*OUTPUT (*OPT) 521
*PLAY 522
*PRINTER 523
*PRINTERFONT 524
*QUIT 525
*REFRESH 526
*RENAME (*REN) 527
*RMDIR (*RD) 528
*RUN 529
*SAVE 530
*SCREENSAVE 531
*SPOOL 532
*SPOOLON 533
*SYS 534
*TEMPO 536
*TIMER 537
*TV 538
*TYPE 539
*UNLOCK 540

? 116
? operator, The

@ 367
@% system variable, The
@cmd$ variable, The
@dir$ and @lib$ variables, The
@tmp$ and @usr$ variables, The

^ operator, The

| operator, The

6

64-bit ('double') floating-point variables
64-bit integer numeric variables
6-bit RGB colours

8

8-bit RGB colours

A

About BBC BASIC
ABS
Access denied
Accessing star commands
Accessing the records
Accuracy lost
ACS
Add-in Utilities
Adding a menu bar
Adding a progress bar
Adding a trackbar
Adding an up-down control
Adding popup and sub-menus
Additional OS Interfaces
address book program, The
Address out of range
ADVAL
Bad directory 651
Bad FOR variable 653
Bad hex or binary 654
Bad key 655
Bad library 656
Bad MODE 657
Bad MOUSE variable 658
Bad name 659
Bad program 660
Bad string 661
Bad subscript 662
Bad use of array 663
Bad use of structure 664
Based-indexed operands 601
BASIC interpreter 33
Basic line drawing modes 169
BBC BASIC file access 551
BGET# 211
Binary Chop, The 596
Blast-through text 959
Block graphics 959
Boxes and buttons 878
BPUT# 213
Browse for Folder 780
BY 215
Byte constant 606
Byte numeric variables 94
Byte variable storage 729
C
Calendar functions 897
CALL 216
Callback routines 938
Calling the Windows API 613
Cannot allocate clipboard memory 636
Cannot allocate memory for file 636
Cannot allocate virtual memory 636
Cannot append program 636
Cannot create character map 636
Cannot create file 636
Cannot create palette 636
Cannot create screen bitmap 636
Cannot create temporary buffer 636
Cannot crunch file 636
Cannot determine encoding 636
Cannot embed file 636
Cannot find BBCTUTOR.CHM 636
Cannot find BBCWIN.CHM 636
Cannot find icon in file 636
Cannot find manifest 636
Cannot open clipboard 636
Cannot open file 636
Cannot open icon file 636
Cannot open temporary file 636
Cannot print program 636
Cannot process resource in file 636
Cannot read file 636
Cannot read icon file 636
Cannot read icon image from file 636
Cannot run Sign Tool 636
Cannot update resources in file 636
Cannot write file 636
Cannot write icon 636
Can't match FOR 665
Caps Lock 42
Capturing the screen contents 190
CASE 221
Centering a hardcopy printout on the page 974
Chain 44
CHAIN 222
Changing the font 187
Changing the graphics origin 183
Changing the loop variable 112
Changing the Print Control Variable 367
Changing the process priority 796
Changing the timer period 345
Changing the window title 743
Channel/Effects 401
Character data files 559
Character dot pattern 187
Character encoding 42
Character set (MODE 7) 954
Checking for a sound card 752
Checking for input focus 811
Choose Colour 780
Choose Font 780
CHR$ 224
CIRCLE 226
CLEAR 227
Clear graphics viewport 183
Clear text viewport 183
CLG 230
CLOSE# 228
Closing a serial port 825
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLS</td>
<td>231</td>
</tr>
<tr>
<td>COLOUR</td>
<td>232</td>
</tr>
<tr>
<td>Coloured text (MODE 7)</td>
<td>957</td>
</tr>
<tr>
<td>Colours</td>
<td>163</td>
</tr>
<tr>
<td>COM automation</td>
<td>919</td>
</tr>
<tr>
<td>COM hierarchy and the Object Browser</td>
<td>919</td>
</tr>
<tr>
<td>Comments</td>
<td>599</td>
</tr>
<tr>
<td>Compatibility considerations</td>
<td>33</td>
</tr>
<tr>
<td>Compatibility limitations</td>
<td>195</td>
</tr>
<tr>
<td>Compatibility summary</td>
<td>195</td>
</tr>
<tr>
<td>Compatible data files</td>
<td>559</td>
</tr>
<tr>
<td>Compile</td>
<td>54</td>
</tr>
<tr>
<td>Compiler</td>
<td>33</td>
</tr>
<tr>
<td>Compiler directives</td>
<td>54</td>
</tr>
<tr>
<td>Conditional assembly</td>
<td>621</td>
</tr>
<tr>
<td>Conditional assembly and macros</td>
<td>621</td>
</tr>
<tr>
<td>Context Menu</td>
<td>79</td>
</tr>
<tr>
<td>Control codes</td>
<td>955</td>
</tr>
<tr>
<td>Control codes summary</td>
<td>955</td>
</tr>
<tr>
<td>Converting data files</td>
<td>826</td>
</tr>
<tr>
<td>Copy</td>
<td>49</td>
</tr>
<tr>
<td>Copy key editing</td>
<td>153</td>
</tr>
<tr>
<td>copy key,The</td>
<td>153</td>
</tr>
<tr>
<td>Copy rectangular block</td>
<td>169</td>
</tr>
<tr>
<td>Copying a structure</td>
<td>104</td>
</tr>
<tr>
<td>COS</td>
<td>234</td>
</tr>
<tr>
<td>COUNT</td>
<td>235</td>
</tr>
<tr>
<td>Creating a screen saver</td>
<td>816</td>
</tr>
<tr>
<td>Creating a scroll bar</td>
<td>797</td>
</tr>
<tr>
<td>Creating a status bar</td>
<td>778</td>
</tr>
<tr>
<td>Creating a toolbar</td>
<td>777</td>
</tr>
</tbody>
</table>
Define text viewport 183
Define word - DW 606
DEG 240
Delete 49
Deleting and inserting menu items 776
Designing the file 570
Determining when a button is clicked 848
Determining when a property sheet button is pressed 872
Device fault 667
Device unavailable 668
Dialogue boxes 848
Differences from BBC BASIC (86) 31
Differences from Intel syntax 601
DIM 241
DIM as a function 241
DIM space 669
Dimensioning arrays 241
Direct port input/output 830
Direct3D graphics 900
Directory was changed 636
Disabling a menu item 774
Disabling and enabling dialogue box items 848
Disabling Windows XP Visual Styles 980
Discovering an 'unknown error' 757
Discovering the Windows version 805
Discussion Group 77
Disk data files 552
Disk fault 670
Disk file commands 552
Disk full 671
Disk read only 672
Display modes 159
Displaying a message box 746
Displaying enhanced metafiles 793
Displaying GIF and JPEG images 801
Displaying icon files 812
Distributing the BBCWIN font 810
DIV 246
Division by zero 673

D
Double height 962
Downloading files from the internet 819
Drag and Drop 79
DRA\\m 247
Draw circle 169
Draw circular arc 169
Draw outline ellipse 169
Draw relative 169
Draw solid disc 169
Draw solid ellipse 169
Draw solid sector 169
Draw solid segment 169
Drawing a line 169
Drawing angled text 818
Drawing graphics to a printer 765
Drawing on the screen 169
Duration 401

E
Edit menu 49
editing pane,The 79
Editor and compiler errors 636
Efficient cache usage 614
<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELLIPSE</td>
<td>248</td>
</tr>
<tr>
<td>ELSE</td>
<td>249</td>
</tr>
<tr>
<td>Email</td>
<td>77</td>
</tr>
<tr>
<td>Embedded files</td>
<td>54</td>
</tr>
<tr>
<td>Enabling the text cursor</td>
<td>332</td>
</tr>
<tr>
<td>END</td>
<td>251</td>
</tr>
<tr>
<td>ENDCALL</td>
<td>252</td>
</tr>
<tr>
<td>ENDIF</td>
<td>253</td>
</tr>
<tr>
<td>Ending a definition</td>
<td>140</td>
</tr>
<tr>
<td>ENDPAR</td>
<td>254</td>
</tr>
<tr>
<td>ENDWHILE</td>
<td>255</td>
</tr>
<tr>
<td>Enhanced line drawing modes</td>
<td>169</td>
</tr>
<tr>
<td>ENVELOPE</td>
<td>256</td>
</tr>
<tr>
<td>EOF#</td>
<td>258</td>
</tr>
<tr>
<td>EOR</td>
<td>259</td>
</tr>
<tr>
<td>ERL</td>
<td>261</td>
</tr>
<tr>
<td>ERR</td>
<td>262</td>
</tr>
<tr>
<td>ERROR</td>
<td>263</td>
</tr>
<tr>
<td>Error handling</td>
<td>131</td>
</tr>
<tr>
<td>Error reporting</td>
<td>131</td>
</tr>
<tr>
<td>Error trapping commands</td>
<td>131</td>
</tr>
<tr>
<td>Error trapping examples</td>
<td>131</td>
</tr>
<tr>
<td>Errors in star commands</td>
<td>483</td>
</tr>
<tr>
<td>Escape</td>
<td>674</td>
</tr>
<tr>
<td>EVAL</td>
<td>265</td>
</tr>
<tr>
<td>Example programs</td>
<td>26</td>
</tr>
<tr>
<td>Executable</td>
<td>54</td>
</tr>
<tr>
<td>Exit</td>
<td>44</td>
</tr>
<tr>
<td>EXIT</td>
<td>267</td>
</tr>
<tr>
<td>EXP</td>
<td>269</td>
</tr>
<tr>
<td>Exponent range</td>
<td>675</td>
</tr>
</tbody>
</table>
Expression priority 90
EXT# 270
Extending the assembler 947
Extending the language 140

F
Facebook Page 77
FALSE 272
File basics 546
File buffering 551
File exists 676
File is not a valid resource file 636
File menu 44
File Open and File Save 780
File opening functions 552
File or path not found 677
File organisation 586
File specifiers 483
Filenames 552
Files in BBC BASIC 549
FILING 26
FILL 274
Find 49
Find Next 49
Finding special folders 807
Finding the current font name 809
Finding the display size 745
Finding the filename 756
Finding the size of a structure 104
Fixed strings 729
Fixing the window size 760
Flashing 961
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FN_newdialog</td>
<td>848</td>
</tr>
<tr>
<td>FN_newpropsheet</td>
<td>872</td>
</tr>
<tr>
<td>FN_nextToken</td>
<td>944</td>
</tr>
<tr>
<td>FN_ontimer</td>
<td>942</td>
</tr>
<tr>
<td>FN_readdate</td>
<td>897</td>
</tr>
<tr>
<td>FN_readlinesocket</td>
<td>910</td>
</tr>
<tr>
<td>FN_readsocket</td>
<td>910</td>
</tr>
<tr>
<td>FN_repEnt</td>
<td>944</td>
</tr>
<tr>
<td>FN_setproc</td>
<td>878</td>
</tr>
<tr>
<td>FN_skipTo</td>
<td>944</td>
</tr>
<tr>
<td>FN_skipToRet</td>
<td>944</td>
</tr>
<tr>
<td>FN_socketerror</td>
<td>910</td>
</tr>
<tr>
<td>FN_sortinit(dir%,smode%)</td>
<td>907</td>
</tr>
<tr>
<td>FN_split</td>
<td>930</td>
</tr>
<tr>
<td>FN_staticbox</td>
<td>878</td>
</tr>
<tr>
<td>FN_tcpcconnect</td>
<td>910</td>
</tr>
<tr>
<td>FN_tcpplisten</td>
<td>910</td>
</tr>
<tr>
<td>FN_title</td>
<td>930</td>
</tr>
<tr>
<td>FN_tobase</td>
<td>930</td>
</tr>
<tr>
<td>FN_today</td>
<td>897</td>
</tr>
<tr>
<td>FN_trackbarpos</td>
<td>866</td>
</tr>
<tr>
<td>FN_trim</td>
<td>930</td>
</tr>
<tr>
<td>FN_upper</td>
<td>930</td>
</tr>
<tr>
<td>FN_writelinesocket</td>
<td>910</td>
</tr>
<tr>
<td>FN_writesocket</td>
<td>910</td>
</tr>
<tr>
<td>FN_year</td>
<td>897</td>
</tr>
<tr>
<td>FNget and FNget$</td>
<td>936</td>
</tr>
<tr>
<td>FNinkey(t%) and FNinkey$(t%)</td>
<td>936</td>
</tr>
<tr>
<td>FNinput</td>
<td>936</td>
</tr>
<tr>
<td>FNlower(string$)</td>
<td>889</td>
</tr>
<tr>
<td>FNupper(string$)</td>
<td>889</td>
</tr>
<tr>
<td>Function/procedure body,The</td>
<td>140</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----</td>
</tr>
<tr>
<td>function/procedure body,The</td>
<td>140</td>
</tr>
<tr>
<td>GAMES</td>
<td>26</td>
</tr>
<tr>
<td>GCOL</td>
<td>279</td>
</tr>
<tr>
<td>GENERAL</td>
<td>26</td>
</tr>
<tr>
<td>General considerations</td>
<td>191</td>
</tr>
<tr>
<td>GET/GET$</td>
<td>281</td>
</tr>
<tr>
<td>GOSUB</td>
<td>286</td>
</tr>
<tr>
<td>GOTO</td>
<td>287</td>
</tr>
<tr>
<td>GRAPHICS</td>
<td>26</td>
</tr>
<tr>
<td>Graphics colours</td>
<td>163</td>
</tr>
<tr>
<td>Graphics viewport</td>
<td>183</td>
</tr>
<tr>
<td>graphics viewport,The</td>
<td>169</td>
</tr>
<tr>
<td>Hardcopy output to a printer</td>
<td>155</td>
</tr>
<tr>
<td>Held graphics</td>
<td>959</td>
</tr>
<tr>
<td>Help menu</td>
<td>77</td>
</tr>
<tr>
<td>Help Topics</td>
<td>77</td>
</tr>
<tr>
<td>High quality sound patch</td>
<td>950</td>
</tr>
<tr>
<td>High speed timers</td>
<td>942</td>
</tr>
<tr>
<td>HIMEM</td>
<td>288</td>
</tr>
</tbody>
</table>
Horizontal line fill to background right 169
Horizontal line fill to foreground 169
Horizontal line fill to non-background 169
Horizontal line fill to non-foreground right 169
How data is read/written 549
How data is stored 549
How files are referred to 549
How strings are stored 549
How the assembler works 617
HTML Help not installed 636

IF 290
Immediate Mode 73
Implementation 953
Incorrect arguments 678
Indentation 67
index, The 586
Indexed data files 586
Indexed database example 586
Indexed files 546
Indexed memory operands 601
Indirect procedure and function calls 140
Indirection 116
Initialising arrays 93
Initialising the contents of a dialogue box 848
Initialising the contents of a property sheet 872
Initialising the scroll bar 797
INKEY negative argument 293
INKEY positive or zero argument 293
INKEY/INKEY$ 293
INPUT 296
<table>
<thead>
<tr>
<th>Category</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input editing</td>
<td>151</td>
</tr>
<tr>
<td>INPUT LINE</td>
<td>298</td>
</tr>
<tr>
<td>INPUT#</td>
<td>299</td>
</tr>
<tr>
<td>Input/output using InpOut32</td>
<td>830</td>
</tr>
<tr>
<td>Insert</td>
<td>44</td>
</tr>
<tr>
<td>Insert/Overtype</td>
<td>42</td>
</tr>
<tr>
<td>INSTALL</td>
<td>301</td>
</tr>
<tr>
<td>INSTR</td>
<td>303</td>
</tr>
<tr>
<td>Insufficient memory</td>
<td>636</td>
</tr>
<tr>
<td>Insufficient memory to crunch file</td>
<td>636</td>
</tr>
<tr>
<td>Insufficient memory to crunch program</td>
<td>636</td>
</tr>
<tr>
<td>INT</td>
<td>304</td>
</tr>
<tr>
<td>Integer numeric variables</td>
<td>94</td>
</tr>
<tr>
<td>Integer variable storage</td>
<td>729</td>
</tr>
<tr>
<td>Invalid channel</td>
<td>679</td>
</tr>
<tr>
<td>Invalid file format</td>
<td>636</td>
</tr>
<tr>
<td>J</td>
<td></td>
</tr>
<tr>
<td>Jump out of range</td>
<td>680</td>
</tr>
<tr>
<td>Jumps, calls and returns</td>
<td>601</td>
</tr>
<tr>
<td>K</td>
<td></td>
</tr>
<tr>
<td>Keyboard commands</td>
<td>79</td>
</tr>
<tr>
<td>Keyword tokens</td>
<td>629</td>
</tr>
<tr>
<td>Keywords</td>
<td>125</td>
</tr>
<tr>
<td>L</td>
<td></td>
</tr>
<tr>
<td>Labelling program lines</td>
<td>156</td>
</tr>
<tr>
<td>Labels</td>
<td>599</td>
</tr>
<tr>
<td>Leaving program loops</td>
<td>112</td>
</tr>
<tr>
<td>LEFT$</td>
<td>306</td>
</tr>
<tr>
<td>LEN</td>
<td>308</td>
</tr>
<tr>
<td>Length of reserved memory</td>
<td>614</td>
</tr>
<tr>
<td>LET</td>
<td>309</td>
</tr>
</tbody>
</table>
MOUSE RECTANGLE l,b,w,h

MOUSE TO x,y

MOVE

Move rectangular block

Moving the graphics cursor

Moving the loop into a procedure

Multi-line IF statement

Multilingual (Unicode) text output

Multiple Document Interface

Multiple label

Multiple output windows

Names

Naming conventions

Negative root

Networking - shared files

New

NEXT

No coprocessor

No room

No such FN/PROC

No such font

No such line

No such printer

No such system call

No such variable

Non-paletted modes

NOT

Not enough room for file

Not in a FN or PROC

Not in a FOR loop
PRINT# 375
Printing coloured text 767
Printing in landscape format 768
PRIVATE 377
PROC 379
PROC_add 837
PROC_addtooltips 841
PROC_callmethod 919
PROC_cascade 893
PROC_checkbox 848
PROC_closedialog 848
PROC_closemdichild 893
PROC_closepropsheet 872
PROC_closesocket 910
PROC_closewin 934
PROC_closewindow 878
PROC_combobox 848
PROC_comexit 919
PROC_cominit 919
PROC_cominitlcid 919
PROC_dlgctrl 848
PROC_dlgitem 848
PROC_dot 837
PROC_editbox 848
PROC_exitmdi 893
PROC_exitsockets 910
PROC_exitsprites 885
PROC_exitXML 944
PROC_gdiparc 914
PROC_gdipbezier 914
PROC_gdipdeletebrush 914
PROC_gdipdeletepen 914
PROC_gdipexit 914
PROC_gdipinit 914
PROC_gdipline 914
PROC_gdippolybezier 914
PROC_gdippolygon 914
PROC_gdippolyline 914
PROC_gdipsector 914
PROC_groupbox 848
PROC_hqinit 950
PROC_initmdi 893
PROC_initsockets 910
PROC_initXML 944
PROC_invert 837
PROC_killtimer 942
PROC_listbox 848
PROC_movesprite 885
PROC_mul 837
PROC_multiwin 934
PROC_pushbutton 848
PROC_putvalue 919
PROC_radiobutton 848
PROC_release 900
PROC_releaseobject 919
PROC_removefloating toolbar 866
PROC_removeprogressbar 866
PROC_removestatusbar 841
PROC_removetoolbar 841
PROC_removetrackbar 866
PROC_render 900
PROC_selectwin 934
Program storage in memory 728
Program structure limitations 112
Property sheets and wizards 872
Pseudo-variables 94
PTR# 381
Putting an icon in the SysTray 813

Q
Query as a byte variable 116
QUIT 382

R
RAD 383
Random (relative) files 570
Random access files 546
Random access inventory program 570
READ 384
Read character dot pattern 542
Read current character 542
READ# 385
Reading a compatible data file 559
Reading a mixed data file 559
Reading a NUL-terminated string 755
Reading a pixel's colour 169
Reading from a file 281
Reading from a serial port 824
Reading serial character data 559
Reading the command line 755
Reading the contents of a dialogue box 848
Reading the contents of a property sheet 872
Reading the keyboard 281
Reading the screen 281
Reasons for trapping errors 131
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recent files</td>
<td>44</td>
</tr>
<tr>
<td>Record structure</td>
<td>570</td>
</tr>
<tr>
<td>RECTANGLE</td>
<td>386</td>
</tr>
<tr>
<td>Rectangle plot and fill</td>
<td>169</td>
</tr>
<tr>
<td>Recursive functions/procedures</td>
<td>140</td>
</tr>
<tr>
<td>Redo</td>
<td>49</td>
</tr>
<tr>
<td>Reducing stack usage</td>
<td>726</td>
</tr>
<tr>
<td>Re-entrant ON MOVE interrupts</td>
<td>976</td>
</tr>
<tr>
<td>Registry settings</td>
<td>86</td>
</tr>
<tr>
<td>REM</td>
<td>388</td>
</tr>
<tr>
<td>Removing the title bar</td>
<td>763</td>
</tr>
<tr>
<td>Renumber</td>
<td>54</td>
</tr>
<tr>
<td>REPEAT</td>
<td>389</td>
</tr>
<tr>
<td>REPEAT ... UNTIL loops</td>
<td>112</td>
</tr>
<tr>
<td>Replace</td>
<td>49</td>
</tr>
<tr>
<td>REPORT/REPORT$</td>
<td>390</td>
</tr>
<tr>
<td>Repositioning the window</td>
<td>758</td>
</tr>
<tr>
<td>Reserving a temporary area of memory</td>
<td>241</td>
</tr>
<tr>
<td>Reserving an area of memory</td>
<td>241</td>
</tr>
<tr>
<td>Reserving memory</td>
<td>614</td>
</tr>
<tr>
<td>RESTORE</td>
<td>391</td>
</tr>
<tr>
<td>RESTORE DATA</td>
<td>391</td>
</tr>
<tr>
<td>RESTORE ERROR</td>
<td>391</td>
</tr>
<tr>
<td>RESTORE LOCAL</td>
<td>391</td>
</tr>
<tr>
<td>RETURN</td>
<td>393</td>
</tr>
<tr>
<td>Returning arrays from procedures and functions</td>
<td>140</td>
</tr>
<tr>
<td>Returning structures from procedures and functions</td>
<td>104</td>
</tr>
<tr>
<td>Reveal/conceal</td>
<td>963</td>
</tr>
<tr>
<td>RIGHT$</td>
<td>395</td>
</tr>
<tr>
<td>RND</td>
<td>397</td>
</tr>
<tr>
<td>Run</td>
<td>73</td>
</tr>
</tbody>
</table>
RUN 398
Run menu 73
Run To Cursor 73
Running BASIC code in a separate file 216
Running BBC BASIC 22
Run-time errors 643

S
Save 44
Save As 44
Screen dimensions 169
Scroll bars 79
Scroll Lock 42
Scrolling the display 797
Scrolling the window 468
Segment override 601
Select All 49
Selecting the character set 187
Separated graphics 959
Serial (sequential) files 546
Serial files 559
Serial input/output 820
Set Colours 67
Set Font 67
Set Printer Font 67
Setting the correct data type 919
Setting the File Open dialogue’s initial directory 973
SGN 399
Sign Tool failed or aborted 636
Similarly named star commands 483
Simple random access database 570
Simple random access file 570
<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single line functions/procedures</td>
<td>140</td>
</tr>
<tr>
<td>Single-line IF statement</td>
<td>290</td>
</tr>
<tr>
<td>Sixteen colour modes: 2, 3, 6 and 8-31</td>
<td>163</td>
</tr>
<tr>
<td>Size mismatch</td>
<td>715</td>
</tr>
<tr>
<td>Size needed</td>
<td>716</td>
</tr>
<tr>
<td>Socket (network) connections</td>
<td>910</td>
</tr>
<tr>
<td>Solid (filled) shapes</td>
<td>191</td>
</tr>
<tr>
<td>Sorting data arrays</td>
<td>907</td>
</tr>
<tr>
<td>SOUND</td>
<td>401</td>
</tr>
<tr>
<td>SOUNDS</td>
<td>26</td>
</tr>
<tr>
<td>SPC</td>
<td>406</td>
</tr>
<tr>
<td>Specification</td>
<td>24</td>
</tr>
<tr>
<td>Sprites</td>
<td>885</td>
</tr>
<tr>
<td>SQR</td>
<td>407</td>
</tr>
<tr>
<td>Star command summary</td>
<td>486</td>
</tr>
<tr>
<td>Starting a definition</td>
<td>140</td>
</tr>
<tr>
<td>Statement separators</td>
<td>88</td>
</tr>
<tr>
<td>Statements</td>
<td>728</td>
</tr>
<tr>
<td>Static variables</td>
<td>94</td>
</tr>
<tr>
<td>Status Bar</td>
<td>67</td>
</tr>
<tr>
<td>status bar, The</td>
<td>42</td>
</tr>
<tr>
<td>Step</td>
<td>73</td>
</tr>
<tr>
<td>STEP</td>
<td>408</td>
</tr>
<tr>
<td>STEP cannot be zero</td>
<td>717</td>
</tr>
<tr>
<td>Step Line</td>
<td>73</td>
</tr>
<tr>
<td>Stop</td>
<td>73</td>
</tr>
<tr>
<td>STOP</td>
<td>409</td>
</tr>
<tr>
<td>STR$</td>
<td>410</td>
</tr>
<tr>
<td>STR$ format control</td>
<td>367</td>
</tr>
<tr>
<td>Straight lines</td>
<td>191</td>
</tr>
<tr>
<td>Text positioning</td>
<td>187</td>
</tr>
<tr>
<td>------------------</td>
<td>-----</td>
</tr>
<tr>
<td>Text viewport</td>
<td>183</td>
</tr>
<tr>
<td>THEN</td>
<td>419</td>
</tr>
<tr>
<td>Ticking a menu item</td>
<td>773</td>
</tr>
<tr>
<td>TIME</td>
<td>421</td>
</tr>
<tr>
<td>TIME$</td>
<td>422</td>
</tr>
<tr>
<td>Timing program execution</td>
<td>753</td>
</tr>
<tr>
<td>TINT</td>
<td>424</td>
</tr>
<tr>
<td>title bar,The</td>
<td>39</td>
</tr>
<tr>
<td>TO</td>
<td>425</td>
</tr>
<tr>
<td>To allow cleanup operations to take place</td>
<td>131</td>
</tr>
<tr>
<td>To allow execution to continue</td>
<td>131</td>
</tr>
<tr>
<td>To ensure the error message is visible</td>
<td>131</td>
</tr>
<tr>
<td>To make your program more 'friendly'</td>
<td>131</td>
</tr>
<tr>
<td>Too many files selected</td>
<td>636</td>
</tr>
<tr>
<td>Too many open files</td>
<td>720</td>
</tr>
<tr>
<td>Toolbar</td>
<td>67</td>
</tr>
<tr>
<td>toolbar,The</td>
<td>41</td>
</tr>
<tr>
<td>Toolbars and status bars</td>
<td>841</td>
</tr>
<tr>
<td>TOOLS</td>
<td>26</td>
</tr>
<tr>
<td>TOP</td>
<td>427</td>
</tr>
<tr>
<td>Trace</td>
<td>54</td>
</tr>
<tr>
<td>TRACE</td>
<td>428</td>
</tr>
<tr>
<td>Trackbars and progress bars</td>
<td>866</td>
</tr>
<tr>
<td>TrackPopupMenu</td>
<td>966</td>
</tr>
<tr>
<td>Transferring files from a BBC Micro</td>
<td>826</td>
</tr>
<tr>
<td>Transferring the files</td>
<td>826</td>
</tr>
<tr>
<td>Trappable errors - BASIC</td>
<td>643</td>
</tr>
<tr>
<td>Trappable errors - OS</td>
<td>643</td>
</tr>
<tr>
<td>Trapped OPENIN, OPENOUT or OPENUP</td>
<td>964</td>
</tr>
<tr>
<td>Triangle plot and fill</td>
<td>169</td>
</tr>
</tbody>
</table>
TRUE 429
Tutorial 77
Two colour modes: 0, 4 and 32 163
Type mismatch 721

U
Undo 49
Unicode 67
Unknown error 722
UNTIL 430
Untrappable errors 643
Updating the screen 748
Use as binary (dyadic) operators 116
User-defined characters 468
User-defined data types 104
User-defined modes 468
Using BASIC input/output 612
Using DATA in installed modules 975
Using dialogue boxes 780
Using DIM to reserve memory 614
Using multimedia timers 794
Using scroll bars 797
Using status bars and toolbars with the Multiple Document Interface 969
Using structures with the Windows API 104
Using system colours 788
Using the clipboard 779
Using the entire screen 764
Using the EXIT statement 112
Using the system registry 815
Using windows larger than 1920 x 1440 pixels 965
USR 432
Utilities menu 54
<table>
<thead>
<tr>
<th>Variable name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAL</td>
<td>434</td>
</tr>
<tr>
<td>Variable names</td>
<td>94</td>
</tr>
<tr>
<td>Variable storage in memory</td>
<td>729</td>
</tr>
<tr>
<td>Variable types</td>
<td>94</td>
</tr>
<tr>
<td>Variables</td>
<td>94</td>
</tr>
<tr>
<td>Variant numeric storage (40 bit)</td>
<td>729</td>
</tr>
<tr>
<td>Variant numeric storage (64 bit)</td>
<td>729</td>
</tr>
<tr>
<td>Variant numeric storage (80 bit)</td>
<td>729</td>
</tr>
<tr>
<td>Variant numeric variables</td>
<td>94</td>
</tr>
<tr>
<td>Variants and variant format</td>
<td>919</td>
</tr>
<tr>
<td>VDU</td>
<td>436</td>
</tr>
<tr>
<td>VDU 0</td>
<td>445</td>
</tr>
<tr>
<td>VDU 1</td>
<td>446</td>
</tr>
<tr>
<td>VDU 10</td>
<td>455</td>
</tr>
<tr>
<td>VDU 11</td>
<td>456</td>
</tr>
<tr>
<td>VDU 12</td>
<td>457</td>
</tr>
<tr>
<td>VDU 127</td>
<td>482</td>
</tr>
<tr>
<td>VDU 13</td>
<td>458</td>
</tr>
<tr>
<td>VDU 14</td>
<td>459</td>
</tr>
<tr>
<td>VDU 15</td>
<td>460</td>
</tr>
<tr>
<td>VDU 16</td>
<td>461</td>
</tr>
<tr>
<td>VDU 17</td>
<td>462</td>
</tr>
<tr>
<td>VDU 18</td>
<td>463</td>
</tr>
<tr>
<td>VDU 19</td>
<td>464</td>
</tr>
<tr>
<td>VDU 2</td>
<td>447</td>
</tr>
<tr>
<td>VDU 20</td>
<td>465</td>
</tr>
<tr>
<td>VDU 21</td>
<td>466</td>
</tr>
<tr>
<td>VDU 22</td>
<td>467</td>
</tr>
<tr>
<td>VDU 23</td>
<td>468</td>
</tr>
<tr>
<td>VDU 24</td>
<td>474</td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Writing serial character data</td>
<td>559</td>
</tr>
<tr>
<td>Writing text to the screen</td>
<td>187</td>
</tr>
<tr>
<td>Writing to a serial port</td>
<td>823</td>
</tr>
<tr>
<td>Z</td>
<td></td>
</tr>
<tr>
<td>Zone width</td>
<td>367</td>
</tr>
</tbody>
</table>